Integration Test Plan Document - Software Engineering 2

PowerEnjoy

Marini Alberto
862838
alberto2.marini@mail.polimi.it

Marrone Matteo
810840
matteo.marrone@mail.polimi.it

Sabatelli Antonella
875666
antonella.sabatelli@mail.polimi.it

January 15th, 2017

Politecnico di Milano

Contents

Chapter 1

1 Introduction

1.1 Purpose and Scope

This document is the Integration **Test Plan Document (ITPD)** for our *PowerEnjoy* app. Its purpose is to determine how to accomplish the integration test of the software, which tools are to be used and which approach will be followed.

PowerEnjoy is an app for electric car sharing in Milan area. Its goal is to grant the possibility of reserving and using a car. *PowerEnjoy* guarantees a smart fair management, by respecting the service rules.

1.2 List of Definitions and Abbreviations

- **RASD**: Requirements Analysis and Specification Document.
- **DD**: Design Document.
- **ITPD**: Integration Test Plan Document.
- **DB**: DataBase.
- **DBMS**: DataBase Manager System.

1.3 List of Reference Documents

- Requirements Analysis and Specification Document (RASD) PowerEnjoy - *Marini, Marrone, Sabatelli*
- Design Document (DD) PowerEnjoy - *Marini, Marrone, Sabatelli*

Chapter 2

2 Integration Strategy

2.1 Entry Criteria

Before integration testing can start, the project should be code-complete, that is, devoid of any missing parts, and JUnit tests should have been developed for every module of the software, possibly covering 85% or more of the lines of code and on of standard white-box testing approaches such as path testing; latest releases of documentation to be used as reference for the integration testing phase (see References section), including this document, should be made available to the developers, as well.

It is assumed that the integration testing will be carried out only after that all testing tools referenced in *Chapter 4* will have been properly installed and configured, making use of hardware components possessing the characteristics described in the Design Document (see References section).

2.2 Elements to be Integrated

The integration process will take place on two levels: the components one and the subsystems one. The main components whose functionality we are interested in testing are the Enterprise Java Beans defining the business logic within the second tier of the system, the one containing the application server that well mark as a subsystem. Well focus in particular on the User Manager, the Search Manager, the Reservation Manager, the Ride Manager and the Communication Manager. As mentioned in the Design Document, the data tier and the car software will be acquired from external agents, with the exclusion of some customisations of the car software added ad hoc, yet they are to be integrated and will be marked as two different subsystems as well. The client tier, where the GUI is located, will be the final subsystem to integrate.

2.3 Integration Testing Strategy

Well make us of a bottom-up approach, beginning from the integration of the smallest core components of the business logic to then proceed with the integration of the subsystem containing them with the other subsystems previously mentioned.

This approach will prove more advantageous than a top-down one since we believe the development of stubs mimicking the modules not yet integrated would be more complex than the one of the driver mocking the skeletal structure of the system, and that the observation of the test results would be easier as we'll.

Finally, in our experience, bugs are more easily located as we'll as fixed when the modules containing them are considered *per se*, rather than when they are already attached to a system in the course of development.

2.4 Sequence of Component/Function Integration

2.4.1 Software Integration Sequence

The component will be tested from the most independent to the least independent, in order to minimize the need for stubs and drivers. The details regarding our reasoning over the relative independence between modules will be further exposed in Chapter 3.

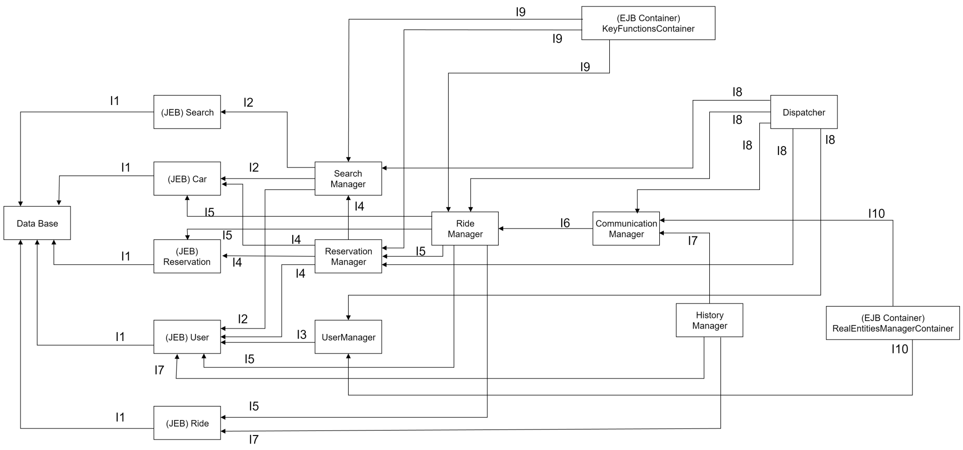

Figure 1: Diagram of the components integration

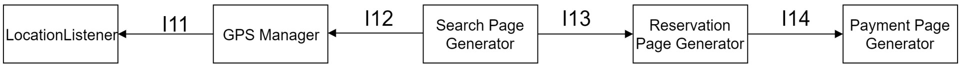

Figure 2: Diagram of the components integration

2.4.2 Subsystem Integration Sequence

Integration between subsystems will be carried out as soon as the modules composing the subsystems will be integrated themselves, as we think that integrating top modules when smaller ones are still in need of development will give us a better idea of how the system is coming out, helping us detect more easily and early where potential faults are. Application server and database, being more central to the system functioning, will be tested before the client and the car. The mobile will be tested before the car, both alone and integrated, as many user functions do not require the car to actually exist to be tested.

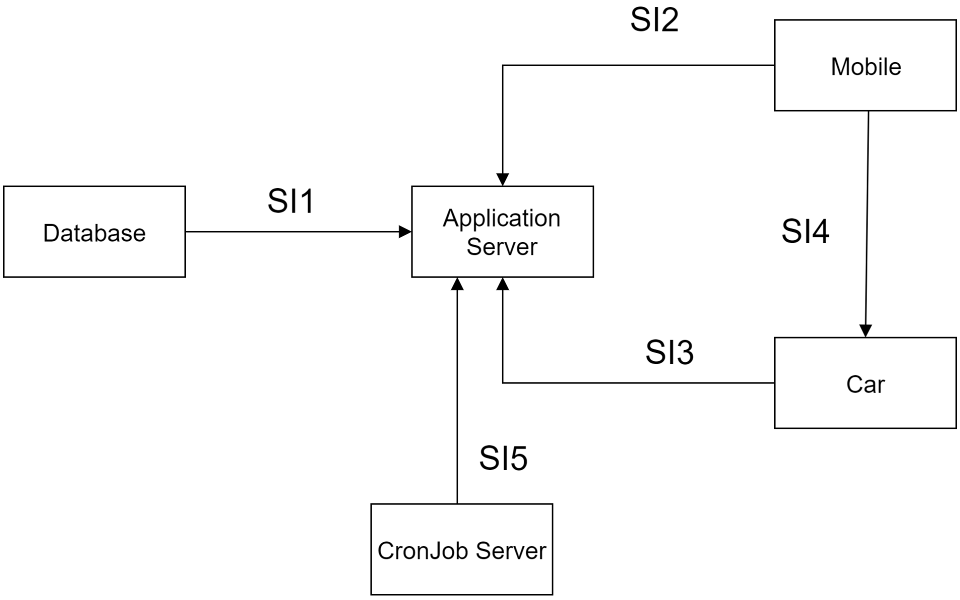

Figure 3: Diagram of the subsystems integration

Table 1: Integration order of the system components

N.	Subsystem	Component	Integrates with
I1	Database, Application Server	(JEB) Search	DBMS
I1	Database, Application Server	(JEB) User	DBMS
I1	Database, Application Server	(JEB) Car	DBMS
I1	Database, Application Server	(JEB) Reservation	DBMS
I1	Database, Application Server	(JEB) Ride	DBMS
I2	Application Server	Search Manager	Search, User, Car (EB), Map API
I3	Application Server	User Manager	User
I4	Application Server	Reservation Manager	Search Manager, Reservation, Car (EB), User
I5	Application Server	Ride Manager	Reservation Manager, Car (EB), User, Reservation, Ride
I6	Application Server	Communication Manager	Ride Manager
I7	Application Server	History Manager	Communication Manager, Ride, User
I8	Application Server	Dispatcher	Search Manager, Reservation manager, Ride Manager, Communication Manager, User Manager, GlassFish Server
I9	Application Server	KeyFuctions Container	Search Manager, Reservation Manager, Ride Manager
I10	Application Server	RealEntities Container	Communication Manager, User Manager
I11	Mobile	GPS Manager	LocationListener
I12	Mobile	Search Page Generator	GPS Manager
I13	Mobile	Search Page Generator	Reservation Page Generator
I14	Mobile	Reservation Page Generator	Payment Page Generator

Table 2: Integration order of the subsystems

N.	**Subsystem**	**Integrates with**
SI1	Application Server	Database
SI2	Mobile	Application Server
SI3	Car	Application Server
SI4	Mobile	Car
SI5	Cronjob Server	Application Server

Chapter 3

3 Individual Steps and Test Description

This chapter describes the individual test cases to be executed. Each test case is identified with a code and is directly mapped with Table 1 for the integration between components and with Table 2 for the integration between subsystems.
Test cases whose code starts with SI are integration tests between subsystems; test cases whose code starts with I are integration tests between components.

3.1 Integration test case I1

Table 3: Test case I1

Test Case Identifier	I1
Test Items	entity beans –>Database
Input Specifications	Typical queries on table Search, User, Car (EB), Reservation, Ride
Output Specification	For each entity bean involved in one of these tests we will check whether the invoking of the methods associated to it will lead to DBMS queries working as expected after their activation by the method.
Environmental Needs	Glassfish server,Test Database, Complete implementation of the Java Entity Beans, Java Persistence API
Testing methods	Automated with JUnit

Taking into account that a reservation is made after a search was carried out, and that whenever a reservation is made user's history is consequently updated through user manager methods, and finally that for a ride to be defined a previous reservation must exist, the first four managers to be tested and integrated will be searchManager, UserManger, ReservationManager and RideManager, in this order.

3.2 Integration test case I2

Table 4: Test case I2

Test Case Identifier	I2

Test Items	searchManager—>Search, User, Car (EB), Maps API
Input Specifications	SearchManager methods call requesting a search to be created and handled.
Output Specification	The purpose of this test is to determine if, once received info regarding the localization fo the user the searchManager returns a map where the locations of the car near to the user are displayed correctly.
Environmental Needs	Glassfish server
Testing methods	JUnit

3.3 Integration test case I3

Table 5: Test case I3

Test Case Identifier	I3
Test Items	userManager—>User
Input Specifications	Call of methods involving the creation of a new user due to the submitting of a registration form, either correctly or incorrectly filled.
Output Specification	The test will end with the creation of a new instance of the User class if the parameters satisfy the registration constraints, with a detailed notification of failure in case some data does not, on the contrary, satisfies such constraints.
Environmental Needs	Glassfish server
Testing methods	JUnit

3.4 Integration test case I4

Table 6: Test case I4

Test Case Identifier	I4
Test Items	ReservationManager—>SearchManager, Reservation, Car (EB), User
Input Specifications	Perception of a selection of a car by the user in the context of a search, a car which can either be already reserved or available.

Output Specification	Creation of a new instance of a reservation object if the selected car was available at the moment of the selection and update of the car status, detailed notification of failure in case that same car was unfortunately reserved by another user having carried out a search at the same time. The test will check also if the car status stayed the same in the second case.
Environmental Needs	Glassfish server
Testing methods	JUnit

3.5 Integration test case I5

Table 7: Test case I5

Test Case Identifier	I5
Test Items	RideManager—>ReservationManager, Car, User , Reservation, Ride
Input Specifications	Invocation of manage ride beginning method, and eventually of ManageMSO method, taking in parameters related to a dummy car having signalled the user who had reserved it having reached and unlocked it within the time allotted.
Output Specification	The test will check whether a ride will have been properly created or initialized (according to the input) and whether the car status will have been properly updated to being driven and the MSO will have been correctly initialized if selected at the end of the test.
Environmental Needs	Glassfish server
Testing methods	JUnit

3.6 Integration test case I6

Table 8: Test case I6

Test Case Identifier	I6
Test Items	CommunicationManager—>Ride Manager
Input Specifications	Calls of methods such as ManageSensorInfo provided with dummy parameters.
Output Specification	The test aim is in first place to determine if in analyzing the information collected by the cars sensors the communicationManager prompts the sending of the correct requests towards the other managers , and if once certain processes, such as MSO Management, have been triggered, the correct info is returned by the method towards cars and safe areas (represented though appropriate dummys).
Environmental Needs	Glassfish server, Drivers for cars/safe areas
Testing methods	JUnit

As pointed out in the Design Document, the History Manager is an interface whose methods purpose is mainly to build and analyze records of usage data to draw out economically significant statistic conclusions, so among the main managers it will be the last to be integrated.

3.7 Integration test case I7

Table 9: Test case I7

Test Case Identifier	I7
Test Items	HistoryManager—>CommunicationManager, Ride, User
Input Specifications	Triggering of the periodic procedure within the HistoryManager responsible for the update of the records.
Output Specification	The test will check whether dramatic changes in the usage data will be correctly matched by an adequate evolution of the Histories and the history analysis by the HistoryManager.
Environmental Needs	Glassfish server
Testing methods	JUnit

3.8 Integration test case I8

Table 10: Test case I8

Test Case Identifier	I8
Test Items	Dispatcher—>SearchManager, ReservationManager, RideManager, CommunicationManager, UserManager, GlassFishServer
Input Specifications	A heavy load of REST requests of different typologies is simulated.
Output Specification	The dispatcher must properly address the requests directed to it by correctly redirecting each request to its intended manager. The test will check also whether the values defined for efficiency parameters such as MTBF in the RASD nonfunctional requirements section stand when the methods are tested within the intended completed environment.
Environmental Needs	Glassfish server
Testing methods	Apache JMeter, Arquillian

3.9 Integration test case I9

Table 11: Test case I9

Test Case Identifier	I9
Test Items	KeyFunctionsContainer—>SearchManager, ReservationManager, RideManager
Input Specifications	Requests toward the Managers.
Output Specification	This test aim is to check whether in presence of a significant number of requests the SearchManager, ReservationManager, RideManager keep providing the expected output even when run within the container planned for them, with no concurrency trouble.
Environmental Needs	Glassfish server
Testing methods	jUnit, Arquillian

3.10 Integration test case I10

Table 12: Test case I10

Test Case Identifier	I10
Test Items	RealEntitiesManagers Container—>CommunicationManager, UserManager
Input Specifications	Requests toward the Managers.
Output Specification	This test aim is to check whether the UserManager and the CommunicationManager keep working in the expected way the expected output even when run within the container planned for them, with no concurrency trouble.
Environmental Needs	Glassfish server
Testing methods	jUnit, Arquillian

3.11 Integration subsystem case S1

Table 13: Subsystem case S1

Test Case Identifier	S1
Test Items	Application Server—>Database
Input Specifications	Standard communication input from the App server towards the database.

Output Specification	We will observe if the queries from the application server toward the database will be carried out according to the permits associated to each request of data reading or manipulation.
Environmental Needs	Test related to the subsystems in the question must have succeeded
Testing methods	Junit

As regards the following, the corresponding test cases for iOS are very similar, but the CoreLocation Framework, Xcode and an iOS Simulator are used instead of Location Listener and the Android emulator

3.12 Integration test case I11

Table 14: Test case I11

Test Case Identifier	I11
Test Items	GPSManager—>LocationListener
Input Specifications	Calls to methods of the AndroidLocation library to get the user location.
Output Specification	It will be verified whether the GPSManager gets the correct user position or if an error is returned.
Environmental Needs	Android Emulator
Testing methods	Automated (Android testing suite)

3.13 Integration test case I12

Table 15: Test case I12

Test Case Identifier	I12
Test Items	Search Page generator—>GPS Manager
Input Specifications	Calls to GPSManager methods to get the users location.
Output Specification	If the user has given his consent for the app to infer its position from the GPS Manager the location data shall be returned, otherwise the user will be presented the choice to either enable the app to interact with the GPS Manager or to provide an address.
Environmental Needs	Android Emulator

Testing methods	Automated Android testing suite

3.14 Integration test case I13

Table 16: Test case I13

Test Case Identifier	I13
Test Items	Search Page generator—>Reservation Page generator
Input Specifications	Request to display the reservation page after a car has been selected within the search page.
Output Specification	We expect to reservation page generator to display a reservation window where details regarding the reservation timers and the selected car position are displayed.
Environmental Needs	Android Emulator
Testing methods	Android Testing Suite, Manual Testing

3.15 Integration test case I14

Table 17: Test case I14

Test Case Identifier	I14
Test Items	Reservation Page Generator—>Payment Page Generator
Input Specifications	Request to display the payment page after a car has been successfully parked in a safe area, the user has gotten out of the car and 5 minutes have been from then.
Output Specification	We expect the payment page to display the final price of the transaction as well as the discounts and the penalties that were applied on the base price for the final price to be such.
Environmental Needs	Android Emulator
Testing methods	Android Testing Suite, Manual Testing

3.16 Integration subsystem case S2

Table 18: Subsystem case S2

Test Case Identifier	S2
Test Items	Mobile—>Application Server

Input Specifications	Typical API calls to the business tier (REST API).
Output Specification	The business tier shall respond accordingly to the API specication.
Environmental Needs	The mobile application interface and the application server as well should be already completely developed
Testing methods	Automated with JUnit

3.17 Integration subsystem case S3

Table 19: Subsystem case S3

Test Case Identifier	S3
Test Items	Car—>Application Server
Input Specifications	Notifications forwarded from the car towards the application server containing info gathered by the car sensors.
Output Specification	The application Server, and in particular the communication Manager these messages are dispatched to, should correctly handle the requests initiating procedures such as the one concerning the managing of the end of the ride.
Environmental Needs	The application server should be completely developed
Testing methods	Snapdragon automotive development platform

3.18 Integration subsystem case S4

Table 20: Subsystem case S4

Test Case Identifier	S4
Test Items	Mobile —>Car
Input Specifications	Attempt by mobile application to establish a Bluetooth connection with the car through Android.Bluetooth Package methods.
Output Specification	The car should open if the pairing code in possession of the mobile device matches its own received at the moment of the reservation, otherwise it should remain close.

Environmental Needs	The mobile application interface and the application server as well should be already completely developed
Testing methods	Snapdragon automotive development platform

3.19 Integration subsystem case S5

Table 21: Subsystem case S5

Test Case Identifier	S5
Test Items	CronJob Server—>Application Server
Input Specifications	Some cars whose status is in charge with a battery level greater than 75% are mocked.
Output Specification	The cronjob is expected to update such cars status to available.
Environmental Needs	The application server should be completely developed
Testing methods	Automated with JUnit

Chapter 4

4 Tools and Test Equipment Required

The main tools used in the context of the integration testing and already mentioned in the previous sections, are JUnit, Arquillian, Mockito and Apache JMeter.

- **JUnit** is the most commonly used framework for unit testing in the Java Environment and it can be used (we plan to use it for this purpose, also) for integration testing as well, though unit tests and integration tests should not be run concurrently.

- **Arquillian** is a testing platform for the Java Virtual Machine (JVM) that allows for tests to be run against the selected target container,for instance in our case Glassfish Server. Since it deals on its own with the tasks of container management and deployment it greatly simplifies integration testing.

- **Mockito** is an open-source test framework useful to generate mock objects, stubs and, what we need more in this context since we make use of a bottom-up approach, drivers.

- **Apache JMeter** will be used to check whether the software will actually be able to achieve the performance expected at previous stages of the project, when non-functional requirements were defined (see RASD); for example, it can be exploited to simulate the response of the application server in remote distributed execution when subjected to a heavy load of requests and see if the MTBF expected stands.

Manual testing will be carried out as well for the more sensitive functions.

Chapter 5

5 Program Stubs and Test Data Required

As stated before we chose the bottom-up approach as our integration testing strategy, so we do not need to define any stub; on the other end well define a driver for each and every java entity bean while the business tier is not fully developed, as well as a mock Maps API provider through Mockito.

Chapter 6

6 Effort Spent

6.1 Working hours

- Marini Alberto: 16 hours
- Marrone Matteo: 16 hours
- Sabatelli Antonella: 16 hours