Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
M.Sc. in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

myTaxiService

Software Engineering 2 - Project

ITPD

Integration Test Plan Document

version 2.0

22nd February 2016

Authors:
Alberto Maria METELLI Matr. 850141
Riccardo MOLOGNI Matr. 852416

Academic Year 2015–2016

Contents

List of Figures

1 Introduction

1.1 Purpose

The purpose of ITPD (*Integration Test Plan Document*) is to give a detailed description of the plan for the integration test of the *myTaxyService* application. The integration test, as a part of the V&V (Verification and Validation) process, is executed after the unit test of the involved modules, which constitutes its main entry condition, and before the system and acceptance test. It is aimed to exercise the interaction between modules and the corresponding interfaces in order to check the compatibility against functional, performance and reliability requirements. The preferred approach for integration testing is *black-box* since single modules are supposed to be already tested isolated and no simple coverage criteria can be defined. Applications are often composed of many modules and the relations among them can be very intricate therefore an *integration test plan* is a key feature of the project plan and has to be compliant to architecture define in the DD and to the build plan, that describes the sequence in which modules will be compounded. The Integration Test Plan Document is intended to be the main reference for this process and it is mainly addressed to the integration test team.

1.2 Scope

The *myTaxyService* is an application intended to optimize taxi service in a large city, making the access to service simpler for the passengers and ensuring a fair management of the taxi queues.

Passengers will be able to request a taxi either through a web application or a mobile app; of course the "traditional" ways to call for a taxi, like a phone call or stopping the taxi along the road, will be still available and integrated into the system to-be. The software will make the procedure of calling a taxi simpler (by using GPS information passenger doesn't need to know the address if the taxi is needed for the current position) and more usable (passenger will be provided with information about the waiting time). Moreover, by means of the application, the passenger can reserve a taxi for a certain date and time, specifying the origin and the destination of the ride.

Taxi drivers will use a mobile app to inform the system about their availability and to confirm that they are going to take care of a call (or to reject it for any reason). The software will make the taxi management more efficient: the system will be able to identify the position of each taxi by using GPS; the city will be divided in virtual zones and a suitable distribution of the taxi among the zones will automatically be computed.

1.3 Definitions, Acronyms, Abbreviations

In this paragraph all the terms, acronyms and abbreviations used in the following sections are listed.

1.3.1 Definitions

- *Request*: the action performed by the passenger of calling a taxi for the current position.
- *Confirmed request*: a request that has been accepted by a taxi driver.
- *Reservation*: the action performed by the passenger of booking a taxi for a specific address and specific date and time.
- *Waiting time*: an estimation of the time required to taxi driver to get to passenger's position.
- *Taxi code*: a unique alphanumerical identifier of the taxi.

- *Available taxi queues*: data structures used to store the references of the available taxis, also used to select the taxis to which forward a request.
- *Automatic geolocalization*: a system that provides the geographic coordinates of the user. For this document it can be either a GPS system or browser geolocalization.
- *Passengers' application*: the applications used by passengers to access to TS system. For this document it can be either PMA or PWA.
- *Login credentials*: username and password.
- *Notification*: communication from TS to taxi driver to move to a specific zone.

1.3.2 Acronyms

- TS: myTaxiService.
- PMA: Passenger mobile application.
- PWA: Passenger web application.
- TMA: Taxi driver mobile application.

1.3.3 Abbreviations

- Sn n-th subsystem.
- SnIm m-th integration test of the n-th subsystem.
- SnIm-Tk k-the test case of m-th integration test of the n-th subsystem.
- SIm m-th integration test of the full system.
- SIm-Tk k-the test case of m-th integration test of the n-th subsystem.
- EIm m-th integration test with external subsystems.

1.4 Reference documents

[1] The assignment of *myTaxiService*.

[2] RASD (Requirements Analysis and Specification Document) of the *myTaxiService*.

[3] DD (Design Document) of the *myTaxiService*.

[4] Software Engineering 2 course slides.

[5] Arquillian documentation. `https://docs.jboss.org/arquillian/reference/1.0.0.Alpha1/en-US/html_single/`

1.5 Document Structure

This document is composed of five sections and an appendix.

- The first section, this one, is intended to define the goal of the ITPD, a very high level description of the main functionalists of the *myTaxiService* system and the resources used to draw up this document.
- The second section constitutes the core of the test plan. This section is devoted to the description of the integration test strategy: the preconditions required to start the integration test will be presented, the main rationals behind the chosen strategy will be discussed; the elements to be integrated will be listed with reference to the ones presented in the DD. Eventually the sequence of integration steps will be clearly illustrated distinguishing between the different levels of abstraction adopted to perform the test.
- The third section contains the definition of the test sets and test cases. Each test will be presented with reference to the sequence explained in the second section, together with the hypothesis about the initial state of the system and the expected results.
- The fourth section is dedicated to the test tools. We will suggest a set of commercial tools suitable to perform the integration test with their main characteristics and the reason why they are appropriate in this context.
- The fifth section is devoted to the description of the program stubs, drivers and the specific test data to accomplish the integration test in some particular scenarios such as external system integration.
- The appendix contains a brief description of the tools used to produce this documents and the number of hours each group member has worked towards the fulfillment of this deadline and the revision history.

2 Integration strategy

This section is devoted to the explanation of the main choices related to the integration testing plan mainly concerning the integration testing strategies.

2.1 Entry criteria

An entry criterion for the integration test is a precondition that is supposed to hold when the integration test is initialized, if one of them is not verified it can compromise or make even impossible the entire process. According the the *software lifecycle*, we identified the following as entry criteria.

1. The implementation phase is terminated therefore the project is code-complete.
2. Unit testing or at least sanity checking[1] for every module/component is complete (every test has been run).
3. All High prioritized bugs fixed and closed (every test has been passed).
4. The internal documentation has been updated to reflect the current state of the product.
5. The ITPD is complete and been approved by QA group.
6. The integration teasing environment setup is completed and stable.

2.2 Elements to be integrated

In this subsection we propose the structure of the designed system, starting from what was described in the DD, in order to clarify the steps needed for the integration testing plan. We do not treat here the problem of integration with external system since a part of section 5 is devoted to it, we assume that part of integration has already been performed.

2.2.1 Preliminary considerations

The integration test plan should be driven by the conceptual system decomposition proposed in the DD, so we construct the plan starting from the components represented there. Although each component can be easily mapped into a programmatic class, being a cohesive and coherent group of functionalists, it is very likely that not all classes needed for the implementation of the application appear as component, most of them will be probably auxiliary class; therefore we assume that during the *unit test*, which constitutes an entry condition for the integration test, the integration *within* the component is performed. If this assumption holds we can proceed to the integration test starting from the components as depicted in the DD.

2.2.2 Levels of integration

Considered the distributed nature and the clear modular structure of myTaxiService application a two level approach of integration testing should be suitable. In particular, the integration phase will be realized at:

[1] *Sanity checking,* sometimes called sanity test, consists in checking that a module/compoent does not contain elementary mistakes or impossibilities, or is not based on invalid assumptions. It is typically a more shallow verification approach with respect to unit testing since just evident mistakes can be manifested. To clarify the distinction, checking that the result of a multiplication of negative numbers is positive is typical of a sanity checking while comparing the result against the one provided by an oracole is typical of unit testing.

- *component level*: each component will be integrated and tested against every dependent component in the contest of the subsystem to which it belongs;
- *subsystems level*: once each subsystem is entirely integrated, all of them will be integrated and tested.

This approach is here just mentioned to understand the following representation of the hierarchies[2] of components/subsystems and will be further discussed in the dedicated section 2.3.

2.2.3 Subsystems

According to the DD the following diagram[3] describes the hierarchy of subsystems to be integrated. The decomposition clearly reflects the JEE architecture, for a detailed description refer to [3].

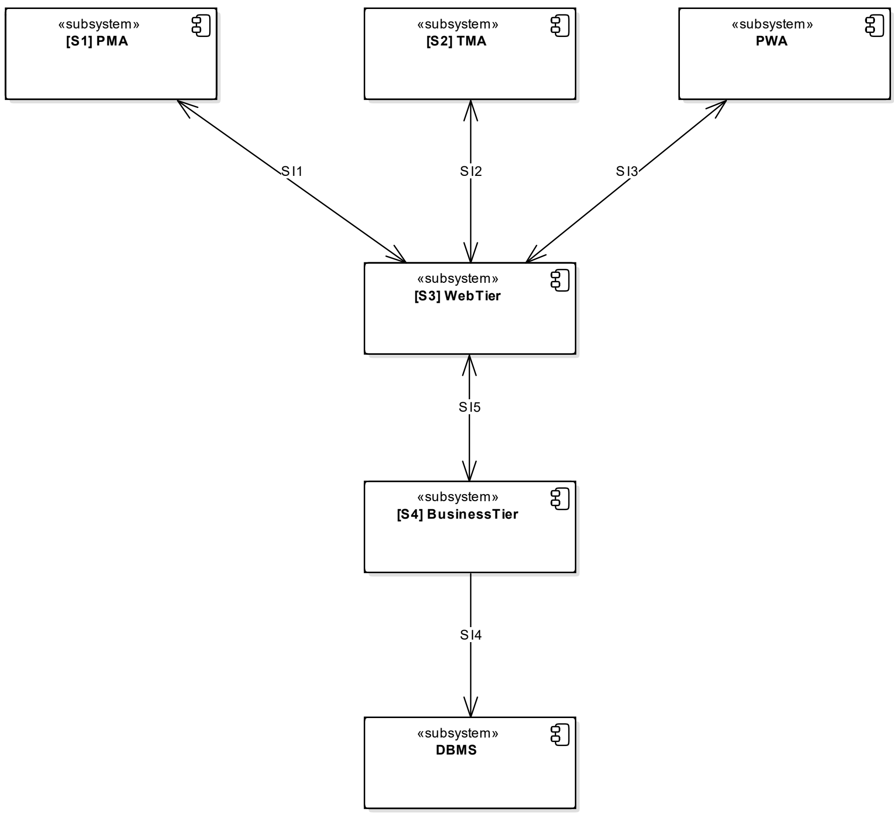

Figure 1: Integration testing plan - subsystem level hierarchy

For each subsystem we present the internal hierarchy of components, since PWA is made of the browser no internal integration is needed, also for the DBMS the internal structure is not known so it is not further specified.

[2]A hierarchy of the dependencies of components is a *DAG* (Directed Acyclic Graph) therefore a bottom-up strategy is actually an plan where integration happens in reverse topological order while in the top-down strategy integration occurs in topological order. If a component A calls a method of another component B (namely A *depends* on B) then B belongs to the layer right below in the hierarchy with respect to A.

[3]The notation used here is a simplified version of UML Component Diagram where just dependencies among modules are represented, while interfaces are not.

2.2.4 PMA

Notice that the hierarchy contains a cycle between PMAUserInterface, PMAController and CCommunicator (so it is not properly speaking a hierarchy) and this is due to the usage of the pattern Observer-Observable where the CCommuncator constitutes the model (actually the link to the remote model)[4]. To break the cycle in order to perform the integration more stubs will be needed.

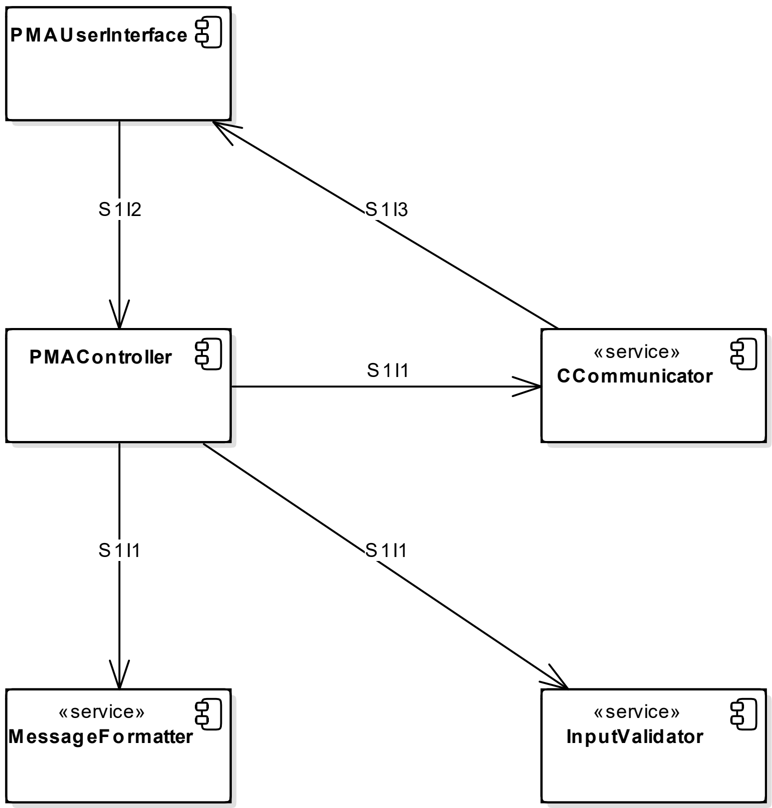

Figure 2: Integration testing plan - PMA

[4]From an implementative point of view that cyclic dependency does not exist since CCommunicator does not accesses directly to PMAUserInterface but just sees the interface Observer.

2.2.5 TMA

Since TMA shares the same structure of PMA the same considerations explained above are valid here.

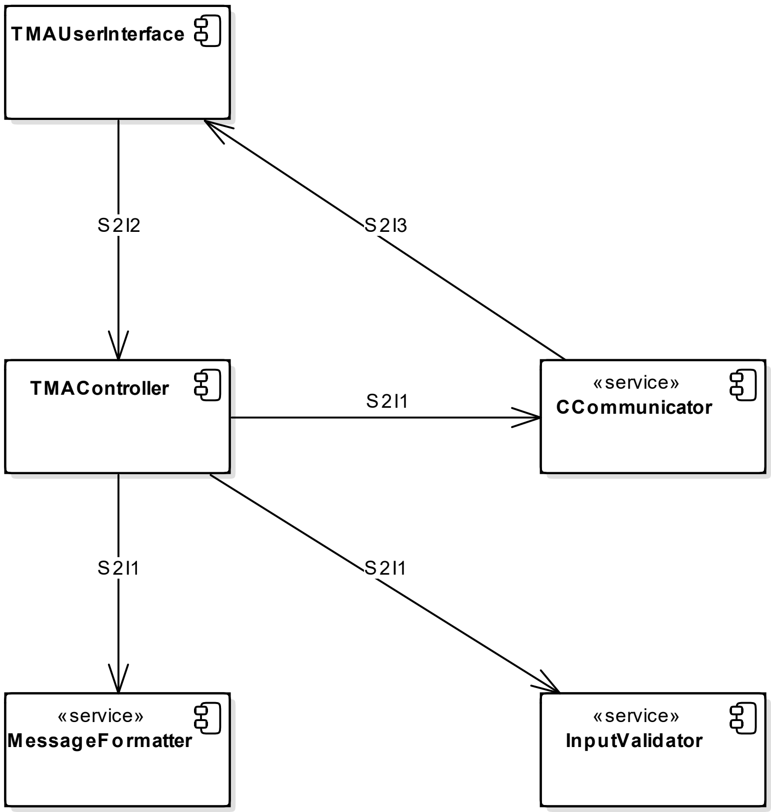

Figure 3: Integration testing plan - TMA

2.2.6 Web subsystem

Notice that the dependency graph is a cycle, this is due to the fact that SCommunicator and CommandEventDispatcher manage the bidirectional flow of messages between clients and Business subsystem. However the interfaces involved in the exchange of messages in the two direction are different (SCommunicator → CommandDispatcher and EventDispatcher → CommunicatorSender).

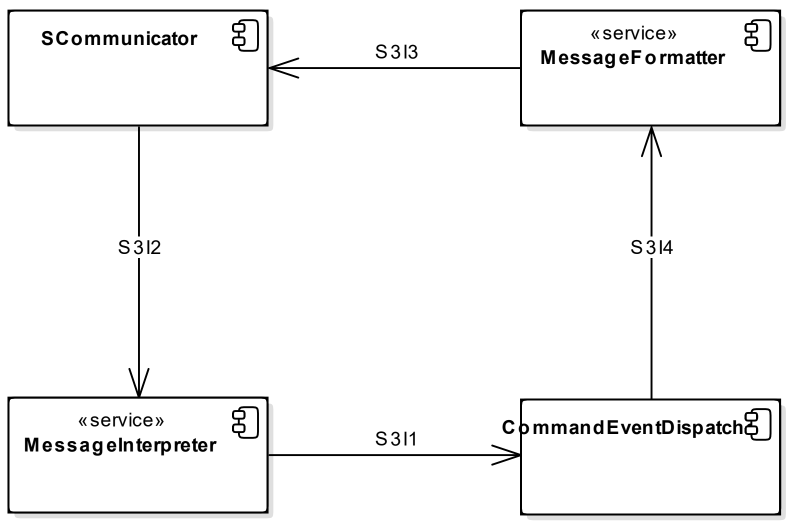

Figure 4: Integration testing plan - TMA

2.2.7 Business subsystem

As it is represented in the DD the business subsystem is in turn composed of an internal macro-component TaxiManager, therefore the process of intergration will be performed integrating TaxiManager before and than completing the integration with the other components. So we present two different hierarchies.

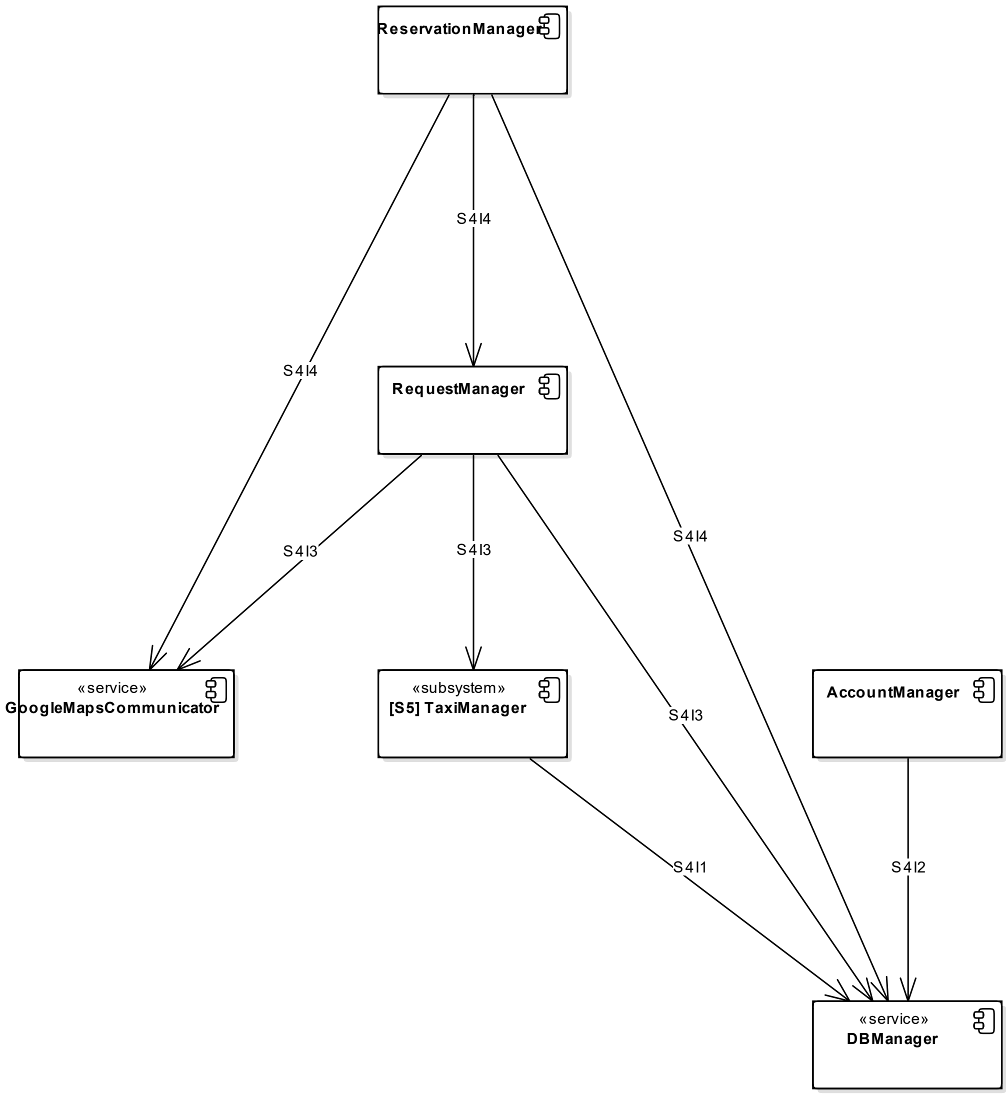

Figure 5: Integration testing plan - Business subsystem

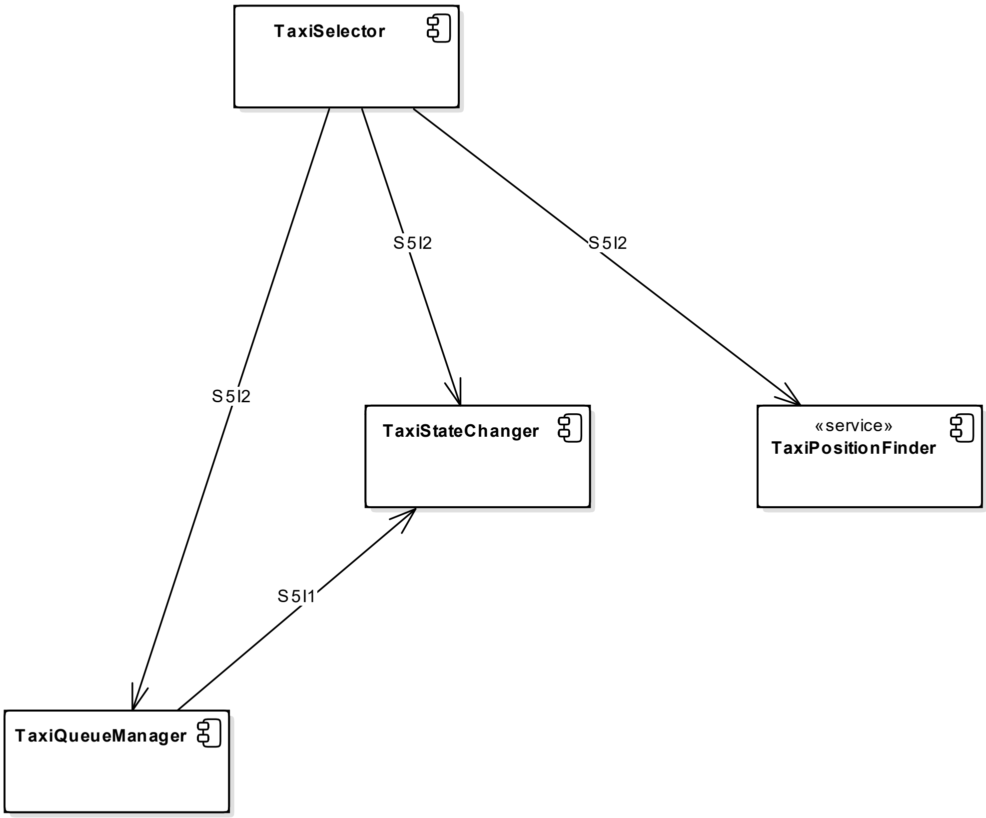

Figure 6: Integration testing plan - Taxi manager

2.3 Integration testing strategy

In this section, starting from the scenario depicted in the previous ones, we illustrate the integration strategy we have chosen providing the rational behind that choice with reference to the software architecture.

2.3.1 Preliminary considerations

Integration process in a client/server application is often a big issue. Coherently with the design and conquer principle, as already mentioned in section 2.2.1, we clearly distinguish between the integration stages at *component level* and at *subsystem level*. This first choice has several advantages.

- At the end of the first stage (integration at component level) it is guaranteed that all subsystems are correctly working, obviously this activity requires the usage of proper drivers and stubs to simulate the environment of the other subsystems.
- The integration at component level for each subsystem can be performed in parallel since they are considered isolated, reducing the project overall time and allowing working of differentiated integration test groups.
- Different integration testing strategies can be selected for the two stages according to the different needs.

2.3.2 Selected integration strategies

Considered the different needs and characteristics of the two levels of abstractions we think a proper solution should adopt different integration testing strategies.

- *Bottom-up* strategy for the integration at component level within each subsystem. The components in the lowest layer of the hierarchy are tested individually, then components belonging to the layer above are integrated and tested until the root of the hierarchy is reached. The main advantages of this approach are the following:
  - only test drivers are used to set up the testing environment and pass the test case, no test stubs are needed;
  - it is suitable for object oriented design methodologies because it starts from the low levels of hierarchies going up to the more abstract elements;
  - favors the evaluation of the performance requirements.
- *Sandwich* strategy for the integration at subsystem level. It combines top-down strategy with bottom-up strategy, favoring parallel testing. The system is viewed as having three layers: a target layer in the middle, a layer above the target and a layer below the target. Testing converges at the target layer. If there are more than three layers, as in myTaxiService, we can exploit heuristics to minimize the number of stubs and drivers; for us it is clear that we may converge towards the Business subsystem. Sandwich strategy brings the following advantages:
  - top and bottom layer tests can be done in parallel;
  - is suitable for large projects having several subprojects.

Notice that the sandwich strategy per se does not prescribe an individual testing strategy of any subsystems (which should be performed simultaneously with the subsystem integration), but since we distinguished between component level and subsystem level integration testing this is automatically implied. This strategy is often referred as *modified sandwich* strategy that, however, does not define the specific approach to be adopted within each subsystem, for us bottom-up.

2.4 Sequence of Component/Function Integration

In this section, according to the hierarchies and integration testing strategies described above we will provide the sequence of integration of components and subsystems. We will exploit the UML activity diagram in order to make the process more clear and highlight the possible parallelizations.

2.4.1 Software Integration Sequence (component level)

At component level the integration testing strategies can be applied in a fully parallelized environment, since we assume to perform integration testing within each subsystem in an isolated way; drivers and stubs for the interacting component will be necessary. Now for each subsystem we provide the sequence in which integration testing will be performed. Drivers will be necessary at each step (refer to section 4 for the detailed description) since we are proceeding bottom-up but also "external" stubs are necessary[5].

[5]Typically Bottom-up integration testing does not require any stub, however since we adopt this strategy at level of components we need to model the other subsystems, those models will be called *external stubs*. They can be real stubs or just the actual subsystems if already integrated.

2.4.2 PMA [S1]

All leaves components are supposed to be individually tested with a suitable drivers.

S1I1 PMAController → InputValidator

S1I1 PMAController → MessageFormatter

S1I1 PMAController → CCommunicator

S1I2 PMAUserInterface → PMAController

S1I3 CCommunicator → PMAUserInterface

For test S1I1 PMAController → CCommunicator an external stub for the Web subsystem is needed while, since there is a cycle between PMAController, PMAUserInterface and CCommunicator a stub for CCommunicator is needed when performing integration test S1I2 (we can actually use the same CCommunicator since it is already unit tested and integrated).

2.4.3 TMA [S2]

All leaves components are supposed to be individually tested with a suitable drivers.

S2I1 TMAController → InputValidator

S2I1 TMAController → MessageFormatter

S2I1 TMAController → CCommunicator

S2I2 TMAUserInterface → PMAController

S2I3 CCommunicator → PMAUserInterface

For test S2I1 TMAController → CCommunicator an external stub for the Web subsystem is needed while, since there is a cycle between TMAController, TMAUserInterface and CCommunicator a stub for CCommunicator is needed when performing integration test S2I2 (we can actually use the same CCommunicator since it is already unit tested and integrated).

2.4.4 Web subsystem [S3]

S3I1 MessageInterpreter → CommandEventDispatcher

S3I2 SCommunicator → MessageInterpreter

S3I3 MessageFormatter → SCommunicator

S3I4 CommandEventDispatcher → MessageFormatter

When performing all those integration test an external stub for the Business Subsystem and the client is needed.

2.4.5 TaxiManager [S5]

All leaves components are supposed to be individually tested with a suitable drivers.

S5I1	TaxiQueueManager → TaxiStateChanger
S5I2	TaxiSelector → TaxiStateChanger
S5I2	TaxiSelector → TaxiPositionFinder
S5I2	TaxiSelector → TaxiStateChanger

When performing all those integration test an external stub for the Business Subsystem is needed, since TaxiManager is one of its subcomponents.

2.4.6 Business subsystem [S4]

S4I1	TaxiManager → DBManager
S4I2	AccountManager → DBManager
S4I3	RequestManager → GoogleMapsCommunicator
S4I3	RequestManager → TaxiManager
S4I3	RequestManager → DBManager
S4I4	ReservationManager → GoogleMapsCommunicator
S4I4	ReservationManager → DBManager
S4I4	ReservationManager → RequestManager

When performing all those integration test an external stub for the DBMS is needed.

2.4.7 Subsystem Integration Sequence (subsystem level)

At subsystem level the integration is performed based on the sandwich strategy. We list the integration steps to be followed; notice that thanks to the modified sandwich strategy SI1, SI2, SI3 and SI4 are to be performed in parallel. Moreover, SI1, SI2 and SI3 go top-down so they need a stub of Business subsystem while SI4 goes bottom-up so it needs a driver of Web subsystem.

SI1	PMA ↔ Web subsystem
SI2	TMA ↔ Web subsystem
SI3	PWA ↔ Web subsystem
SI4	Business subsystem → DBMS
SI5	Web subsystem ↔ Business subsystem

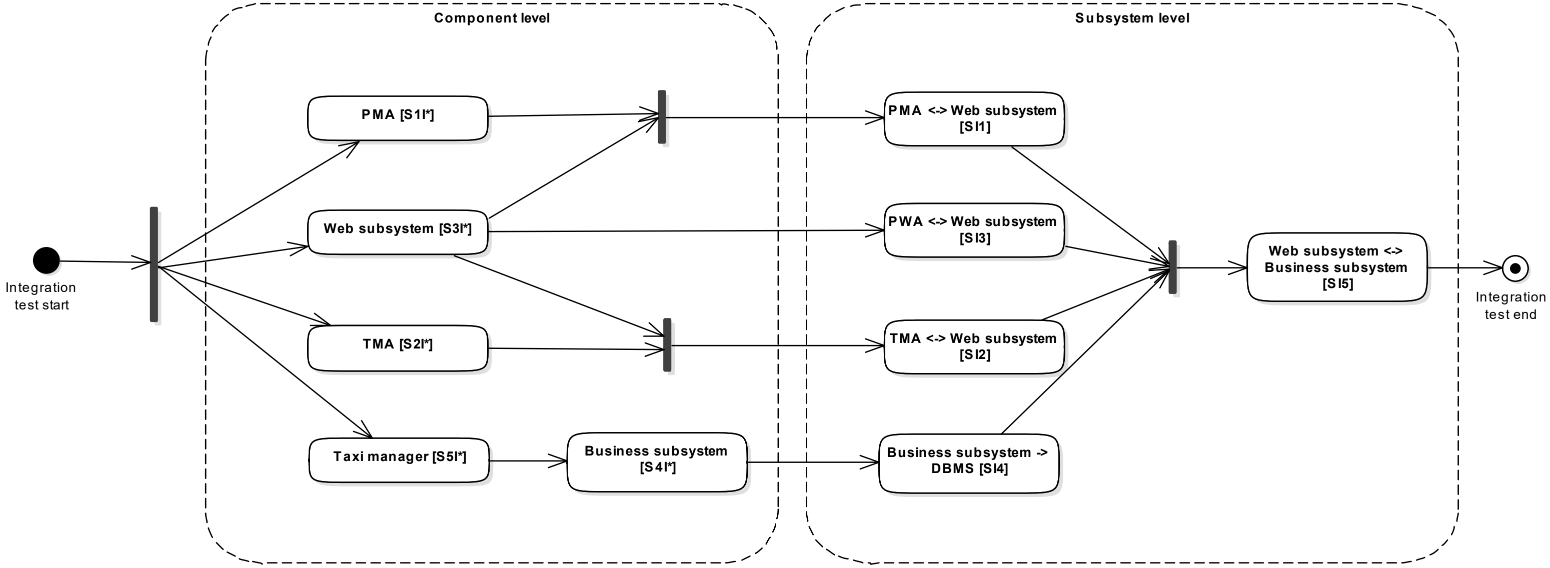

Figure 7: Integration test plan - activity diagram

3 Individual steps and test description

In this section, following the two steps depicted in the previous sections, we present the test cases we have identified according to the integration strategy chosen. For each test case we report the involved items, either components or subsystems, the input/output specification, the functional/non functional requirements tested with reference to the RASD and the DD and the suggested technique to be used.

3.1 Component level integration test

3.1.1 [S1] PMA

Test case identifier	**S1I1-T1**
Test items	PMAController MessageFormatter
Input specification	Create typical PMAController input
Output specification	Check if the correct methods are called in the MessageFormatter
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand is called on PMAController, format is called on MessageFormatter.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S1I1-T2**
Test items	PMAController InputValidator
Input specification	Create typical PMAController input
Output specification	Check if the correct methods are called in the InputValidator
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand with a register command is called on PMAController validateEmail and validateCredentials are called on InputValidator. 2. Check that when sendCommand with a login command is called on PMAController validateCredentials is called on InputValidator. 3. Check that when sendCommand with a reservation command is called on PMAController validateDate and validateTime are called on InputValidator.
Tested non functional requirements	1. (Reliability) Check that all the user input are verified by the local application before being sent to the server.
Testing technique	Automated

Test case identifier	**S1I1-T3**
Test items	PMAController CCommunicator (CommunicatorSender)
Input specification	Create typical PMAController input
Output specification	Check if the correct methods are called in the CCommunicator
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand is called on PMAController send is called on CCommunicator (CommunicatorSender interface).
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S1I2-T1**
Test items	PMAUserInterface PMAController
Input specification	Create typical PMAUserInterfaceinput
Output specification	Check if the correct methods are called in the PMAController
Environmental needs	S1I1 succeeded
Tested functional requirements	1. Check that when an action by the user is performed on the user interface the method sendCommand is called on PMAController.
Tested non functional requirements	1. (UI) Check that every functionality shall be reached surfing no more than 4 pages. 2. (Reliability) Check that the response time is smaller that 10 ms in 99% of times for local elaborations (not involving Internet connection).
Testing technique	Manual

Test case identifier	**S1I3-T1**
Test items	CCommunicator (CCommunicator) PMAUserInterface
Input specification	Create typical CCommunicator input
Output specification	Check if the correct methods are called in the PMAUserInterface
Environmental needs	S1I2 succeeded
Tested functional requirements	1. Check that when notify method is called on CCommunicator (CCommunicatorI interface) the method show is called on PMAUserInterface
Tested non functional requirements	-
Testing technique	Automated

3.1.2 [S2] TMA

Test case identifier	**S2I1-T1**
Test items	TMAController MessageFormatter
Input specification	Create typical TMAController input
Output specification	Check if the correct methods are called in the MessageFormatter
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand is called on TMAController format is called on MessageFormatter.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S2I1-T2**
Test items	TMAController InputValidator
Input specification	Create typical TMAController input
Output specification	Check if the correct methods are called in the InputValidator
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand with a login command is called on TMAController validateCredentials is called on InputValidator. 2. Check that when sendCommand with an 'input' command is called on TMAController validateDate and validateTime are called on InputValidator.
Tested non functional requirements	1. (Reliability) Check that all the user input are verified by the local application before being sent to the server.
Testing technique	Automated

Test case identifier	**S2I1-T3**
Test items	TMAController CCommunicator (CommunicatorSender)
Input specification	Create typical TMAController input
Output specification	Check if the correct methods are called in the CCommunicator
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand is called on TMAController send is called on CCommunicator.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S2I2-T1**
Test items	TMAUserInterface TMAController
Input specification	Create typical TMAUserInterface input
Output specification	Check if the correct methods are called in the TMAController
Environmental needs	S2I1 succeeded
Tested functional requirements	1. Check that when an action by the user is performed on the user interface the method sendMessage is called on TMAController.
Tested non functional requirements	1. (UI) Check that every functionality shall be reached surfing no more than 4 pages. 2. (Reliability) Check that the response time is smaller that 10 ms in 99% of times for local elaborations (not involving Internet connection).
Testing technique	Manual

Test case identifier	**S2I3-T1**
Test items	CCommunicator (CCommunicatorI) PMAUserInterface
Input specification	Create typical CCommunicator input
Output specification	Check if the correct methods are called in the PMAUserInterface
Environmental needs	S2I2 succeeded
Tested functional requirements	1. Check that when notify method is called on CCommunicator the method show is called on TMAUserInterface
Tested non functional requirements	-
Testing technique	Automated

3.1.3 [S3] WebTier

Test case identifier	**S3I1-T1**
Test items	MessageInterpreter CommandEventDispatcher (CommandDispatcher)
Input specification	Create typical MessageInterpreter input
Output specification	Check if the correct methods are called in the CommandEventDispatcher
Environmental needs	-
Tested functional requirements	1. Check that when interpretMessage is called on MessageInterpreter dispatchCommand is called on CommandEventDispatcher
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S3I2-T1**
Test items	SCommunicator (SCommunicatorI) MessageInterpreter
Input specification	Create typical SCommunicator input
Output specification	Check if the correct methods are called in the MessageInterpreter
Environmental needs	S3I1 succeeded
Tested functional requirements	1. Check that when submit is called on SCommunicator interpretMessage is called on MessageInterpreter
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S3I3-T1**
Test items	MessageFormatter SCommunicator (CommunicatorSender)
Input specification	Create typical MessageFormatter input
Output specification	Check if the correct methods are called in the SCommunicator
Environmental needs	S3I2 succeeded
Tested functional requirements	1. Check that when formatMessage is called on MessageFormatter send is called on SCommunicator.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S3I4-T1**
Test items	CommandEventDispatcher (EventDispatcher) MessageFormatter
Input specification	Create typical CommandEventDispatcher input
Output specification	Check if the correct methods are called in the MessageFormatter
Environmental needs	S3I3 succeeded
Tested functional requirements	1. Check that when dispatchEvent is called on CommandEventDispatcher formatMessage is called on MessageFormatter.
Tested non functional requirements	-
Testing technique	Automated

3.1.4 [S5] TaxiManager

Test case identifier	**S5I1-T1**
Test items	TaxiQueueManager TaxiStateChanger
Input specification	Create typical TaxiQueueManager input
Output specification	Check if the correct methods are called in the TaxiStateChanger
Environmental needs	-
Tested functional requirements	1. Check that when move is called on TaxiQueueManager changeState is called on TaxiStateChanger. 2. Check that when moveToTheEnd is called on TaxiQueueManager changeState is called on TaxiStateChanger.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S5I2-T1**
Test items	TaxiSelector TaxiStateChanger
Input specification	Create typical TaxiSelector input
Output specification	Check if the correct methods are called in the TaxiStateChanger
Environmental needs	S5I1 succeeded
Tested functional requirements	1. Check that when confirm is called on TaxiSelector changeState is called on TaxiStateChanger.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S5I2-T2**
Test items	TaxiSelector TaxiPositionFinder
Input specification	Create typical TaxiSelector input
Output specification	Check if the correct methods are called in the TaxiPositionFinder
Environmental needs	S5I1 succeeded
Tested functional requirements	1. Check that when selectTaxi is called on TaxiSelector getTaxiPosition is called on TaxiPositionFinder.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S5I2-T3**
Test items	TaxiSelector TaxiQueueManager
Input specification	Create typical TaxiSelector input
Output specification	Check if the correct methods are called in the TaxiQueueManager
Environmental needs	S5I1 succeeded
Tested functional requirements	1. Check that when reject is called on TaxiSelector moveToTheEnd is called on TaxiQueueManager.
Tested non functional requirements	-
Testing technique	Automated

3.1.5 [S4] BusinessTier

Test case identifier	**S4I1-T1**
Test items	TaxiManager DBManager
Input specification	Create typical TaxiManager input
Output specification	Check if the correct methods are called in the DBManager
Environmental needs	Taxi manager integrated and tested
Tested functional requirements	1. Check that when all methods involving DB access are called on TaxiManager, method query is called on DBManager.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I2-T1**
Test items	AccountManager DBManager
Input specification	Create typical AccountManager input
Output specification	Check if the correct methods are called in the DBManager
Environmental needs	-
Tested functional requirements	1. Check that when login, forgotPassword and register are called on AccountManager method query is called on DBManager.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I3-T1**
Test items	RequestManager DBManager
Input specification	Create typical RequestManager input
Output specification	Check if the correct methods are called in the DBManager
Environmental needs	-
Tested functional requirements	1. Check that when sendRequest, getWaitingTime and getIncomingTaxiCode are called on RequestManager method query is called on DBManager.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I3-T2**
Test items	RequestManager GoogleMapsCommunicator
Input specification	Create typical RequestManager input
Output specification	Check if the correct methods are called in the GoogleMapsCommunicator
Environmental needs	-
Tested functional requirements	1. Check that when sendRequest is called on RequestManager, validateAddress is called on GoogleMapsCommunicator. 2. Check that when getWaitingTime is called on RequestManager, getTravellingTime is called on GoogleMapsCommunicator.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I3-T3**
Test items	RequestManager TaxiManager
Input specification	Create typical RequestManager input
Output specification	Check if the correct methods are called in the TaxiManager
Environmental needs	S4I1 succeeded
Tested functional requirements	1. Check that when sendRequest is called on RequestManager, selectTaxi is called on TaxiManager.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I4-T1**
Test items	ReservationManager GoogleMapsCommunicator
Input specification	Create typical ReservationManager input
Output specification	Check if the correct methods are called in the GoogleMapsCommunicator
Environmental needs	S4I3 succeeded
Tested functional requirements	1. Check that when sendReservation is called on ReservationManager, validateAddress is called on GoogleMapsCommunicator.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I4-T2**
Test items	ReservationManager RequestManager
Input specification	Create typical ReservationManager input
Output specification	Check if the correct methods are called in the RequestManager
Environmental needs	S4I3 succeeded
Tested functional requirements	1. Check that when sendReservation is called on ReservationManager, sendRequest is called on RequestManager.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I4-T3**
Test items	ReservationManager DBManager
Input specification	Create typical ReservationManager input
Output specification	Check if the correct methods are called in the DBManager
Environmental needs	S4I3 succeeded
Tested functional requirements	1. Check that sendReservation, deleteReservation, modifyReservation, getReservation and getReservations are called on RequestManager method query is called on DBManager.
Tested non functional requirements	-
Testing technique	Automated

3.2 Subsystem level integration test

Test case identifier	**SI1-T1**
Test items	PMA → Web subsystem
Input specification	Create typical PMA input
Output specification	Check if the correct methods are called in Web subsystem
Environmental needs	PMA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when a user input is performed a message is received by Web subsystem through the network.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Manual

Test case identifier	**SI1-T2**
Test items	Web subsystem →PMA
Input specification	Create typical Web subsystem input
Output specification	Check if the correct methods are called in PMA
Environmental needs	PMA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when an event is generated as input of Web subsystem a message is recieved by PMA.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Automated

Test case identifier	**SI2-T1**
Test items	TMA → Web subsystem
Input specification	Create typical TMA input
Output specification	Check if the correct methods are called in Web subsystem
Environmental needs	TMA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when a user input is performed a message is recieved by Web subsystem through the network.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Manual

Test case identifier	**SI2-T2**
Test items	Web subsystem →TMA
Input specification	Create typical Web subsystem input
Output specification	Check if the correct methods are called in TMA
Environmental needs	TMA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when an event is generated as input of Web subsystem a message is recieved by TMA.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Automated

Test case identifier	**SI3-T1**
Test items	PWA → Web subsystem
Input specification	Create typical PWA input
Output specification	Check if the correct methods are called in Web subsystem
Environmental needs	PWA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when a user input is performed a message is recieved by Web subsystem through the network.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Manual

Test case identifier	**SI3-T2**
Test items	Web subsystem →PWA
Input specification	Create typical Web subsystem input
Output specification	Check if the correct methods are called in PWA
Environmental needs	PWA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when an event is generated as input of Web subsystem a message is received by PWA.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Automated

Test case identifier	**SI4-T1**
Test items	Business subsystem → DBMS
Input specification	Create typical Business subsystem input
Output specification	Check if the correct methods are called in DBMS
Environmental needs	Business subsystem integrated and tested
Tested functional requirements	1. Check that when a command which requires DB access is received by Business subsystem a query is performed on the DBMS.
Tested non functional requirements	1. Check that the system is a transactional system (both operations generated by users and internal operations carried out by system are transactions). 2. Check that the elaboration is carried out in less then 100 ms in 99% of times.
Testing technique	Automated

Test case identifier	**SI5-T1**
Test items	Web subsystem→ Business subsystem
Input specification	Create typical Web subsystem input
Output specification	Check if the correct methods are called in Business subsystem
Environmental needs	SI1, SI2, SI3 and SI4 succeeded
Tested functional requirements	1. Check that when a message is received by web subsystem a proper command is called on business subsystem.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**SI5-T2**
Test items	Business subsystem → Web subsystem
Input specification	Create typical Business subsystem input
Output specification	Check if the correct methods are called in Web subsystem
Environmental needs	SI1, SI2, SI3 and SI4 succeeded
Tested functional requirements	1. Check that when an event is generated as a result of a previous command it is dispatched to the web subsystem.
Tested non functional requirements	-
Testing technique	Automated

4 Tools and test equipment required

In this section we present some tools that can be useful for the integration testing described in the previous sections. Those do not represent a constraint for the integration test execution phase but just a suggestion for the testing team.

4.1 Overview

Whatever approach is chosen for the integration testing, either *manual testing* or *automated testing*, it has to be at least compatible, preferably suitable, and simple for the architecture chosen in the implementative phase. As stated in the DD, the suggested architecture is JEE so we will describe possible approaches to integration testing advised for JEE.

- *Manual integration testing* refers to the testing process performed by hand.
- *Automated integration testing* refers to the testing process performed using the facilities of ad hoc testing tools, we will describe Arquillian that the main one adopted for JEE applications.

Automated testing should be preferred for reliability, precision and time saving (as it is done in unit testing) however for integration testing it is usually not enough. Sometimes automated tests cannot spot all forms of unexpected error conditions which can manifest only by a direct intervention of the developer who knows the dependency structure of the application.

4.2 Possible approaches

Mainly taken from [5].

4.2.1 Manual integration testing

Manual integration testing consists in the process of discovering defects in the interaction between components/subsystems by means of exercising the software with proper input from test cases and comparing the output with the expected output. Manual testing is typically slower and less reliable then automated testing however the "100% Automation is not possible" especially in integration testing so manual testing plays an important role in this context. Often there are not automatic tools to test the user interface therefore manual testing can be exploited.

4.2.2 Automated integration testing: Arquillian

Integration testing is very important in Java EE. The reason is two-fold:

- business components often interact with resources or sub-system provided by the container;
- many declarative services get applied to the business component at runtime.

Therefore to do integration tests on a JEE application it requires that you run them inside a JEE container.

Fortunately this problem is solved by an open source project called Arquillian (`http://www.jboss.org/arquillian`) which boots the container, allows the injection of managed beans and EJB beans into unit test classes. Arquillian is a *container-oriented testing framework* developed in Java Enterprise that brings your test to the runtime rather than requiring you to manage the runtime from the test. This strategy eliminates setup code and allows the test to behave more like the components it's testing. The end result is that integration testing becomes no more complex than unit testing.

Arquillian combines a unit testing framework (JUnit or TestNG), ShrinkWrap, and one or more supported target containers (Java EE container, servlet container, Java SE CDI environment, etc) to provide a simple, flexible and pluggable integration testing environment. Beside Arquillian functionalities JUnit features (like assertions, annotations...) can be still exploited in order to write ad-hoc integration testing procedures. Other tools mainly intended to be used as unit test facilities can be exploited to build the necessary stub, one for all Mockito.

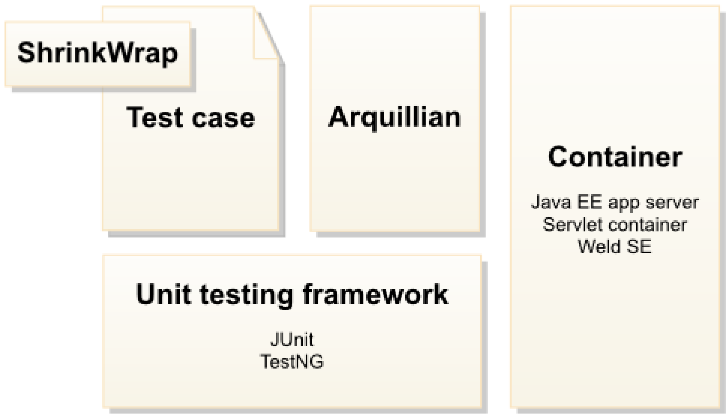

Figure 8: Arquillian architecture.

4.3 Suggested process

In order to exploit as much as possible the advantages, in terms of time and quality of results, of the automated integration testing we suggest to exploit manual testing only when strictly necessary, for instance in case of user interface integration testing. For all the other cases generating test data would be possible therefore automated tasting can be used. In section 3, for each test case, the suggested technique is specified.

5 Program stubs and test data required

5.1 Program drivers and stubs

In sections 2 and 3 we discussed the integration test strategies and the elements, either components or subsystems, to be integrated; we mentioned the need to use driver and stubs in order to set up the environment in which perform the test activity. In this section we present in details this infrastructure with reference to the test cases previously defined.

5.1.1 Component level integration testing

The strategy adopted for component level integration testing is *bottom-up* therefore drivers are necessary at each step of integration, in addition some stubs has to be used to simulate the functionalists of the other subsystems.

5.1.1.1 Drivers

Needed for each test case.

5.1.1.2 Stubs

Test id	*Items*	*Stubs*	*Interfaces*
S1I1	PMAController → CCommunicator	Web subsystem	SCommunicatorI
S1I2	PMAUserInterface → PMAController	CCommunicator	CCommunicatorI
S2I1	PMAController → CCommunicator	Web subsystem	SCommunicatorI
S2I2	PMAUserInterface → PMAController	CCommunicator	CCommunicatorI
S3I1	MessageInterpreter → CommandEventDispatcher	Business subsystem	RequestManager, ReservationManager, AccountManager, TaxiManager
S3I3	MessageFormatter → SCommunicator	PMA, PMA	CCommunicatorI
S5I1	TaxiQueueManager → TaxiStateChanger	Business subsystem	DBManager
S5I2	TaxiSelector → TaxiStateChanger	Business subsystem	DBManager
S5I2	TaxiSelector → TaxiStateChanger	Business subsystem	DBManager
S4I1	TaxiManager → DBManager	DBMS	DBMSConnector
S4I2	AccountManager → DBManager	DBMS	DBMSConnector
S4I3	RequestManager → DBManager	DBMS	DBMSConnector
S4I4	ReservationManager → DBManager	DBMS	DBMSConnector

5.1.2 Subsystem level integration testing

The strategy adopted for component level integration testing is *sandwich* therefore both stubs and drivers are necessary at each step of integration.

5.1.2.1 Drivers

Test id	*Items*	*Driver*	*Functionality*
SI1	PMA → Web subsystem	-	-
SI1	Web subsystem → PMA	Business subsystem	Call methods of Web subsystem
SI2	TMA → Web subsystem	-	-
SI2	Web subsystem → TMA	Business subsystem	Call methods of Web subsystem
SI3	PWA →Web subsystem	-	-
SI3	Web subsystem → PWA	Business subsystem	Call methods of Web subsystem
SI4	Business subsystem → DBMS	Web subsystem	Call methods of Business subsystem
SI5	Web subsystem → Business subsystem	-	-
SI5	Business subsystem → Web subsystem	-	-

5.1.2.2 Stubs

Test id	*Items*	*Stubs*	*Interfaces*
SI1	PMA → Web subsystem	Business subsystem	RequestManager, ReservationManager, AccountManager, TaxiManager
SI1	Web subsystem → PMA	-	-
SI2	TMA → Web subsystem	Business subsystem	RequestManager, ReservationManager, AccountManager, TaxiManager
SI2	Web subsystem → TMA	-	-
SI3	PWA → Web subsystem	Business subsystem	RequestManager, ReservationManager, AccountManager, TaxiManager
SI3	Web subsystem → PWA	-	-
SI4	Business subsystem → DBMS	-	-
SI5	Web subsystem → Business subsystem	-	-
SI5	Business subsystem → Web subsystem	-	-

5.2 Test data requirements (external system integration)

In this subsection we briefly discuss the problem of integration testing with external subsystems, in particular we will focus on the integration with the GPS system in the smartphone of the passenger, the GoogleMapsAPI for address retrieval and the GPS system installed on taxis. For simplicity we assume those integration testing are performed before the integration testing of the system.

EI1 PMAController → GPSInterface

EI2 PMAController → GoogleMapsAPI

EI3 TMAController → GoogleMapsAPI

EI4 GoogleMapsCommunicator → GoogleMapsAPI

EI5 TaxiPositionFinder → TaxiGPSInterface

A Appendix

Used tools

1. LyX visual editor for LaTeX (`http://www.lyx.org/`) to write this document.
2. Enterprise Architect 11 (`http://www.sparxsystems.com.au/products/ea/`) for UML diagrams.

Hours of works

Time spent by each group member:

- Alberto Maria Metelli: 12h
- Riccardo Mologni: 12h

Revision history

Version	*Date*	*Revision description*	*Revision notes*
0.1	15-1-2016	Initial draft	-
1.0	21-1-2016	Final draft	-
2.0	22-2-2016	Final release	Fixed introduction and some terminology.