# Integration Testing Plan Document

**myTaxiService**

*Fabio Catania, Matteo Di Napoli, Nicola Di Nardo*

January 21, 2016

# Contents

# 1 Introduction

## 1.1 Purpose

The purpose of this Integration Testing Plan Document (ITPD) is to describe the process of testing the interfaces between the myTaxiService components in order to verify that they are properly assembled.

ITPD is a reference point document for every team member who cooperates in the integration testing process.

The integration tests described in this document are at the component level. The integration tests of lower level code modules are described in the corresponding components unit test. Unit tests are described in the UTP. Furthermore, structural quality testing is not an argument of this document.

Software structural quality refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly. To know more about that STP should be consulted.

## 1.2 Scope

The system aims at optimizing the taxi service in London. The software is intended to:

- integrate the current telephonic reservation system by offering the possibility to request a taxi through a web application and a mobile app;
- guarantee a fair management of the taxi queues;
- offer the possibility to share the taxi with other people. The goals of the application are illustrated more in detail in the section 1.5 Goal of the Requirements Analysis and Specifications Document.

## 1.3 Definitions, acronyms and abbreviations

### 1.3.1 Definitions

- Cloud Computing: a kind of Internet-based computing, where shared resources,data and information are provided to computers and other devices on-demand.
- Design Pattern: a general reusable solution to a commonly occurring problem within a given context in software design;
- Distributed system: a software system in which components located on networked computers communicate and coordinate their actions by passing messages;
- Software architecture: the high level structures of a software system and the documentation of these structures;

- Software Design: the process by which an agent creates a specification of a software artifact, intended to accomplish goals, using a set of primitive components and subject to constraints;
- Software Component: a unit of composition with contractually specified interfaces and explicit context dependencies only;
- Software Interface: a shared boundary across which two separate components of a computer system exchange information;
- Software Layer: a group of classes that have the same set of link-time module dependencies to other modules.
- Test Case: a set of conditions under which a tester will determine whether an application, software system or one of its features is working as it was originally established for it to do.

### 1.3.2 Acronyms

- API: Application Programming Interface
- DB: Data Base.
- DBMS: Data Base management system.
- DD: Design Document
- EJB: Enterprise Java Bean
- GPS: Global Position System
- GUI: Graphical User Interface
- HTML: HyperText Markup Language
- ICT: Information and Communication Technology
- IEEE: Institute of Electrical and Electronics Engineers
- JB:Java Bean
- LD: Logical Design
- MVC: Model-View-Controller
- OS: Operative System
- PaaS: Platform as a Service
- QoS: Quality of Service
- RASD: Requirements Analysis and Specification Document
- REST: Representational State Transfer
- UTP: Unit Test Plan.

## 1.4 References

- RASD_definitive.pdf
- Design Document.pdf
- Assignments 4  integration test plan.pdf
- Integration Test Plan Example.pdf

## 1.5 Document Overview

This document is structured following the specification by the *Assignment 4 integration test plan.pdf* document.

It is essentially organized in five parts:

- Section 1: Introduction, it gives a description of the document and it is thought as supporting material for its lecture;
- Section 2: Integration strategy, it identifies the components to be integrated and describes the their integration testing approach;
- Section 3: Individual Steps and Test Description, it describes the tests used to verify that the elements integrated perform as expected;
- Section 4: Tools and Test Equipment Required, it identifies all tools and test equipment needed to accomplish the integration;
- Section 5: Program Stubs and Test Data Required, it is based on the testing strategy and on the test design and it identifies any program stubs or special test data required for each integration step.

# 2 Integration Strategy

## 2.1 Entry and Exit Criteria

Before the integration testing starts, should be met the following conditions:

- the RASD must be completed and approved;
- the DD must be completed and approved;
- the components to be integrated must be finished;
- the units in the components to be integrated must be already successfully tested and the UTP must be delivered.

A unit is intended as a group of methods implementing a single small functionality.

Issues not covered within this document include the unit testing document. This area will be addressed at a greater depth as more detailed design of the system takes place.

In addition, before a specific integration test can begin, must be delivered:

- the drivers and/or the stubs necessary for this specific integration test (this is described below);
- the input data for this specific integration test.

The following items must be delivered when a specific integration test is finished:

- integration test report;
- integration test output data;
- problem reports (if necessary).

All the integration test reports must be delivered when testing on all code modules has finished.

## 2.2 Testing Tasks

The following tasks are necessary to perform a specific integration test:

- designing the integration test;
- designing a driver and/or stubs (if necessary);
- designing input test data;
- setting up a system, the components involved, the driver, the stubs and the input test data;
- performing the integration test.

## 2.3 Elements to be Integrated

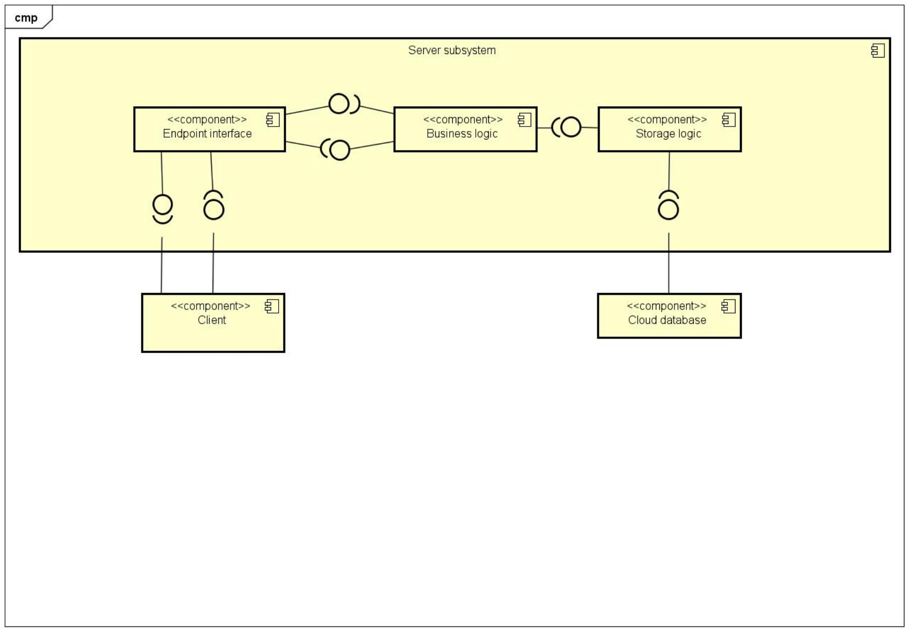
Making reference to what described in the DD, there is no pre-existent system to be integrated.

*myTaxiService* system is a distributed software developed from scratch and its production can be treated as the mutual integration of its subsystems. These are two: the server subsystem and the client subsystem.

### 2.3.1 Server Subsystem

The server used is provided as a service by Google and it is arranged in three components according to their function:

- the Endpoints interface, it exposes the services provided by the server to the client and communicate the received requests to the business logic;
- the business logic: it communicates with the endpoint interface and with the storage logic, is responsible for processing the User UI requests and to respond to them;
- the storage logic: it communicates with the business logic and takes care of the Google Cloud database calls.



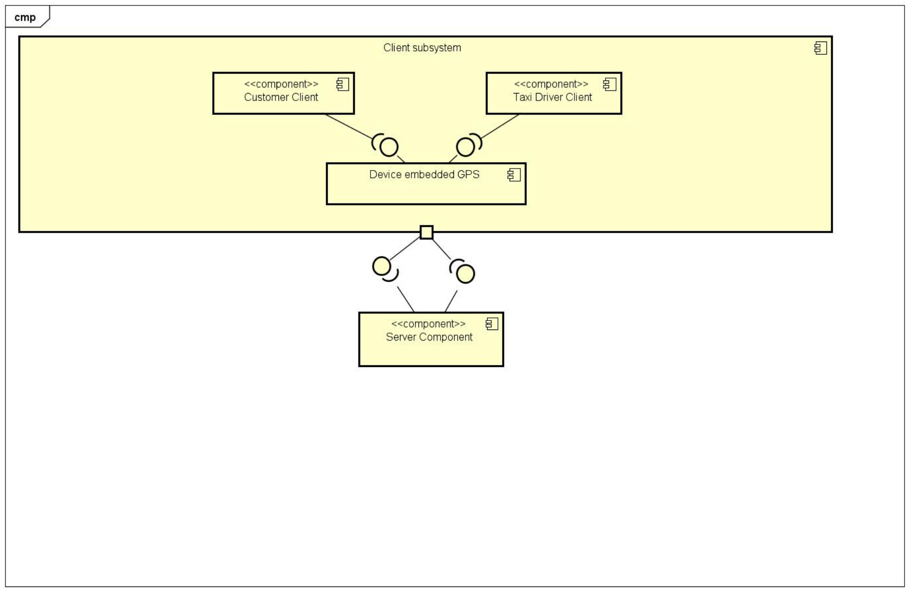
### 2.3.2 Client Subsystem

The client subsystem is responsible for the interaction with the user such as getting UI requests, displaying their answers and for the communication of that requests to the server.

In addition, in order to execute its tasks, the subsystem must integrate the already existing GPS (since the assumption [D14] of the RASD it is assumed to be valid) of the device of the client to track his position.

The client subsystem is organized in two components:

- the customer client: it refers to the interaction with the customer;
- the taxi driver client: it regards the interaction with the taxi driver.



## 2.4 Integration Testing Strategy

During the integration testing phase, all the methods implementing the interfaces between the components and the subsystems should be ideally tested in order to verify that they are properly assembled.

Integration tests will be constructed following the black-box approach, because with this technique missing functionalities in the system can be easily identified.

Before each test starts, for every test case must be drawn a state diagram to represent the components in case with their functionalities. The diagram is composed by states and transitions. Each transition can be thought as a *precondition-postcondition* pair. As coverage criterion transition coverage is chosen and the minimum approval rate is fixed to 90

To cover the whole system with the integration testing activity, a **sandwich strategy** will be used. It is a structural oriented integration strategy and a combination of the bottom-up and the top-down approaches.

In particular, the **bottom-up** approach will be used for the integration verification of the components in the server and in the client subsystems and the **top-down** way will be used during the subsystem integration testing.

## 2.5 Sequence of Components Integration

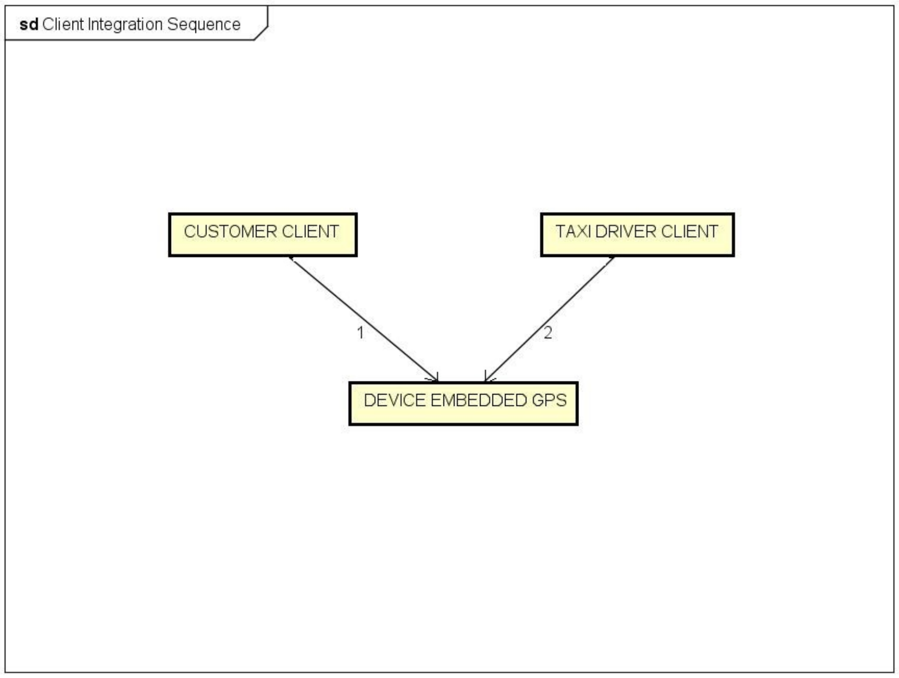
As just specified, the integration activity and consequently the integration testing are organized in two phases:

- integration among the components in the subsystems;
- integration among the subsystems.

In this section is indicated the integration sequence in these two phases. The numbers above arrows represent the order of integration testing.

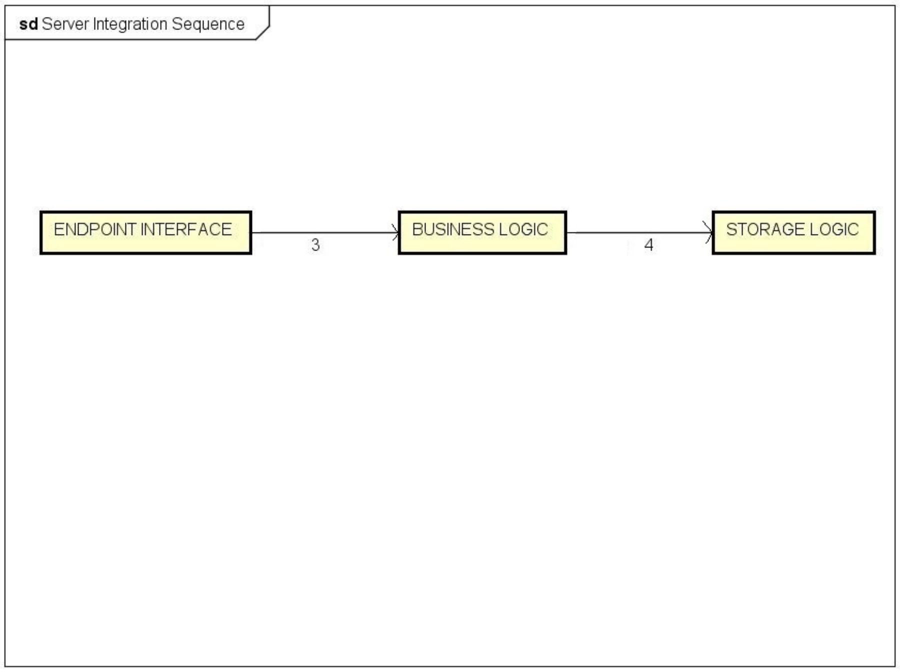
### 2.5.1 Sequence of Client Components Integration

| ID | Integration Test | Paragraphs |
|---|---|---|
| I1 | User Client → Embedded GPS | 3.1.1 3.2.1 |
| I2 | Taxi Driver Client → Embedded GPS | 3.1.2 3.2.1 |



### 2.5.2 Sequence of Server Components Integration

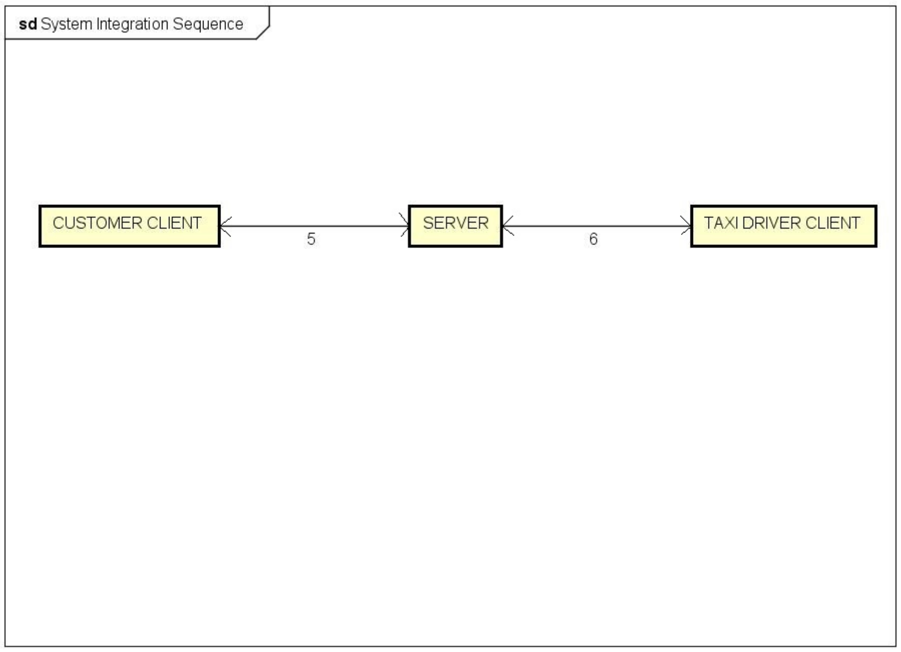
| ID | Integration Test | Paragraphs |
|---|---|---|
| I3 | Endpoint Interface → Business Logic | 3.1.3 3.2.2 |
| I4 | Business Logic → Storage Logic | 3.1.4 3.2.2 |



### 2.5.3 Sequence of System Components Integration

In this case is represented the integration between the server and client as two separated integrations between the server and the customer client and the server and the taxi driver client. This choice just to highlight the adopted integration order.

| ID | Integration Test | Paragraphs |
|---|---|---|
| I5 | Server → Customer Client | 3.1.5 3.2.3 |
| I6 | Server → Taxi Driver Client | 3.1.6 3.2.3 |



# 3 Individual Steps and Test Description

## 3.1 Test Case Specifications

### 3.1.1 Integration test case I1

| | |
|---|---|
| **Test Case Identifier** | I1T1 |
| **Test Item(s)** | User Client → Embedded GPS |
| **Input Specification** | Position request. Create a typical User input requesting his position. |
| **Output Specification** | Check if the correct method is called in the GPS interface. |
| **Environmental Needs** | GPS interface stub<br>GPS interface unit tested |

### 3.1.2 Integration test case I2

| | |
|---|---|
| **Test Case Identifier** | I2T1 |
| **Test Item(s)** | Taxi Driver Client → Embedded GPS |
| **Input Specification** | Position request. Create a typical Taxi Driver input getting his position. |
| **Output Specification** | Check if the correct method is called in the GPS interface. |
| **Environmental Needs** | GPS interface stub<br>GPS interface unit tested |

### 3.1.3 Integration test case I3

| | |
|---|---|
| **Test Case Identifier** | I3T1 |
| **Test Item(s)** | Endpoint Interface → Business Logic |
| **Input Specification** | It organizes the received user input (Regular Call Request) and provide them to the logic controller |
| **Output Specification** | Check if the correct methods are called in the Business Logic module |
| **Environmental Needs** | Business Logic driver<br>Endpoint Interface module unit tested<br>Business Logic module unit tested |

| | |
|---|---|
| **Test Case Identifier** | I3T2 |
| **Test Item(s)** | Endpoint Interface → Business Logic |
| **Input Specification** | It organizes the received user input (Reservation Call Request) and provide them to the logic controller. |
| **Output Specification** | Check if the correct methods are called in the Business Logic module. |
| **Environmental Needs** | Business Logic driver<br>Endpoint Interface module unit tested<br>Business Logic module unit tested |

| | |
|---|---|
| **Test Case Identifier** | I3T3 |
| **Test Item(s)** | Endpoint Interface → Business Logic |
| **Input Specification** | It organizes the received user input (Shared Call Request) and provide them to the logic controller. |
| **Output Specification** | Check if the correct methods are called in the Business Logic module. |
| **Environmental Needs** | Business Logic driver<br>Endpoint Interface module unit tested<br>Business Logic module unit tested |

| | |
|---|---|
| **Test Case Identifier** | I3T4 |
| **Test Item(s)** | Endpoint Interface → Business Logic |
| **Input Specification** | It organizes the received taxi driver input (e.g. Declare Availability) and provide them to the logic controller. |
| **Output Specification** | Check if the correct methods are called in the Business Logic module |
| **Environmental Needs** | Business Logic driver<br>Endpoint Interface module unit tested<br>Business Logic module unit tested |

### 3.1.4 Integration test case I4

| | |
|---|---|
| **Test Case Identifier** | I4T1 |
| **Test Item(s)** | Business Logic $\rightarrow$ Storage Interface |
| **Input Specification** | Simulate the information read request of the Business Logic module |
| **Output Specification** | Check if the correct method is called in the Storage Interface module |
| **Environmental Needs** | Business Logic unit tested<br>Storage Interface unit tested |

### 3.1.5 Integration test case I5

| | |
|---|---|
| **Test Case Identifier** | I5T1 |
| **Test Item(s)** | Customer Client $\rightarrow$ Server |
| **Input Specification** | Generate and send to the Application Server typical Client Data with a Regular Call request to be handled |
| **Output Specification** | Check if the Server Logic is able to handle the client request using the correct methods of his involved components |
| **Environmental Needs** | Client and Server components already tested at a lower level<br>Client stub mocking the input |

| | |
|---|---|
| **Test Case Identifier** | I5T2 |
| **Test Item(s)** | Customer Client $\rightarrow$ Server |
| **Input Specification** | Generate and send to the Application Server typical Client Data with a Reservation Call request to be handled |
| **Output Specification** | Check if the Server Logic is able to handle the client request using the correct methods of his involved components |
| **Environmental Needs** | Client and Server components already tested at a lower level<br>Client stub mocking the input |

| | |
|---|---|
| **Test Case Identifier** | I5T3 |
| **Test Item(s)** | Customer Client → Server |
| **Input Specification** | Generate and send to the Application Server typical Client Data with many and different Shared Call requests to be handled, each associated randomly with different city zones |
| **Output Specification** | Check if the Server Logic is able to handle the client requests using the correct methods of his involved components<br>Check if server logic couple the Shared Call Request following the specification of the involved algorithm |
| **Environmental Needs** | Client and Server components already tested at a lower level<br>Client stub mocking the input |

| | |
|---|---|
| **Test Case Identifier** | I5T4 |
| **Test Item(s)** | Customer Client → Server |
| **Input Specification** | Generate and send to the Application Server typical Client Data with duplicate identical requests from the same device, of each kind of available requests. |
| **Output Specification** | Check if the Server Logic is able to handle the client requests using the correct methods of his involved components<br>Check if server logic is able to recognize duplicate requests and handle them only a single time |
| **Environmental Needs** | Client and Server components already tested at a lower level<br>Client stub mocking the input |

| | |
|---|---|
| **Test Case Identifier** | I5T5 |
| **Test Item(s)** | Customer Client → Server |
| **Input Specification** | Generate and send to the Application Server **invalid** Client Data |
| **Output Specification** | Check if the Server Logic is able to recognize and handle correctly invalid input data when the client-side input validation for any reason fails |
| **Environmental Needs** | Client and Server components already tested at a lower level<br>Client stub mocking the input |

| | |
|---|---|
| **Test Case Identifier** | I5T6 |
| **Test Item(s)** | Server → Client Customer |
| **Input Specification** | Generate and send to the Customer Client typical input data representing a notification of accepted or rejected Call (Regular, Reservation or Shared) |
| **Output Specification** | Check if the Client is able to recognize and correctly display incoming notifications from the Server Side |
| **Environmental Needs** | Client and Server components already tested at a lower level<br>Server stub mocking the input |

### 3.1.6 Integration test case I6

| | |
|---|---|
| **Test Case Identifier** | I6T1 |
| **Test Item(s)** | Server → Taxi Driver Client |
| **Input Specification** | Generate and send to the Taxi Driver Client typical input data representing a notification of incoming Call to be accepted or rejected |
| **Output Specification** | Check if the Client is able to recognize and correctly display incoming notifications from the Server Side, also giving the opprtunity to interact to accept or reject them |
| **Environmental Needs** | Client and Server components already tested at a lower level<br>Server stub mocking the input |

| | |
|---|---|
| **Test Case Identifier** | I6T2 |
| **Test Item(s)** | Taxi Driver Client → Server |
| **Input Specification** | Generate and send to the Server typical data representing the acceptation of an incoming request |
| **Output Specification** | Check if the Server is able to recognize and correctly interpret the input calling the correct methods to manage it |
| **Environmental Needs** | Client and Server components already tested at a lower level<br>Taxi Driver Client stub mocking the input |

| | |
|---|---|
| **Test Case Identifier** | I6T3 |
| **Test Item(s)** | Taxi Driver Client → Server |
| **Input Specification** | Generate and send to the Server typical data representing the rejection of an incoming request |
| **Output Specification** | Check if the Server is able to recognize and correctly interpret the input calling the correct methods to manage it |
| **Environmental Needs** | Client and Server components already tested at a lower level<br>Taxi Driver Client stub mocking the input |

## 3.2 Test Procedures

### 3.2.1 Integration test procedure TP1

| | |
|---|---|
| **Test Procedure Identifier** | TP1 |
| **Purpose** | This test procedure verifies whether the client subsystem software:<br>- can handle mobile user position requests<br>- can handle browser user position requests<br>- can handle taxi driver position requests<br>- can output the correct information to the requesting client |
| **Procedure Steps** | I1-I2 or I2-I1 |

### 3.2.2 Integration test procedure TP2

| | |
|---|---|
| **Test Procedure Identifier** | TP2 |
| **Purpose** | This test procedure verifies whether the server subsystem software:<br>- can handle the Endpoint Interface inputs<br>- communicates the right parameters<br>to the Business Logic module (model)<br>- can handle the Business Logic inputs<br>- can output the right actions on the<br>Storage Interface<br>- can correctly output information to the Endpoint Interface |
| **Procedure Steps** | I3-I4-I5 |

#### 3.2.3 Integration test procedure TP3

| | |
|---|---|
| **Test Procedure Identifier** | TP3 |
| **Purpose** | This test procedure verifies whether the Server subsystem software:<br>- can handle incoming data both from Customer Client and the Taxi Driver Client<br>-manages to communicate and correctly send notifications to both kind of Clients<br>-offers an internal validation of data input |
| **Procedure Steps** | I5-I6 or I6-I5 |

# 4 Tools and Test Equipment Required

## 4.1 Test Environment

To operate all the identified test procedures three different test environments are required: for the mobile client applications an emulator of different devices could be preferred, for the web client application all the browsers identified in the RASD document are eligible environments and finally an application server instance on Google Cloud with its Endpoints interface is required for server testing.

## 4.2 Employed Tools

### 4.2.1 Mockito

Mockito provides a framework for interaction testing adding predefined response to Mock object methods. It has been used in the Server Subsystem and Client Subsystem internal integration tests.

### 4.2.2 Citrus Framework

Citrus Framework provides automated integration tests for message protocols and data formats. During the test run Citrus is able to act on both sides as client and/or server simulating request/response messages. It has been used in the integration between Client and Server subsystems.

# 5 Program Stubs and Test Data Required

In this part of the document it is identified all the required program stubs and/or drivers (scaffolding) in order to perform any single integration step, based on the adopted strategy.

Tools used in this part of the integration are listed in the previous chapter.

## 5.1 Integration test case I1 and I2

Working from the top level toward the bottom (top-down approach) in the Client subsystem means that a stub is needed to simulate the behaviour of the GPS component. This stub contains all the called and used modules at this step of integration between any kind of user and the Embedded GPS.

## 5.2 Integration test case I3

In the Server subsystem the integration starts from the leaves of the uses hierarchy that is the Endpoint Interface. This component organizes the received client inputs and provide them to the Business Logic, therefore a driver must be implemented to make this possible.

## 5.3 Integration test case I5

### 5.3.1 Test Cases 1,2,3,4,5

These test cases inspect the integration between Customer Client module and the Application Server, starting from the client toward server responses. A stub mocking typical client requests is implemented to make possible checking if the Application Server is able to correctly handle them.

### 5.3.2 Test Case 6

The last test case of this integration step starts from the Application Server which generates inputs to send to the Customer Client. Once again a stub is needed, but this time replacing the server. The stub generates and sends to the Customer Client typical input data representing any kind of notification

## 5.4 Integration test case I6

### 5.4.1 Test Case 1

The test case inspect the integration between the Application Server and the Taxi Driver Client starting from the server which generates inputs to send to the specific client. A server stub mocking these inputs is implemented. It generates and sends to the Taxi Driver Client typical data representing notifications, as an incoming Call to be accepted or rejected.

### 5.4.2 Test Cases 2,3

In these test cases the integration concerns the Application Server and the Taxi Driver Client again, but this time it is the Taxi Driver Client to generate inputs to be send to the Application Server. Therefore, a client stub mocking these inputs is implemented. It generates and sends to the Application Server typical data representing the acceptance or the rejection of an incoming request.