Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

---

myTaxiService
Integration Test Plan (ITP)

Supervisor:
Prof. Elisabetta Di Nitto

Students:
Alberto Gasparin
Vito Matarazzo

Year 2015-2016

Contents

Revision History

Date	Reason for changes	Version
January 20, 2016		v1.0

1 Introduction

1.1 Purpose and Scope

This document describes the Integration Test Plan (ITP) for the myTaxiService project. It describes the necessary tests to verify that all of the components of system are properly assembled. Integration testing is performed after unit testing and it becomes necessary to verify the software modules work in unity. In that phase of software verification we focus mainly on the interfaces and flow of data between the modules, so we give priority to those aspects rather then single unit functions (that are already been tested).

1.2 Definitions, Acronyms, Abbreviations

1.2.1 Definition

Driver: a program that simulate a module that calls the interface procedures of the module being tested.

Fault: wrong or missing function in the code.

Failure: the manifestation of a fault during execution.

Malfunction: according to its specification the system does not meet its specified functionality.

Stub: a program that has the same interface procedures as a module that is being called by the module being tested but is simpler.

1.2.2 Acronyms

DD: Design Document

DB: Database

ITP: Integration Test Plan

RASD: Requirement Analysis and Specification Document

1.3 Reference Document

Specification Document: myTaxiService Project AA 2015-2016.pdf.

RASD, Alberto Gasparin, Vito Matarazzo, v1.1, November 2015.

DD, Alberto Gasparin, Vito Matarazzo, v1.0, December 2015.

Mockito documentation (http://site.mockito.org)

Arquillan documentation (http://arquillian.org)

2 Integration Strategy

2.1 Entry Criteria

Here we state those criteria that have to be fulfilled in order to proceed with integration testing of single elements:

- All modules/components have been unit-tested.
- The components involved have been identified. [Section 2.2 of this document]
- Stubs needed by the specific integration test have been coded. [Section 5 of this document]
- Input test data needed by the specific integration test have been identified [Section 5 of this document]
- Critical modules have been identified. These need to be tested on priority.
- Test cases are created in order to verify all of the interfaces between internal modules. [Section 3 and 4 of this document]
- Test cases are created in order to verify all of the interfaces to database and external software application (Google Maps). [Section 3 and 4 of this document]

2.2 Elements to be Integrated

The integration tests described in this document are at the component level.

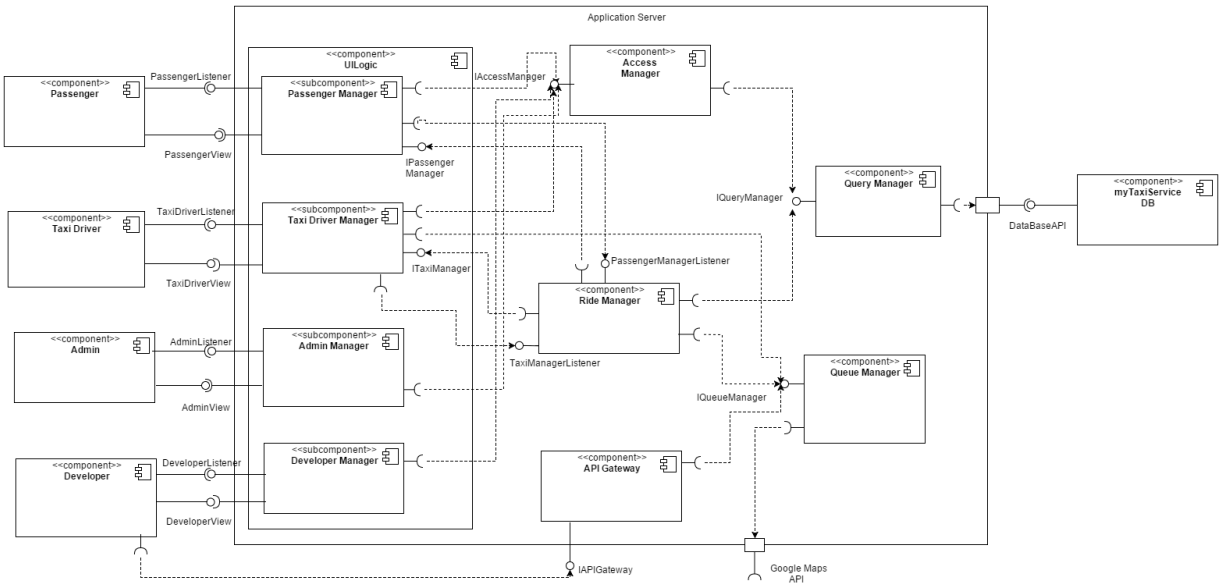

Figure 1: Component View from Design Document

2.3 Integration Testing Strategy

For testing we chose the top-down approach. That means that we will only test in isolation the module(s) that is(are) higher in the hierarchy, this module is then merged with those modules directly called by the former and the combination is tested. We go on like that until the whole system has been integrated and tested. We choose this approach because in this way critical upper level modules can be tested first (like the User Interface). Moreover top-down approach requires stubs to be coded, and stubs are easier to code then drivers (which are required in a bottom-up approach). Another problem with the bottom-up approach is that it's difficult to really know how a component needs to be used by its clients until the clients are implemented. Of course this choice has also some drawback that we will have to face, like the fact that testing upper level modules is harder then testing lower level modules or that top-level modules cannot be really tested perfectly and every time the stubs are replaced with the real modules, the modules which are calling should be properly re-tested again for integrity.

2.4 Sequence of Component / Function Integration

The order of execution of the different integration test cases follows directly from the following table.

ID	Integration Test
I1	Passenger ⟶ PassengerManager
I2	TaxiDriver ⟶ TaxiDriverManager
I3	Administrator ⟶ AdministratorManager
I4	Developer ⟶ DeveloperManager
I5	PassengerManager, TaxiDriverManager, AdministratorManager, DeveloperManager ⟶ AccessManager
I6	PassengerManager,TaxiDriverManager ⟶ RideManager
I7	Developer ⟶ APIGateway
I8	AccessManager, RideManager ⟶ QueryManager
I9	TaxiDriverManager, RideManager, APIGateway ⟶ QueueManager
I10	QueryManager ⟶ Database
I11	QueueManager ⟶ GoogleMapsAPI

3 Individual Steps and Test Description

3.1 Passenger ⟶ PassengerManager

Test ID	I1.1
Test Item	Passenger ⟶ PassengerManager
Test Purpose	This test procedure verifies whether the interface PassengerListener handles correctly: 1. login request 2. logout request 3. registration of new users 4. taxi request 5. taxi reservation 6. accept / decline of ride proposal
Input Specification	1. Credentials of the user (username,password) 2. Parameters required for registration (username,password,email,...) 3. Parameters required for requesting a Taxi (username, pickup location, destination, number of seats) 4. Parameters required for reserving a Taxi (username, pickup location, destination, number of seats, date, time) 5. Parameters required to cancel a reservation (username, reservation ID) 6. Request ID of the ride proposal to accept or decline
Expected Output	1. User's login request is sent 2. User's registration request is sent 3. User's request for a taxi is sent 4. User's reservation is placed 5. User's reservation is cancelled 6. User's response is received
Environmental Needs	Stub for PassengerManager

Test ID	I1.2
Test Item	PassengerManager $\longrightarrow$ Passenger
Test Purpose	This test procedure verifies whether the interface PassengerView handles correctly: 1. the display of all the pages needed by the passenger's GUI 2. messages of taxi proposal 3. error messages
Input Specification	1. Parameters required for each custom screen 2. Parameters required for an incoming taxi proposal (username, pickup location, destination, number of seats, ETA, taxi code) 3. Parameters required for error messages to display (username, error code)
Expected Output	1. User's GUI displays the correct screen 2. User's GUI displays the message containing the taxi proposal (and all its attributes) 3. User's GUI displays the correct error message
Environmental Needs	Passenger component, since this test will be performed after test I1.1

3.2 TaxiDriver ⟶ TaxiDriverManager

Test ID	I2.1
Test Item	TaxiDriver ⟶ TaxiDriverManager
Test Purpose	This test procedure verifies whether the interface TaxiDriverListener handles correctly: 1. login request 2. logout request 3. change status request 4. accept / decline response to taxi request 5. signal of unexpected event 6. GPS coordinates sent by taxi drivers' device every 30 seconds
Input Specification	1. Credentials of the user (username,password) 2. Parameters required for changing the taxi driver status (taxiCode) 3. Request ID of the ride request to accept or decline 4. Parameters required for signaling unexpected events(taxiCode, message) 5. GPS coordinates
Expected Output	1. User's login request is sent 2. User's change status request is sent 3. User's response is received 4. The unexpected event is received 5. GPS coordinates are received and updated every 30 seconds
Environmental Needs	Stub for TaxiDriverManager

Test ID	I2.2
Test Item	TaxiDriverManager ⟶ TaxiDriver
Test Purpose	This test procedure verifies whether the interface TaxiDriverView handles correctly: 1. the display of all the pages needed by the taxi driver's GUI 2. messages of ride request 3. confirm / refuse ride notification 4. error messages
Input Specification	1. Parameters required for each custom screen 2. Parameters required for an incoming ride request (username, pickup location, destination, number of seats) 3. Parameters required for error messages to display (username, error code)
Expected Output	1. User's GUI displays the correct screen 2. User's GUI displays the message containing the ride request (and all its attributes) 3. User's GUI displays the correct error message
Environmental Needs	TaxiDriver component, since this test will be performed after test I2.1

3.3 Admin ⟶ AdminManager

Test ID	I3.1
Test Item	Admin ⟶ AdminManager
Test Purpose	This test procedure verifies whether the interface AdminListener handles correctly: 1. login request 2. logout request 3. registration of new taxi drivers
Input Specification	1. Credentials of the user (username,password) 2. Parameters required for registration (username,password,email,driving license,...)
Expected Output	1. User's login request is sent 2. Taxi Driver's registration is sent
Environmental Needs	Stub for AdminManager

Test ID	I3.2
Test Item	AdminManager ⟶ Admin
Test Purpose	This test procedure verifies whether the interface AdminView handles correctly: 1. the display of all the pages needed by the administrator's GUI 2. error messages
Input Specification	1. Parameters required for each custom screen 2. Parameters required for error messages to display (username, error code)
Expected Output	1. User's GUI displays the correct screen 2. User's GUI displays the correct error message
Environmental Needs	Admin component, since this test will be performed after test I3.1

3.4 Developer ⟶ DeveloperManager

Test ID	I4.1
Test Item	Developer ⟶ DeveloperManager
Test Purpose	This test procedure verifies whether the interface DeveloperListener handles correctly: 1. login request 2. logout request 3. registration of new users 4. update profile request 5. request of API key and API documentation
Input Specification	1. Credentials of the user (username, password) 2. Parameters required for registration (username, password, email,...) 3. Parameters required for profile modification (username, password, email,...) 4. Parameters required for requesting an API key (username)
Expected Output	1. User's login request is sent 2. User's registration request is sent 3. User's update profile request is sent 4. User's request of API key is received
Environmental Needs	Stub for DeveloperManager

Test ID	I4.2
Test Item	DeveloperManager $\longrightarrow$ Developer
Test Purpose	This test procedure verifies whether the interface DeveloperView handles correctly: 1. the display of all the pages needed by the developer's GUI 2. the display of the requested API key or documentation 3. error messages
Input Specification	1. Parameters required for each custom screen 2. API key 3. API documentation 4. Parameters required for error messages to display (username, error code)
Expected Output	1. User's GUI displays the correct screen 2. User's GUI displays the requested API key 3. User's GUI displays correctly the API documentation 4. User's GUI displays the correct error message
Environmental Needs	Developer component, since this test will be performed after test I4.1

3.5 PassengerManager, TaxiDriverManager, AdministratorManager, DeveloperManager ⟶ AccessManager

Test ID	I5
Test Item	PassengerManager, TaxiDriverManager, AdministratorManager, DeveloperManager ⟶ AccessManager
Test Purpose	This test procedure verifies whether the interface IAccessManager handles correctly: 1. login request 2. registration of all type of user 3. update profile request 4. API keys assignment The results of the queries above should be returned properly to the caller.
Input Specification	1. Credentials of the user (username,password) 2. Parameters required for registration (username,password,email,...) 3. Parameters for performing a profile update (username,password, fields to modify) 4. Parameters to assign the APIkey (username)
Expected Output	1. A confirmation or an error code is sent to the caller depending on the parameters submitted before 2. A confirmation or an error code is sent to the caller depending on the parameters submitted before 3. A confirmation or an error code is sent to the caller depending on the parameters submitted before 4. The value of the API key or an error code is sent to the caller depending on the parameters submitted before.
Environmental Needs	Stub for the AccessManager

3.6 PassengerManager, TaxiDriverManager ⟶ RideManager

Test ID	I6.1
Test Item	PassengerManager ⟶ RideManager
Test Purpose	This test procedure verifies whether the interface PassengerManagerListener can: 1. create a new Request / Reservation item 2. delete a Reservation item 3. handle accept or decline messages of taxi proposal
Input Specification	1. Parameters required for creating a taxi request (username, pickup location, destination, number of seats) 2. Parameters required for creating a taxi reservation (username, pickup location, destination, number of seats, date, time) 3. Parameters required to cancel a reservation (username, reservation ID) 4. Request ID of the ride proposal to accept or decline
Expected Output	1. The taxi request is saved 2. The reservation is saved 3. The requested reservation is cancelled 4. Passenger's response is received
Environmental Needs	Stub for the RideManager

Test ID	I6.2
Test Item	TaxiDriverManager ⟶ RideManager
Test Purpose	This test procedure verifies whether the interface TaxiManagerListener can: 1. handle accept or decline messages of ride requests from taxi drivers 2. receive signal of unexpected event
Input Specification	1. Request ID of the ride request to accept or decline 2. Parameters required for signaling unexpected events(taxiCode, message)
Expected Output	1. Taxi driver's response is received 2. The unexpected event is received
Environmental Needs	Stub for the RideManager

Test ID	I6.3
Test Item	RideManager ⟶ PassengerManager
Test Purpose	This test procedure verifies whether the interface IPassengerManager can: 1. send ride proposals to the requesting passenger 2. send error messages to the correct passenger
Input Specification	1. Parameters required for sending a taxi proposal (username, ETA, taxiCode) 2. Parameters required for sending error messages (username, error code)
Expected Output	1. The taxi proposal is received and sent to the correct passenger client 2. The error message is received and sent to the correct passenger client
Environmental Needs	PassengerManager already tested

Test ID	I6.4
Test Item	RideManager ⟶ TaxiDriverManager
Test Purpose	This test procedure verifies whether the interface ITaxiManager can: 1. send ride requests to the correct taxi driver 2. send a confirm / cancel message of a ride request to the correct taxi driver
Input Specification	1. Parameters required for sending a ride request (taxiCode, requestID, pickupLocation, destination) 2. Request ID of the ride to confirm or cancel
Expected Output	1. The ride request is received and sent to the correct taxi driver client 2. The confirm / cancel message is received and sent to the correct taxi driver client
Environmental Needs	TaxiDriverManager already tested

3.7 Developer ⟶ APIGateway

Test ID	I7
Test Item	Developer ⟶ APIGateway
Test Purpose	This test procedure verifies whether the Developer can communicate with the APIGateway of the Application Server, in particular it must be able to: 1. receive HTTP request from clients 2. check the validity of APIkey 3. forward the requests to the QueryManager 4. return responses containing the requested resources and status codes indicating success or failure
Input Specification	1. HTTP requests (GET, POST, PUT, DELETE, and HEAD) with the requested resource 2. API key of the developer
Expected Output	1. status code indicating success or failure 2. HTTP responses with the requested resource
Environmental Needs	Stub for the QueueManager in order to answer to the API requests

3.8 AccessManager, RideManager → QueryManager

Test ID	I8.1
Test Item	AccessManager→ QueryManager
Test Purpose	This test procedure verifies whether the interface IQueryManager handles correctly: 1. the search of a user given certain parameters 2. the add of a new Passenger,TaxiDriver,Administrator,Developer 3. the update of a profile for a given user
Input Specification	1. Parameters required for researching a user by firstname or lastanme or e-mail or username. 2. Parameters required for adding a new instance of Passenger (or TaxiDriver or Administrator or Developer). This parameters are the one inserted by the user during the registration process 3. Parameters required for updating a profile (username, password and all the fields that need to be modified).
Expected Output	1. A list of user matching the research parameters is returned to the Access Manager 2. New user is added to the DB. The Access Manager is aware of the result of the operation (if some fails has occurred it has been notified). 3. The requested fields are modified in the DB. The Access Manager is aware of the result of the operation (if some fails has occurred it has been notified).
Environmental Needs	Stub for the QueryManager

Test ID	I8.2
Test Item	RideManager→ QueryManager
Test Purpose	This test procedure verifies whether the interface IQueryManager handles correctly: 1. the add of a new Ride(or Reservation) for a given Passenger 2. the delete of a Reservation for a given Passenger 3. the research of all the Reservations available for the current day
Input Specification	1. Parameters required for adding a new instance of a Ride (or Reservation). 2. Parameters required for deleting a Reservation. 3. The current day so that the research can be executed
Expected Output	1. New ride is added to the DB. The Ride Manager is aware of the result of the operation (if some fails has occurred it has been notified). 2. The tuple corresponding to the reservation are deleted from the DB. The Ride Manager is aware of the result of the operation (if some fails has occurred it has been notified). 3. A list of the reservations placed for the current day is returned to the Ride Manager
Environmental Needs	Stub for the QueryManager

3.9 TaxiDriverManager, RideManager, APIGateway ⟶ QueueManager

Test ID	I9
Test Item	TaxiDriverManager, RideManager, APIGateway ⟶ QueueManager
Test Purpose	This test procedure verifies whether the interface IQueueManager handles correctly: 1. the search of a Taxi that can take care of a given request 2. the change of the status of a Taxi driver 3. the information about the position of each Taxi Driver 4. the search of all Taxi Driver available in a certain zone 5. The calculus of the ETA given a certain location
Input Specification	1. Parameters for researching a Taxi that can take care of a given request (pickup loction, number of seats) 2. Parameters for changing the status of a Taxi driver (Taxi code) 3. GPS coordinates of taxi drivers 4. Parameters for researching all taxi driver available in a certain zone (pickup location) 5. Parameters for calculating the best ETA given a certain location (pickup location)
Expected Output	1. The information about best taxi available is returned to the caller. 2. The status of the taxi driver is changed correctly. 3. GPS coordinates are saved and updated every 30 seconds 4. A list of taxi drivers available in the required zone is returned to the caller. 5. The best ETA is returned to the caller
Environmental Needs	Stub for QueueManager

3.10 QueryManager ⟶ Database

Test ID	I10
Test Item	QueryManager ⟶ Database
Test Purpose	This test procedure verifies whether the link between the QueryMnager and the database works properly. Particularly it should: 1. handle correctly insert, delete, updates operation performed on the DB 2. produce the correct answer to the query submitted by the QueryManager
Input Specification	1. SQL queries
Expected Output	1. The DB is correctly update and a confirmation is sent back to the caller (In the case of INSERT, UPDATE, DELETE, CREATE,...) 2. The result is returned to the caller (In the case of SELECT queries)
Environmental Needs	No stub is required

3.11 QueueManager ⟶ GoogleMapsAPI

Test ID	I11
Test Item	QueueManager ⟶ GoogleMapsAPI
Test Purpose	This test procedure verifies whether the link between the the Queuemanager and GoogleMaps works properly. Particularly it should: 1. Supply the best path available to bring the taxi driver from his current position to the pickup location where the passenger is waiting. It should also return the ETA.
Input Specification	1. Pickup location and destination
Expected Output	1. Optimal route and the ETA are returned to the caller
Environmental Needs	No stub is required

4 Tools and Test Equipment Required

For the execution of the various test cases identified in previous sections we will use a mix of manual testing and automatic tools. Manual testing is performed directly by testers using the program as an end user would, and then determining whether or not the program acts appropriately. This is particularly useful for some aspects of the application such as the graphical user interfaces, which are very difficult to test automatically since their layout changes frequently and only real users can catch visual considerations like the position and size of images or the font size. Moreover, any bugs that may pop up while a user navigates through the different screens of the application in a certain way are more likely to be caught with manual testing. So for the integration between the different clients and the UI logic subcomponents at least some part of the actual tests will be performed manually. On the other side, with automated testing we can run tests that repeat predefined actions, comparing a developing program's expected and actual outcomes. In this way, once the test cases are set up, they can be run quickly and effectively every time they are needed, without having to continuously repeat manually the same operations. The tools we will use are in particular:

Mockito : which is a tool used to mock up the dependencies of a certain component in order to test individually its particular funtionalities. This is a software mainly addressed to unit testing, but it can be useful also for the integration part, since we can use some of the stubs already developed for the unit testing of each component also as stubs of the called components during the integration testing. In this way, we can add predefined responses to the called inteface methods, even if the invoked component has not been developed.

Arquillian : which provides a simple test harness to produce a broad range of integration tests for JavaEE applications. In particular, Arquillian builds a container to manage the different components and to test the interaction between them, while handling all aspects of test execution, including:

- managing the lifecycle of the container (start/stop),
- resolving injection and resource dependencies of each component,
- deploying the archive to test (deploy/undeploy) and
- capturing results and failures.

Therefore, we can use Arquillian to develop integration tests for the server beans just like regular unit test. Rather than instantiating component classes using Java's new operator, which is customary in a unit test, the correct instance of the component is injected directly into the test class, so that we are testing the actual component, just as it runs inside the application. Another important scenario in integration testing is performing data access, in order to verify the interaction with the database. For this purpose, we can create Arquillian tests to ensure that

the QueryManager, which interacts with the database server, is able to persist and subsequently retrieve all the entities saved in the database.

4.1 Test environment

The environment on which the application will be tested is typically different from the runtime environment, but it should be as close as possible to the real one, in order to uncover any environment/configuration related issues. We plan to use:

- at least two Android smartphones, one with the Passenger app and one with the Taxi Driver app. Each app should be tested on different Android versions for avoiding compatibility issues, for example once with the latest version(Android 6.0) and once with an older version (e.g. Android 2.3),
- a computer with Firefox or Google Chrome to test the browser interface of the system;
- a computer with JavaRE version 7 and Eclipse Java EE IDE running GlassFish Application Server;
- another computer running MySQL Server

5 Program Stubs and Test Data Required

It is common in unit and integration tests to stub collaborators of the class under test so that the test is independent from the implementation of the collaborators. The basic technique is to implement the collaborators as concrete classes which only exhibit the small part of the overall behaviour of the collaborator which is needed by the class under test. For the interaction between each client and the respective UI logic component, we can simply create stub classes that print on the output a confirmation of the received request, since the UI logic components don't actually perform the requested operations, but have to forward them to the correct business component. The stub coded for the AccessManager would return to the caller sample values for APIkeys,and confirmation code as default to answer to login/update and register request. The other stubs would perform similarly to the stubs described before, each of them will return to the caller sample values coherently to what is written in the Expected Output (of the table in which the stubs is involved) in section 4. As for the test data, the testing activity will need that the Database (or a stub for it in the case it is not already integrated) already contains some stored data, in order to test specific interactions with it, such as retrieving certain attributes of a registered user and searching for all the reservations available for the current day. In particular, we need:

- registered user profiles (of all types) to test the login / logout / view profile operations
- some rides associated to a specific passenger to test the display of his/her rides history
- some reservations to test the managing of reservations of a specific day

6 Appendix

6.1 Software Tool used

- TexMaker (http://www.xm1math.net/texmaker/): LaTeXeditor, used to redact this document.

6.2 Hours of work

- Alberto Gasparin $\sim$ 6 h
- Vito Matarazzo $\sim$ 6 h