Software Engineering 2 Project: PowerEnJoy

Integration **T**est **P**lan **D**ocument

Marco Ieni, Francesco Lamonaca, Marco Miglionico
Politecnico di Milano, A.A. 2016/2017

January 15, 2017
v1.0

Contents

Chapter 1

Introduction

1.1 Purpose and Scope

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

Integration testing is a fundamental activity to guarantee that all the different subsystems of PowerEnJoy interoperate consistently with the requirements they are supposed to fulfil and without unexpected behaviours.

In the following sections we are going to provide:

- A list of the subsystems and their subcomponents involved in the integration activity that will be tested;
- The criteria that must be met by the project status before integration testing of the outlined elements may begin;
- A description of the integration testing approach and the reasoning behind it;
- The sequence in which the different subsystems will be integrated;
- A description of the planned testing activities for each integration step, including their input data and the expected output;
- Some performance measures that should be performed on the subsystems to check that they are fulfilling the requirements;
- A list of all the tools that will have to be employed during the testing activities, together with a description of the operational environment in which the tests will be executed.

1.2 List of Definitions and Acronyms

1.2.1 Definitions

- **Subcomponent**: each of the low level components realizing the functionalities of a component.
- **Subsystems**: a high-level functional unit of the system.

1.2.2 Acronyms

- **RASD:** Requirements Analysis and Specification Document.
- **DD:** Design Document.
- **ITPD:** Integration Test Plan Document (this document).
- **DBMS:** Database Management System.
- **API:** Application Programming Interface.
- **GPS:** Global Positioning System.

1.3 Reference Documents

This document refers to the following documents:

- Project goal, schedule and rules of the Software Engineering 2 project;
- PowerEnJoy Requirement Analysis and Specification Document: rasd.pdf;
- PowerEnJoy Design Document: dd.pdf;
- The Integration Test Plan Example document: Integration Testing Example Document.pdf;

Chapter 2

Integration Strategy

In this chapter the integration strategy will be described. In the section 2.1 the prerequisites for the tests will be presented. The section 2.2 is dedicated to the required subsystems to be integrated in the system to execute some tests. Finally, in the sections 2.3 and 2.4 the strategy used to test the integrations will be discussed, paying attention to the order.

2.1 Entry Criteria

This section describes the prerequisites that need to be met before integration testing can start and produce meaningful results. All the classes and methods must pass thorough **unit tests**, which should reasonably discover major issues in the structure of the classes or in the implementation of the algorithms. Unit tests should have a minimum coverage of 90% of the lines of code and should be run automatically at each build using JUnit. Unit testing is not in the scope of this document and will not be specified in further detail.

Moreover, the **documentation** of all classes and functions, written using JavaDoc, has to be complete and up-to-date, in order to be used as a reference for integration testing development. In particular, the public interfaces of each class and module should be well specified. Where necessary, a formal specification language can be used. The following **documents** must have been fully written before integration testing can begin:

- **Requirement Analysis and Specification Document of PowerEnJoy**;
- **Design Document of PowerEnJoy**;

- **Integration Testing Plan Document of PowerEnJoy** (this document).

This a required phase to have a complete picture of the interactions between the different components of the system and of the functionalities they offer.

Finally, the integration testing phase can start also if some components don't have the minimum completion percentage necessary to consider it for integration (90%), this is to reflect their order of integration and to take into account the required time to fully perform integration testing.

2.2 Elements to be Integrated

In the following paragraph we are going to provide a list of all the components that need to be integrated together.

The integration process of our software is performed on two levels.

1. **Low Level:** Integration of the different subcomponents (classes, Java Beans) inside the same subsystem;
2. **High Level:** Integration of different subsystems.

The first step needs to be performed only for the components which contains the pieces of software that we are going to develop, namely the business tier, the mobile application in the client tier.

In particular the main subcomponents that we will integrate are:

- **User Manager:** This subcomponent includes Registration, Login, Edit Profile and User Banning;
- **Email Sender**;
- **Operator Safe Area Manager:** This subcomponent includes Add Safe Area, Delete Safe are, Add Power Grid Station and Delete Power Grid Station;
- **User Safe Area Manager**;
- **Ride Manager:** This subcomponent includes Update Ride Status and End Ride;
- **Car Manager:** This subcomponent includes Reserve Car, Update Position and Unlock Car;

- **Fee Manager:** This subcomponent include Enable Saving Money Option and Disable Saving Money Option;
- **Payment Manager:** This subcomponent include Pay Ride and Check Payment Info.

The second step needs to be performed on the three major high-level components that we outlined in the Design Document, that correspond to the tiers of the system, which – from now on – will be referred to as subsystems:

- **Client tier:** The client tier consists of our mobile application, the operator terminal and the on-board tablet.
- **Business tier:** This subsystem implements all the application logic, communicates with the front-ends and the external systems.
- **Database tier:** This is the DBMS, it is not part of the software to be developed, but has to be integrated.

It is important to underline that the business tier needs to be integrated with some external systems that we will consider as black box, because we are not going to develop them but just to integrate them.

These external systems are the following:

- **Handy Car Board**;
- **External System For Driving License Validation**;
- **External System For Payments**.

2.3 Integration Testing Strategy

The integration strategy that we are going to implement is the bottom-up approach. This choice comes natural since we assume we already have the unit tests for the smallest components, so we can proceed from the bottom. In this way, we will start integrating together those components that do not depend on other components to operate, or that only depend on already developed components.

The main advantages of this approach are the possibility to perform integration tests on components that are almost fully developed, to obtain feedback on how the system will react and fail in real world situations. The other advantage is that we can start testing the components following the development process, in this way we can reduce time and maximize parallelism.

Moreover, the higher-level subsystems outlined in section 2.2 are well separated and loosely coupled since they correspond to different tiers. They also communicate through well-defined interfaces (RESTful API), so they will not be hard to integrate at a later time.

As we said before the **Handy Car Board**, the **External System For Driving License Validation**, the **External System For Payments** and the **DBMS** are considered as black box, that have already been developed and that can be immediately used in a bottom-up approach.

2.4 Sequence of Component/Function Integration

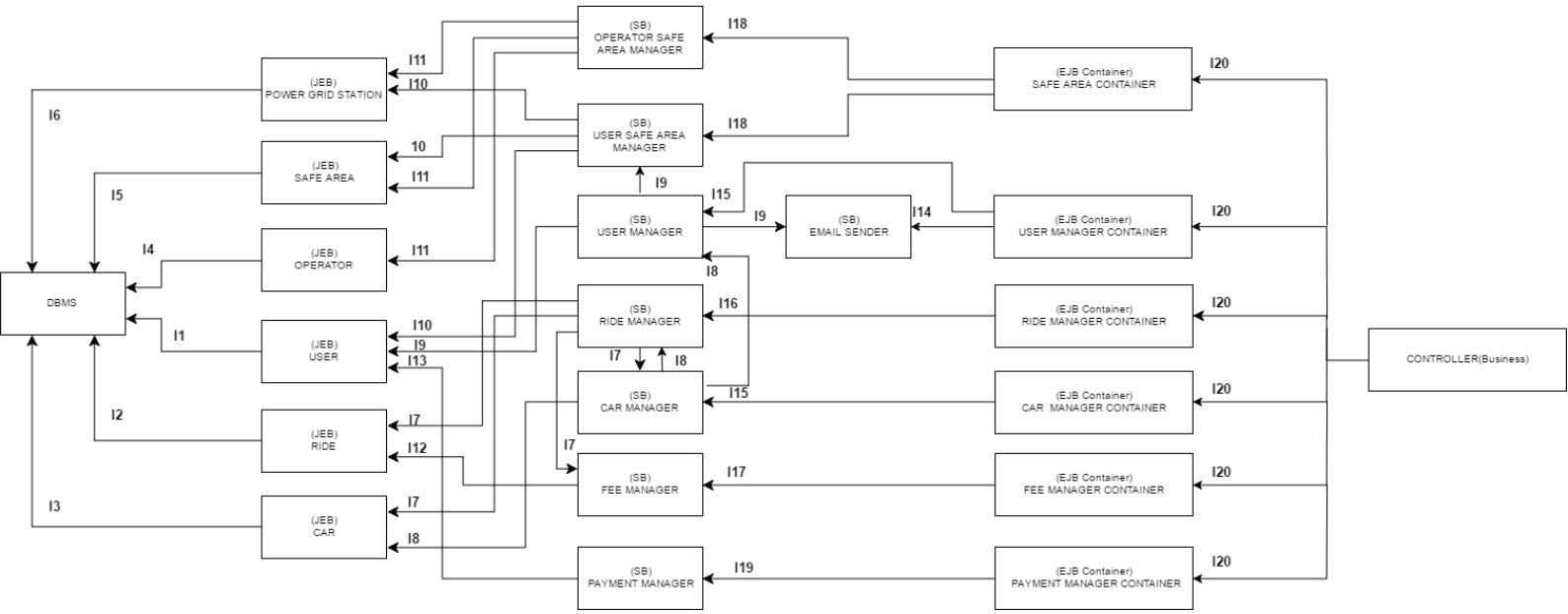
2.4.1 Software Integration Sequence

In this section we are going to describe the order of integration (and integration testing) of the various components of PowerEnJoy.

The components are tested starting from the most independent to the less one. This because when less independent components are tested, the components which they rely on have already been integrated. The components are integrated within their classes in order to create an integrated subsystem which is ready for subsystem integration.

As a notation, an arrow going from component C1 to component C2 means that C1 is necessary for C2 to function and so it must have already been implemented. The integration sequence of the components is described in the following figure and table:

COMPONENTS INTEGRATION DIAGRAM



N.	Subsystem	Component	Integrates with
I1	Database,Business	(JEB)User	DBMS
I2	Database,Business	(JEB)Ride	DBMS
I3	Database,Business	(JEB)Car	DBMS
I4	Database,Business	(JEB)Operator	DBMS
I5	Database,Business	(JEB)Safe Area	DBMS
I6	Database,Business	(JEB)Power Grid Station	DBMS
I7	Business	(SB)Ride Manager	Car Manager Fee Manager Ride Car
I8	Business	(SB)Car Manager	Car Ride Manager
I9	Business	(SB)User Manager	User User Safe Area Manager Email Sender
I10	Business	(SB)User Safe Area Manager	Safe Area User Power Grid Station
I11	Business	(SB)Operator Safe Area Manager	Safe Area Operator Power Grid Station User Safe Area Manager
I12	Business	(SB)Fee Manager	Ride
I13	Business	(SB)Payment Manager	User
I14	Business	(EJB Container)Safe Area Container	Operator Safe Area Manager User Safe Area Manager
I15	Business	(EJB Container)User Manager Container	Email Sender User Manager
I16	Business	(EJB Container)Ride Manager Container	Ride Manager
I17	Business	(EJB Container)Car Manager Container	Car Manager

I18	Business	(EJB Container)Fee Manager Container	Fee Manager
I19	Business	(EJB Container)Payment Manager Container	Payment Manager
I20	Business	Controller	Safe Area Container User Manager Container Ride Manager Container Car Manager Container Car Manager Container Fee Manager Container Payment Manager Container

2.4.2 Subsystem Integration Sequence

A choice was made to proceed with the integration process from the database tier to the business tier and finally to the client tier. The reason to do so is that in order to have a functioning client you need to have a working business tier. The business tier, instead, can be tested without any client. The integration sequence of the subsystems is described in the following table and figure 2.1:

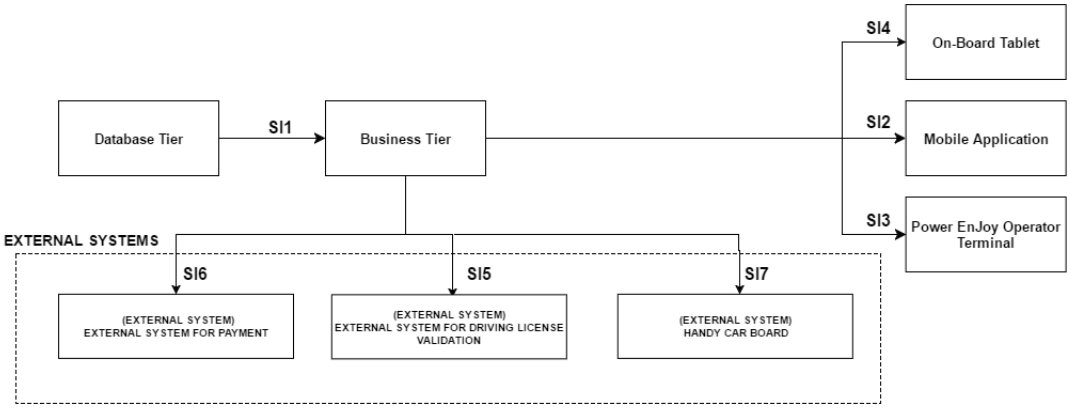


Figure 2.1: Order of integration of the subsystems.

N.	**Subsystem**	**Integrates with**
SI1	Business Tier	Database Tier
SI2	Mobile Application	Business Tier
SI3	Power EnJoy Operator Terminal	Business Tier
SI4	On-Board Tablet	Business Tier
SI5	External System For Payment	Business Tier
SI6	External System For Driving License Validation	Business Tier
SI7	Handy Car Board	Business Tier

Chapter 3

Individual Steps and Test Description

In this chapter we provide a detailed description of the individual test cases to be executed on each pair of components that have to be integrated.

Each test case is identified with a code and is directly mapped with Table 2.1 for the integration between components and with Table 2.2 for the integration between subsystems. Test cases whose code starts with SI are integration tests between subsystems, while test cases whose code starts with I are integration tests between components.

For each method we are going to provide a brief description of the input values and the corresponding expected effects on the system.

3.1 Tier interactions

3.1.1 Integration test case SI1

Test Case Identifier	SI1T1
Test Item(s)	Business tier → Data Tier
Input Specification	Most typical calls (both correct and intentionally invalid ones) to the methods of the JPA Entities, which are mapped with tables in the Data tier.
Output Specification	Check if the Data tier does the correct queries on a test database. Also, it must react in the right way both if the requests are made correctly or wrongly and if they come from an unauthorized user.
Environmental Needs	Complete implementation of the Java Entity Beans; Java Persistence API; a test database; a driver that interacts with the Java Entity Beans.
Test Description	For each input, the response of the data tier will be compared with the expected output of the queries.
Testing Method	Automated with JUnit.

3.1.2 Integration test case SI2

Test Case Identifier	SI2T1
Test Item(s)	Mobile application → Business Tier
Input Specification	Typical REST API calls (both correct and intentionally invalid ones) from the mobile application to the business tier.
Output Specification	Check if the business tier responds accordingly to the API specification. Also, it must react in the right way both if the requests are made correctly or wrongly and if they come from an unauthorized user.
Environmental Needs	Complete implementation of the Business tier; a driver that simulates a mobile client through the REST API calls.
Test Description	For each API call of the clients, the response of the business tier will be compared with the expected output. The driver used for this test is a REST API client implemented in Java.
Testing Method	Automated with JUnit.

Test Case Identifier	SI2T2
Test Item(s)	Mobile application → Business Tier
Input Specification	Multiple concurrent (typical and correct) requests to the REST API of the business tier.
Output Specification	Check if the business tier answers the requests in a reasonable amount time with respect to the applied load.
Environmental Needs	Complete implementation of the Business tier; GlassFish Server; Apache JMeter.
Test Description	This test case evaluates whether the business tier satisfies the performance requirements stated in the RASD (section 3.3, *Performance requirements*).
Testing Method	Automated with Apache JMeter.

3.1.3 Integration test case SI3

Test Case Identifier	SI2T1
Test Item(s)	Power EnJoy Operator Terminal → Business Tier
Input Specification	Typical REST API calls (both correct and intentionally invalid ones) from the Power EnJoy Operator Terminal to the business tier.
Output Specification	Check if the business tier responds accordingly to the API specification. Also, it must react in the right way both if the requests are made correctly or wrongly and if they come from an unauthorized user.
Environmental Needs	Complete implementation of the Business tier; a driver that simulates a mobile client through the REST API calls.
Test Description	For each API call of the clients, the response of the business tier will be compared with the expected output. The driver used for this test is a REST API client implemented in Java.
Testing Method	Automated with JUnit.

3.1.4 Integration test case SI4

Test Case Identifier	SI4T1
Test Item(s)	On-board Tablet → Business Tier
Input Specification	Methods call from the on-board tablet to the application server.
Output Specification	Check if server can handle user's interrogations from the on-board tablet correctly.
Environmental Needs	GlassFish Server; complete implementation of the HandyCar System and the onboard tablet.
Test Description	Verify if the on-board tablet can communicate user's input to the server correctly and if the server can handle them in the right way.
Testing Method	Automated with JUnit.

3.2 External services

3.2.1 Integration test case SI5

Test Case Identifier	SI5T1
Test Item(s)	Business tier → Driving license validation service
Input Specification	Requests for the typical REST API calls (both correct and intentionally invalid ones) from the business tier to the external driving license validation service.
Output Specification	Check if the business tier interacts with the external service accordingly to the API specification.
Environmental Needs	Complete implementation of the Java Entity Beans; a driver that simulates the driving license validation service.
Test Description	For each request of the business tier, the interaction with the driving license validation service will be compared with the expected one.
Testing Method	Automated with JUnit.

3.2.2 Integration test case SI6

Test Case Identifier	SI6T1
Test Item(s)	Business tier → Payment service
Input Specification	Requests for the typical REST API calls (both correct and intentionally invalid ones) from the business tier to the external payment service.
Output Specification	Check if the business tier interacts with the external service accordingly to the API specification.
Environmental Needs	Complete implementation of the Java Entity Beans; a driver that simulates the payment service.
Test Description	For each request of the business tier, the interaction with the payment service will be compared with the expected one.
Testing Method	Automated with JUnit.

3.2.3 Integration test case SI7

Test Case Identifier	SI7T1
Test Item(s)	HandyCar Board → Business Tier
Input Specification	All the REST API calls (both correct and intentionally invalid ones) from the HandyCar Board to the business tier.
Output Specification	Check if the business tier responds accordingly to the API specification. Also, it must react in the right way even if the requests are made wrongly.
Environmental Needs	Complete implementation of the Business tier; a driver that simulates a HandyCar Board through the REST API calls.
Test Description	For each API call of the boards, the response of the business tier will be compared with the expected output. The driver used for this test is a REST API client implemented in Java.
Testing Method	Automated with JUnit.

Test Case Identifier	SI7T2
Test Item(s)	Business Tier → Handy Car Board
Input Specification	Multiple concurrent correct requests to the REST API of the business tier.
Output Specification	Check if the business tier answers the requests in a reasonable amount time with respect to the applied load.
Environmental Needs	Complete implementation of the Business tier; GlassFish Server; Apache JMeter.
Test Description	This test case evaluates whether the business tier satisfies the performance requirements stated in the RASD (section 3.3, *Performance requirements*).
Testing Method	Automated with Apache JMeter.

3.3 Entity Beans

In what follows, we are going to see the integration test cases from I01 to I06, that evaluate the integration between the Java Entity Beans and the Data tier. Only some part of the test cases differ from each other, so they are grouped together.

3.3.1 Integration test case I01-I06

Test Case Identifier	I01T1
Test Item(s)	User → DBMS
Input Specification	Typical queries on table User.
Test Case Identifier	I02T1
Test Item(s)	Ride → DBMS
Input Specification	Typical queries on table Ride.
Test Case Identifier	I03T1
Test Item(s)	Car → DBMS
Input Specification	Typical queries on table Car.
Test Case Identifier	I04T1
Test Item(s)	Operator → DBMS
Input Specification	Typical queries on table Operator.
Test Case Identifier	I05T1
Test Item(s)	SafeArea → DBMS
Input Specification	Typical queries on table SafeArea.
Test Case Identifier	I06T1
Test Item(s)	PowerGridStation → DBMS
Input Specification	Typical queries on table PowerGridStation.
Output Specification	Check if the queries return the expected result.
Environmental Needs	Complete implementation of the Java Entity Beans; GlassFish server; Java Persistence API; a test database; a driver that interacts with the Java Entity Beans.
Test Description	For each input, the response of the data tier will be compared with the expected output of the queries.
Testing Method	Automated with JUnit.

3.4 Ride Manager

3.4.1 Integration test case I7

Test Case Identifier	I7T1
Test Item(s)	Ride Manager → Car Manager, Car, Ride
Input Specification	Methods call from Ride Manager to Car Manager, to retrieve informations about the cars and rides.
Output Specification	Check if the cars and rides informations are correct and up-to-date.
Environmental Needs	GlassFish Server.
Test Description	Verify that the Ride Manager can access correctly to the car and ride informations.
Testing Method	Automated with JUnit.

Test Case Identifier	I7T2
Test Item(s)	Ride Manager → Fee Manager
Input Specification	Methods call from Ride Manager to Fee Manager, to add new unlocked fee variator or to calculate the final fee.
Output Specification	Check if the fee information are updated correctly.
Environmental Needs	GlassFish Server.
Test Description	Verify that the Ride Manager is able to communicate correctly the information about fee variators to the Fee Manager.
Testing Method	Automated with JUnit.

3.5 Car Manager

3.5.1 Integration test case I8

Test Case Identifier	I8T1
Test Item(s)	Car Manager → Ride Manager, Car
Input Specification	Methods call from Car Manager to Ride Manager, to update the information of the rides (in particular to add a new ride) and cars.
Output Specification	Check if the rides and cars information are updated correctly.
Environmental Needs	GlassFish Server.
Test Description	Verify that the Car Manager is able to communicate correctly the information about the new rides to the Ride Manager.
Testing Method	Automated with JUnit.

Test Case Identifier	I8T2
Test Item(s)	Car Manager → User Manager
Input Specification	Methods call from Car Manager to User Manager, to report that a ride it's over and so that the user has to pay the fee.
Output Specification	Check if the ride information are reported correctly.
Environmental Needs	GlassFish Server.
Test Description	Verify that the Car Manager is able to communicate correctly the information about the new rides to the User Manager.
Testing Method	Automated with JUnit.

3.6 User Manager

3.6.1 Integration test case I9

Test Case Identifier	I9T1
Test Item(s)	User Manager → User SafeArea Manager
Input Specification	Methods call from User Manager to SafeArea Manager, to start a new query on the SafeArea table.
Output Specification	Check if the right query starts and if it is executed correctly.
Environmental Needs	Complete implementation of the Java Entity Beans; GlassFish Server; Java Persistence API; a test database.
Test Description	Verify if the User Manager can communicate the right parameters in order to start the query and that the right query starts.
Testing Method	Automated with JUnit.

Test Case Identifier	I9T2
Test Item(s)	User Manager → User,Email Sender
Input Specification	Methods call from User Manager to the Email Sender in order to guarantee a right email authentication process.
Output Specification	The email authentication process must be correctly handled.
Environmental Needs	GlassFish Server;e-mail sender and receiver
Test Description	Ensure that a user can properly verify his/her email address in order to start using the system functionalities. In order to do that, a mock email address manager which simulates the user behaviour is needed.
Testing Method	Automated with JUnit.

3.7 User Safe Area Manager

3.7.1 Integration test case I10

Test Case Identifier	I10T1
Test Item(s)	User Safe Area Manager → User, Safe Area, Power grid Station
Input Specification	Methods call from User Safe Area Manager, to start a new query on the Safe Area or Power Grid Station or User table.
Output Specification	Check if the right query starts and if it is executed correctly.
Environmental Needs	GlassFish Server
Test Description	Verify if the User Safe Area Manager can communicate the right parameters in order to start the query and that the right query starts, providing information about Power Grid Stations and Safe Areas.
Testing Method	Automated with JUnit.

3.8 Operator Safe Area Manager

3.8.1 Integration test case I11

Test Case Identifier	I11T1
Test Item(s)	Operator Safe Area Manager → Operator, Safe Area, Power grid Station
Input Specification	Methods call from Operator Safe Area Manager, to start a new query on the Safe Area or Power Grid Station or Operator table.
Output Specification	Check if the right query starts and if it is executed correctly.
Environmental Needs	GlassFish Server
Test Description	Verify if the Operator Safe Area Manager can communicate the right parameters in order to start the query and that the right query starts, that allows the operator to add or remove Power Grid Stations and Safe Areas.
Testing Method	Automated with JUnit.

3.9 Fee Manager

3.9.1 Integration test case I12

Test Case Identifier	I12T1
Test Item(s)	Fee Manager → Ride
Input Specification	Methods call from Fee Manager, to manage all the informations about the fee retrieved form a specific ride
Output Specification	The ride information must be correct and up to date.
Environmental Needs	GlassFish Server
Test Description	Verify that the fee is updated correctly on a specific ride with or without variation and also when the money saving option is enabled .
Testing Method	Automated with JUnit.

3.10 Payment Manager

3.10.1 Integration test case I13

Test Case Identifier	I13T1
Test Item(s)	Payment Manager → User
Input Specification	Methods call from Payment Manager, to handle the payment of the fee that the user has to pay
Output Specification	The payment informations must be correct and up to date.
Environmental Needs	GlassFish Server
Test Description	Verify that the payment process is handled and completed correctly.
Testing Method	Automated with JUnit.

3.11 Container

In what follows, we are going to see the integration test cases from I14 to I19, that evaluate the SessionBean assignment from the relative container. The test are very similar one another, so they are grouped together.

3.11.1 Integration test case I14-I19

Test Case Identifier	I14T1
Test Item(s)	User Manager Container → User Manager
Test Case Identifier	I15T1
Test Item(s)	Car Manager Container → Car Manager
Test Case Identifier	I16T1
Test Item(s)	Ride Manager Container → Ride Manager
Test Case Identifier	I17T1
Test Item(s)	Fee Manager Container → Fee Manager
Test Case Identifier	I18T1
Test Item(s)	SafeArea Manager Container → SafeArea Manager
Test Case Identifier	I19T1
Test Item(s)	Payment Manager Container → Payment Manager
Input Specification	Request the relative SessionBean.
Output Specification	The requested SessionBean must be assigned correctly and the concurrency between requests must be managed in the right way.
Environmental Needs	GlassFish Server.
Test Description	Multiple request for the same SessionBean have to be made in the same time, in order to check the right concurrency management.
Testing Method	Automated with JUnit and Arquillan.

3.12 Controller

Test Case Identifier	I20T1
Test Item(s)	Controller → Payment Manager Container, Fee Manager Container, Car Manager Container, Ride Manager Container, User Manager Container, Safe Area Container
Input Specification	Requests from Controller to the containers for the functionalities offered by Session Beans within containers.
Output Specification	The controller has to be able to provide the right functionality carrying out the proper request to the containers.
Environmental Needs	GlassFish Server.
Test Description	Ensure that the controller is able to provide the functionalities of the system offered by the containers.
Testing Method	Automated with JUnit and Arquillian.

Chapter 4

Tools and Test Equipment Required

4.1 Tools

The software tools required for the integration testing are the following:

JUnit[1] a simple framework to write repeatable tests and it is an instance of the xUnit architecture for unit testing frameworks. We are going to use it both for unit testing activities of the single components and for the integration testing, where we verify that the interactions between components are producing the expected results.

Apache JMeter[2] a powerful tool designed to load test functional behaviour and measure performance. In particular, it is used to simulate load on the business tier in order to check if the latter satisfies the performance requirements stated in the RASD.

Arquillan[3] Arquillian is an innovative and highly extensible testing platform for the JVM that enables developers to easily create automated integration, functional and acceptance tests for Java middleware. We plan to use it mainly to test the containers and their integration with the Java Beans.

4.2 Test Equipment

In order to test the various components of the system, we require at least this test equipment:

- An android smartphone for each of the following screen sizes: 4.3", 4.7", 5", 5.5", 6". The screen sizes can have a tolerance of 0.1".
- An iphone for each model from 4S to 7.
- A server where we can simulate the business and the data tier.
- Two electric cars with HandyCar board installed.

Chapter 5

Program Stubs and Test Data Required

5.1 Drivers

As already said in the section 2.3 our testing approach is bottom-up. In order to apply this approach, it is necessary to develop some drivers. These drivers will generate the necessary inputs to manage the test in the right way. We deem necessary to develop two different drivers: the user diver and the ride one.

5.1.1 User Driver

This driver will be used for all those tests that demands an interaction with the user, such as a testing the login or the start of a ride. The user driver will provide all the necessary inputs, taken from a predefined pool.

5.1.2 Ride Driver

This driver will simulate an ongoing ride, providing all the data of a ride such as the starting time or which car has been rented. Its task is to call all the functions useful during a ride, such as those relative to the fee or the car status. Like the user driver, the ride driver will take all the data from a predefined pool.

5.2 Stubs

Despite the bottom-up approach we think it is important to develop a stub: the car stub.

5.2.1 Car Stub

This stub will mimic a car of the PowerEnjoy service. During testing, when the system will need a piece of data of a car (such as car status or the battery level) this stub will be called. In this way, we will speed up the first part of the testing phase, making the testing of the system independent from the state of the Handycar System and onboard tablet implementation.

5.3 Mocked Services

Our system must talk with some external services. In order to proceed with tests, we have to mock them. The system to mock will be:

- Payment Service;
- External service for Driving License validation.

5.4 Database

To test queries to the database we need to fill it with a reduced set of data of all entities described in the Design Document (see section 2.2.1 of the Design Document for more details).

Appendix A

Appendix

A.1 Used software and tools

- LaTeX [1], for typesetting this document.
- Texmaker[2], for the writing of this document.
- GitHub[3] for version control and distributed work.
- GitHub desktop[4] used to collaborate in the team and to keep track of the changes.

A.2 Work hours

The statistics about commits and code contribution are available on the GitHub repository of the project[5].

These are our estimation of the work hours spent on this project:

- Marco Ieni: 12 hours
- Francesco Lamonaca: 12 hours
- Marco Miglionico: 12 hours

[1] http://https://www.latex-project.org/
[2] http://www.xm1math.net/texmaker/
[3] https://github.com/marcomiglionico94/Software-Engineering-2-Project
[4] https://desktop.github.com/
[5] https://github.com/marcomiglionico94/Software-Engineering-2-Project