Integration Test Plan Document

M. Albanese, M. Bianchi, A. Carlucci

January 21, 2016

Contents

1 Introduction

1.1 Purpose

The purpose of this document, the **Integration Test Plan Document** (ITPD), is to describe how integration tests are to be performed.

The tests that will be described here will mainly focus on the flow of information between different modules rather than on the modules themselves. In particular, it describes the adopted methodologies, the sets of all tests to be performed, the tools that will be used during the whole process.

1.2 Scope

The system will be an optimization of a pre-existing, non-software solution for renting taxis already in use in the city. The new system will let users to rent or reserve a taxi through a mobile or a web application and will also let taxi drivers to take care of the users' requests in a more simple and effective way. In addition to a better user interface, the new system will focus on a smarter organization of the vehicles deployed in each city zone, resulting in a more efficient service for the citizens.

1.3 List of definitions and abbreviations

JUnit The tool used for unit testing. See Section 4 for more information.

Mockito A mocking framework used in conjuction with JUnit.

Arquillian The tool used for the actual integration testing. See Section 4 for more information.

RASD Requirements & Analysis Specification Document

DD Design Document

ITPD Integration Test Plan Document

DBMS Database Management System

GPS Global Positioning System

API Application Programming Interface

UI User Interface

2 Integration Strategy

2.1 Entry Criteria

Several entry criteria must be met before Integration Testing phase.

The whole architecture must be designed according to the contents of the **Design Document**, to be written along with this one. Following this phase, each component that will be integrated must be - of course - coded.

After development phase, each module must successfully pass a thorough **unit testing**, which guarantees the absence of critical bugs in the codebase. As stated in Section 4, `JUnit` will be the privileged tool for this phase.

Along with unit testing, **code inspection** is done, via both manual and automated work; all these devices will lead to well-formed and robust modules, ready to be composed together.

In addition to this, the following must have been produced:

- A thorough Javadoc documentation covering public methods
- The Requirements Analysis and Specification Document (RASD) [3]
- The Design Document (DD) [4]
- Code Inspection Document (CID) [5]

2.2 Elements to be integrated

The list of all elements to be integrated is reported in Table 1. As you may see, this closely parallels what was written in [4]. For more information on the order of the integration tests, see Section 2.4.1; for more information about the tests themselves, see Section 3.

2.3 Integration Testing Strategy

The strategy that will be adopted is the so-called **bottom-up** approach.

An incremental approach is fundamental, in order to prevent all shortcomings that are typical to *Big Bang approach* (you have to wait until all modules are complete, localizing the faulty components can be difficult, some interfaces could be missed easily during testing...).

We decided to choose bottom-up over top-down because there are many components at the lower levels (see Figure 2.4.2); this means that only few drivers, if any, are needed.

ID	Component	Subsystem
1	GuestManager	Client
2	Customer	Client
3	CustomerManager	Client
4	Address	TaxiRes
5	Call	TaxiRes
6	Reservation	TaxiRes
7	Request	TaxiRes
8	Ride	TaxiRes
9	TaxiResManager	TaxiRes
10	RideManager	TaxiRes
11	TaxiAllocationDaemon	TaxiAllocation
12	Zone	TaxiAllocation
13	TaxiHandler	TaxiAllocation
14	Taxi	TaxiDriver
15	TaxiDriverManager	TaxiDriver

Table 1: List of all components to be integrated

2.4 Sequence of Component/Function Integration

As the system consists of several different parts interrelated one to the other, we decided to plan integration testing under two different points of view, in accordance to what was written in [4, p. 4].

In particular, Section 2.4.1 will explain how the main software components will be integrated, as they were defined in [4] in the *High Level Components* section, whilst the following section will explain how integration will take part on a upper level by considering subsystems only.

2.4.1 Software Integration Sequence

First of all, all transactions with the DB must be operative in order to proceed with all other integrations: for this reason, all modules interacting with the DBMS are tested first.

After that, `Client` and `TaxiReservation` are tested, so that `Client` is completely functional.[1]

Since the allocation depends on at least a call, the integration between `TaxiAllocation` and `TaxiReservation` is done next, followed by the

[1] As written in [4], `Client` contains the data and the logic for managing registered users and guests. `TaxiReservation` is needed for the retrieval of all calls done by a specified user

Test ID	Component 1	Subsystem 1	Component 2	Subsystem 2
1	Taxi	TaxiDriver	DBMS	DBMS
2	Zone	TaxiAllocation	DBMS	DBMS
3	Ride	TaxiReservation	DBMS	DBMS
4	Customer	Client	DBMS	DBMS
5	Address	TaxiReservation	DBMS	DBMS
6	Call	TaxiReservation	DBMS	DBMS
7	Request	TaxiReservation	DBMS	DBMS
8	Reservation	TaxiReservation	DBMS	DBMS
9	TaxiDriverMgr	TaxiDriver	Taxi	TaxiDriver
10	TaxiHandler	TaxiAllocation	Taxi	TaxiDriver
11	TaxiHandler	TaxiAllocation	Ride	TaxiReservation
12	RideManager	TaxiReservation	Ride	TaxiReservation
13	CustomerManager	Client	Customer	Client
14	GuestManager	Client	Customer	Client
15	TaxiResManager	TaxiReservation	RideManager	TaxiReservation
16	TaxiAllocationDaemon	TaxiAllocation	TaxiHandler	TaxiAllocation
17	TaxiAllocationDaemon	TaxiAllocation	Zone	TaxiAllocation
18	TaxiAllocationDaemon	TaxiAllocation	Ride	TaxiReservation
19	TaxiResManager	TaxiReservation	Call	TaxiReservation

Table 2: List of all components to be integrated

integration between `TaxiDriver` (which contains all logic for updating a specified taxi) and `TaxiAllocation` (which contains the code for explicitly reserving a taxi).

Eventually the core is complete; the last integration test is done on the mobile app of each taxi driver and the `TaxiDriver` component (for example, to check if the `updatePosition()` method is working fine).

N.	Subsystem	Integrates with
1	Client	DBMS
2	TaxiRes	DBMS
3	TaxiAllocation	DBMS
4	Client	TaxiReservation
5	TaxiAllocation	TaxiReservation
6	TaxiDriver	TaxiAllocation
7	Mobile App	TaxiDriver

Table 3: Integration order of the components

N.	Subsystem	Integrates with
1	Core (Back-end)	DBMS
2	Taxi Driver mobile application (logic and UI)	Core
3	Mobile UI	Core
4	Web UI	Core

Table 4: Integration order of the subsystems

2.4.2 Subsystem Integration Sequence

Four main subsystems make up the whole architecture, listed in Table 4. All components described in [4] make up the Core subsystem, which contains the main logic of the service; as stated in the previous section, the first important integration to be done is the one with the `DBMS` . When the core system is ready, the mobile application for a taxi driver must be tested in conjunction with the main service. Finally, the UI must be integrated with the business tier.

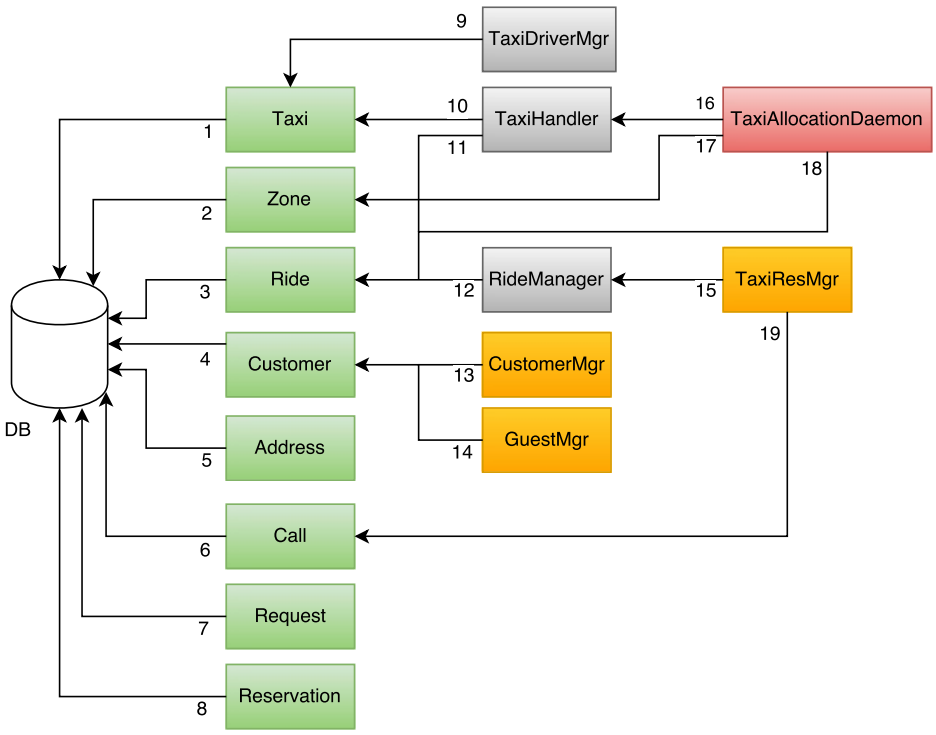

Figure 1: The elements to be integrated; the number refer to the tests described in Section 3

3 Individual Steps and Test Description

3.1 Component Tests

3.1.1 Test 1: Taxi - Access to DB

Test Case Identifier	1
Test Item(s)	`Taxi` → `DBMS`
Input Specification	Frequent queries on table Taxi.
Output Specification	The corresponding tuples to the query.
Environmental Needs	Working Glassfish Server, Working DB Server.
Test Description	Verify that all entities in the DB are correctly mapped through JavaBeans.

3.1.2 Test 2: Zone - Access to DB

Test Case Identifier	2
Test Item(s)	`Zone` → `DBMS`
Input Specification	Frequent queries on table Zone.
Output Specification	The corresponding tuples to the query.
Environmental Needs	Working Glassfish Server, Working DB Server.
Test Description	Verify that all entities in the DB are correctly mapped through JavaBeans.

3.1.3 Test 3: Ride - Access to DB

Test Case Identifier	3
Test Item(s)	`Ride` → `DBMS`
Input Specification	Frequent queries on table Ride.
Output Specification	The corresponding tuples to the query.
Environmental Needs	Working Glassfish Server, Working DB Server.
Test Description	Verify that all entities in the DB are correctly mapped through JavaBeans.

3.1.4 Test 4: Customer - Access to DB

Test Case Identifier	4
Test Item(s)	`Customer` → `DBMS`
Input Specification	Frequent queries on table Customer.
Output Specification	The corresponding tuples to the query.
Environmental Needs	Working Glassfish Server, Working DB Server.
Test Description	Verify that all entities in the DB are correctly mapped through JavaBeans.

3.1.5 Test 5: Address - Access to DB

Test Case Identifier	5
Test Item(s)	`Address` → `DBMS`
Input Specification	Frequent queries on table Address.
Output Specification	The corresponding tuples to the query.
Environmental Needs	Working Glassfish Server, Working DB Server.
Test Description	Verify that all entities in the DB are correctly mapped through JavaBeans.

3.1.6 Test 6: Call - Access to DB

Test Case Identifier	6
Test Item(s)	`Call` → `DBMS`
Input Specification	Frequent queries on table Call.
Output Specification	The corresponding tuples to the query.
Environmental Needs	Working Glassfish Server, Working DB Server.
Test Description	Verify that all entities in the DB are correctly mapped through JavaBeans.

3.1.7 Test 7: Request - Access to DB

Test Case Identifier	7
Test Item(s)	`Request` → `DBMS`
Input Specification	Frequent queries on table Request.
Output Specification	The corresponding tuples to the query.
Environmental Needs	Working Glassfish Server, Working DB Server.
Test Description	Verify that all entities in the DB are correctly mapped through JavaBeans.

3.1.8 Test 8: Reservation - Access to DB

Test Case Identifier	8
Test Item(s)	`Reservation` → `DBMS`
Input Specification	Frequent queries on table Reservation.
Output Specification	The corresponding tuples to the query.
Environmental Needs	Working Glassfish Server, Working DB Server.
Test Description	Verify that all entities in the DB are correctly mapped through JavaBeans.

3.1.9 Test 9: TaxiDriverManager - Update Taxi entity

Test Case Identifier	9
Test Item(s)	`TaxiDriverManager` → `Taxi`
Input Specification	Update information on position and status of the taxi.
Output Specification	Taxi has modified data.
Environmental Needs	Working Glassfish Server, Working Database Server.
Test Description	The purpose of this test is to verify that all the information are correctly updated on the server.

3.1.10 Test 10: TaxiHandler - Taxi assignment to a ride

Test Case Identifier	10
Test Item(s)	`TaxiHandler` → `Taxi`
Input Specification	There is a new ride available.
Output Specification	This ride is assigned to taxi and the corresponding taxist is notified.
Environmental Needs	Working Glassfish Server, Working Database Server.
Test Description	The aim of this test is to verify that a new ride is properly assigned to a taxi in the correct zone and to ensure that the correct taxi driver is notified. This means that the taxi must be temporarily dequeued and set to BUSY status.

3.1.11 Test 11: TaxiHandler - Ride completion with a taxi

Test Case Identifier	11
Test Item(s)	`TaxiHandler` → `Ride`
Input Specification	A new ride is on schedule.
Output Specification	A Taxi is assigned to the specified ride.
Environmental Needs	Working Glassfish Server, Working Database Server.
Test Description	This test aims to verify that a `Taxi` is correctly assigned to the ride when a request needs to be handled.

3.1.12 Test 12: RideManager - Operations on rides

Test Case Identifier	12
Test Item(s)	`RideManager` → `Ride`
Input Specification	Some `Ride` needs to be changed, according to user or system decision.
Output Specification	The corresponding `Ride` is correctly updated.
Environmental Needs	Working Glassfish server, Working Database server.
Test Description	The purpose of the test is to verify that RideManager updates the correct information about a `Ride` properly. This means that all other rides must be unaffected by the update.

3.1.13 Test 13: CustomerManager - Customer Services

Test Case Identifier	13
Test Item(s)	`CustomerManager` → `Customer`
Input Specification	There are some pending operations initiated by users, which aim is to change `User` entities. In particular, the following operartions must be tested: editing a profile, deleting one or logging out.
Output Specification	For each operation, every involved `User` entity must be updated correctly, according to the specified operation.
Environmental Needs	Working Glassfish Server, Working Database server.
Test Description	With this test, we can ensure that CustomerManager is working properly.

3.1.14 Test 14: GuestManager - Guest Services

Test Case Identifier	14
Test Item(s)	`GuestManager` → `Customer`
Input Specification	There are some pending operations initiated by unauthenticated users.
Output Specification	Every operation must be executed correctly, *according to the user level* (unregistered!). This includes logging in, signing up and confirm his own mail (by using a mock mail sender - see Section 4)
Environmental Needs	Working Glassfish Server, Working Database server.
Test Description	This test will check that every `Guest` operation is working properly and that guests could do actions *if and only if* those actions are allowed to be done by guests.

3.1.15 Test 15: TaxiResManager - Check existing rides

Test Case Identifier	15
Test Item(s)	`TaxiResManager` → `RideManager`
Input Specification	Call to existRide() method.
Output Specification	Return value must be chosen according to the existance of this `Ride` .
Environmental Needs	Working Glassfish Server, Working Database server.
Test Description	Check that `RideManager` gives back valid results. For example, if the `Ride` exists, the output should be `True` . Otherwise, `False` will be the expected result.

3.1.16 Test 16: TaxiAllocationDaemon - Creation of a handler for an incoming ride

Test Case Identifier	16
Test Item(s)	`TaxiAllocationDaemon` → `TaxiHandler`
Input Specification	Ride to be handled.
Output Specification	Association between a ride and a taxi.
Environmental Needs	Working Glassfish Server, Working Database server.
Test Description	The aim of the test is to check that an handler is created for every incoming ride. This should test that TaxiAllocationDaemon allocates only one ride to a single TaxiHandler.

3.1.17 Test 17: TaxiAllocationDaemon - Available taxis in zones

Test Case Identifier	17
Test Item(s)	`TaxiAllocationDaemon` → `Zone`
Input Specification	Queue related to a zone.
Output Specification	Number of available taxis for a zone.
Environmental Needs	Working Glassfish Server, Working Database server.
Test Description	The purpose of the test is to check that the number of available taxis in every zone is equal to each corresponding zone queue. This is done to guarantee consistency on results between different components.

3.1.18 Test 18: TaxiAllocationDaemon - Retrieve ride information

Test Case Identifier	18
Test Item(s)	`TaxiAllocationDaemon` → `Ride`
Input Specification	Ride on schedule.
Output Specification	Ride used in the creation of new TaxiHandler.
Environmental Needs	Working Glassfish Server, Working Database server.
Test Description	The purpose is to check that the proper information are correctly retrived and used for the creation of a new TaxiHandler. The rides inside TaxiAllocationDaemon are the ones got by a periodic refresh on the DB (by using `refreshRideQueue()`).

3.1.19 Test 19: TaxiResManager - Creation of a call

Test Case Identifier	19
Test Item(s)	`TaxiResManager` → `Call`
Input Specification	Information provided by the user.
Output Specification	New Call entity.
Environmental Needs	Working Glassfish Server, Working Database server.
Test Description	Verify that a new call object is correctly created, and then added to the database through Java Entity Beans with the correct information.

3.2 Subsystem Tests

3.2.1 Test 20: Application Server - Main functionalities with DB

Test Case Identifier	20
Test Item(s)	`Core (back-end)` → `DBMS`
Input Specification	Standard query by the application server on the DB
Output Specification	Correct tuples, based on the actual component that made the request
Environmental Needs	Working Glassfish Server, Working Database server.
Test Description	Several components in the application server use DB for their functionalities, by either inserting, updating or deleting information. For more information, see the first tests in section Section 3.1

3.2.2 Test 21: Taxi Driver Mobile Application - Integration with the application server

Test Case Identifier	21
Test Item(s)	`Taxi Driver Mobile Application` → `Core (back-end)`
Input Specification	A request for a ride by the system
Output Specification	A notification by the driver that he accepted / refused
Environmental Needs	Working Glassfish Server, Working Database server, mobile phone with GPS enabled
Test Description	When a new Ride has to be assigned, TaxiHandler starts to send push notifications to taxi drivers in its queue, as specified in [4, p. 22]. The driver must take a decision, which is sent to back to the server

3.2.3 Test 22: UI - Webapp

Test Case Identifier	22
Test Item(s)	`UI` → `Core (back-end)`
Input Specification	UI for the web app
Output Specification	Results on the application server
Environmental Needs	Working Glassfish Server, Working Database server, PC
Test Description	All principal functionalities must be tested through the web application. It must be guaranteed that the UI is easily displayed on all common browsers on different resolutions, respecting the laws about accessibility and privacy.

3.2.4 Test 23: UI - Mobile app

Test Case Identifier	23
Test Item(s)	`UI` → `Core (back-end)`
Input Specification	UI for the mobile app
Output Specification	Results on the application server
Environmental Needs	Working Glassfish Server, Working Database server, Mobile Phone
Test Description	All principal functionalities must be tested through the mobile application. It must be guaranteed that the UI is easily displayed on all supported operating systems and being responsive.

4 Tools and Test Equipment Required

JUnit JUnit is a simple framework to write repeatable tests. Each test is done on a single unit, usually composed of one public class. Tests can also be grouped in Suites for multiple instances at once.

Since complex environments must be described within this project (for example, the interaction between `TaxiHandler` and `Taxi` or all interactions between a module and the DBMS), a mock framework is necessary. Among several and similar products(JMock, EasyMock, Powermock...), we decided to use **Mockito** for its simplicity and clearness.

These tools are used *before* Integration Testing happens (described throughout this document).

Arquillian Arquillian is an integration testing framework for business objects that are executed inside a container or that interact with the container as a client.

It combines a unit testing framework (JUnit), and one or more supported target containers (Java EE container, servlet container, etc) to provide a simple, flexible and pluggable integration testing environment.

Arquillian strives to make integration testing no more complicated than basic unit testing, so we decided to use Arquillian as our integration testing framework for its semplicity.

Cellular Phones As described in [3], the mobile application will be available for customers using Android and iOS. This implies that at least two cellular phones, one for each OS, have to be used for testing. In particular:

- Android phones must be updated to at least Android JellyBean
- iOS phones must be updated to v.9
- Each phone must have a working Internet connection and a GPS system enabled

PCs At least a personal computer must be used to ensure accessibility through the web app; in order to accomplish this, several browsers are used (mainly Google Chrome, Mozilla Firefox, Internet Explorer and Edge)

5 Program Stubs and Test Data Required

In an undeveloped environment, following the criteria of **bottom-up approach**, we need to define and use *drivers* in order to have a complete environment in which we can test the developed parts.

Test Server: A working Glassfish test server is needed in order to properly host the Application server.

Test database: The target environment must have a working and configured `DBMS` , in which test data and tables must reflect the entities and the relations described in the ER-diagram showed in the Design Document. This mock DB should contain random valid and invalid data about `Taxi` , `Zone` , `Ride` , `Customer` , `Address` , `Call`

Drivers for JavaEE: Drivers used to test the proper behaviour of the Java Entity Beans while the Application server is not fully implemented. These are placeholders for all Managers in the application (`RideManager` , `CustomerManager` ...) and can be eliminated as the upper-level modules are ready.

API client: It is also necessary to emulate a client application with an API-client which interacts with the server via HTTP requests.

Test e-mail confirmation: An email sender/receiver is needed in order to test and automate the email confirmation process when a user signs up for the service.

Fake GPS data: In order to test position handling for taxis, a set of random GPS data are generated, both inside and outside the city.

References

[1] Prof. Di Nitto - *Assignment 4 - Integration Test Plan*

[2] AA.VV. - *SpinGrid - Integration Test Plan Example*

[3] Albanese Michele, Bianchi Mattia, Carlucci Alain - *myTaxiService: Requirements Analysis and Specification Document*

[4] Albanese Michele, Bianchi Mattia, Carlucci Alain - *myTaxiService: Design Document*

[5] Albanese Michele, Bianchi Mattia, Carlucci Alain - *myTaxiService: Code Inspection Document*

[6] AA.VV. - *Arquillian Reference Guide* (`https://docs.jboss.org/author/display/ARQ/Reference+Guide?_sscc=t`)

[7] AA.VV. - *Mockito Reference Guide* (`http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html`)

Hours spent

Each member has spent 10 hours while writing this document.