**I**ntegration **T**est **P**lan **D**ocument



Version 1.0

Luca Santini 808710

Riccardo Remigio 874939

Index

[1. Introduction 3](#_Toc472203147)

[1.2 Purpose and Scope 3](#_Toc472203148)

[1.3 List of definitions and abbreviations 3](#_Toc472203149)

[1.4 List of reference documents 3](#_Toc472203150)

[2. Integration strategy 4](#_Toc472203151)

[2.1 Entry criteria 4](#_Toc472203152)

[2.2 Elements to be integrated 5](#_Toc472203153)

[2.3 Integration testing strategy 5](#_Toc472203154)

[2.4 Sequence of component/Function integration 6](#_Toc472203155)

[2.4.1 Software integration sequence 6](#_Toc472203156)

[3. Individual Steps and Test Description 10](#_Toc472203157)

# Introduction

## 1.2 Purpose and Scope

This document specifies all the necessary information about the integration test plan for PowerEnJoy.

The purpose of integration testing is to verify functional and non functional requirements placed on RASD and Design Document, by testing the interaction between the different modules composing the whole software.

## 1.3 List of definitions and abbreviations

RASD: Requirements and Specifications Document

DD: Design Document

API: Application Programming Interface

DBMS: Database Management System

## 1.4 List of reference documents

* Assignments AA 2016-2017.pdf
* Verification and validation, part I.pdf
* Verification and validation, part II.pdf
* PowerEnJoy RASD
* PowerEnJoy DD
* Integration testing example document.pdf

# Integration strategy

## 2.1 Entry criteria

Before beginning the integration test phase, there are some other things that have to be necessarily completed.

For example, they must have been issued the RASD and the Design Document, and must already be written part of the code that covers the components that should be integrated.

When will begin integration testing there will still be some incomplete components that will be necessarily completed before finishing the integration testing.

Before starting the integration test must have almost completed all the unit tests, which will continue during the integration test.

We do not write the precise percentage, but we can say that, using a bottom-up approach, the first components to be tested are: Data Manager and Payment Manager.

Then follow Profile Manager and Ride Manager, then Vehicle manager.

Finally, Map Manager and Reservation Manager, and for last Request Manager that is called only by the web browser.

To be able to proceed smoothly so we need to follow this pattern during the writing of code, so completing first the components to be tested first.

In the following chapters will be clearer integration methods and the criteria we will use

## 2.2 Elements to be integrated

In this chapter we list all the components that have to be integrated in the testing phase, following the architecture written in the design document.

Based on the component view of the design document, it’s possible to identify the interaction between the components of the application server, which is the main high level component of our architecture.

These components are:

-Client request manager

-Reserve manager

-Profile manager

-Payment handler

-Data manager

-Car handler

-Position handler

We will test also the integration between the components above and the external interfaces given by outer services.

## 2.3 Integration testing strategy

We decided to use a bottom-up strategy to choose the order of the components to be integrated. This decision was taken as it is a simple solution to effectively applicable to our system that is formed by few and small components.

Using this strategy, we will not need to use stubs but we need to use Drivers for testing the components.

## 2.4 Sequence of component/Function integration

This section is dedicated to the detailed description of the order of integration of the components that had been previously mentioned. Using the bottom-up approach we will use the drivers to make calls from the higher levels even when these are not yet integrated.

### 2.4.1 Software integration sequence

To decide the order of the components to be tested first, we rely on the component view of the design document, respecting the bottom-up method.

In this way we follow the possible interaction between the components as shown in the diagram quoted above.

First of all, we integrate the components that are used by multiple components, and which interact with external services.

These components are:

**-Data manager**, which permits the access to data to the other components.

**-Payment manger**, which manage all the operation involved with payment



In the second step we develop other driver that will be used to test:

**-Profile manager**

**-Ride manager**



In the next step we develop another driver to test:

**-Vehicle Manager**



In the next step we integrate, using two new drivers,

**-Map Manager**

**-Reservation Manager**

The last one is the client request manager, because it is called only by the browser of the client through an interface.

We can now delete all the drivers and we have the whole system integrated.

# Individual Steps and Test Description

In this chapter we describe the test that must be executed on each component, so to arrive at the integration level shown in the chapter before. We list some methods involved in the interactions between components and for each of them we describe the effect for some critical inputs. Furthermore we specify the components that should call the method but that are implemented by a driver and that will be integrate in the next steps.

Driver -> Data Manager:

|  |  |
| --- | --- |
| checkPassword(email,psw) | |
| **Input** | Effect |
| An email in the database with the corresponding password | Returns the object user, corresponding to this email |
| An email in the database with an incorrect password | An InvalidPasswordException is raised |
| An email not in the database | An InvalidEmailException is raised |
| A null parameter | An NullPointerException is raised |

|  |  |
| --- | --- |
| readVehicleInfo() | |
| **Input** | Effect |
| - | If there are vehicles in the database this method return a list with all the vehicles, if the database is empty an emptyDatabaseException is raised |

|  |  |
| --- | --- |
| newUser(form) | |
| **Input** | Effect |
| Null parameter | NullArgumentException is raised |
| Invalid data in the form | Returns false |
| Correct data | The system creates a password, then sends it by email to the user and the method returns true |
| Incomplete data in the form | Returns false |

|  |  |
| --- | --- |
| findVehicleInfo(vehicle) | |
| **Input** | Effect |
| Null parameter | NullArgumentException is raised |
| A vehicle with an id in the database | Returns the vehicle in the database |
| A vehicle with an id not in the database | Returns the vehicle in the database |

|  |  |
| --- | --- |
| createReservation(user,vehicle) | |
| **Input** | Effect |
| A user and a vehicle in the database | Creates a new reservation object in the database and returns true |
| A user not in the database | IllegalArguementException is raised |
| A vehicle not in the database | IllegalArguementException is raised |
| A null parameter | NullPointerException is raised |

|  |  |
| --- | --- |
| readReservation(user) | |
| **Input** | Effect |
| A user in the database | Returns the reservation associated to the user |
| A user not in the database | IllegalArguementException is raised |
| A user in the database without a reservation | IllegalArguementException is raised |
| A null parameter | NullPointerException is raised |

|  |  |
| --- | --- |
| newRide(ride) | |
| **Input** | Effect |
| A ride with valid attributes | Creates a new ride object in the database with the correct attributes |
| A ride with wrong attributes | IllegalArguementException is raised |
| A null parameter | NullPointerException is raised |

|  |  |
| --- | --- |
| updatePosition(vehicle,position) | |
| **Input** | Effect |
| A vehicle in the database and a valid position | Edits the vehicle position in the database |
| A vehicle not in the database | IllegalArguementException is raised |
| A null parameter | NullPointerException is raised |
| An invalid position | IllegalArguementException is raised |

|  |  |
| --- | --- |
| findPaymentInformation(user) | |
| **Input** | Effect |
| A user in the database | Returns an object paymentInfo associated with the user |
| A user not in database | IllegalArguementException is raised |
| A null parameter | NullPointerException is raised |

|  |  |
| --- | --- |
| findAllChargingStation() | |
| **Input** | Effect |
| - | Returns the list of the charging station in the database |

|  |  |
| --- | --- |
| updateState(state,vehicle) | |
| **Input** | Effect |
| A valid state and a vehicle in the database | Edits the vehicle state stored in the database |
| A null parameter | NullPointerException is raised |
| An invalid state | IllegalArguementException is raised |
| A vehicle not in database | IllegalArguementException is raised |

Driver -> Payment Manager

|  |  |
| --- | --- |
| **chargeUser(accreditedMoney, paymentInfo)** | |
| **Input** | **Effect** |
| Null parameters | NullArgumentException is raised |
| Negative value of accreditedMoney | IllegalArgumentException is raised |
| User hasn’t enough money in his payment system | The system must put the user in the blacklist and the method returns false |
| Wrong paymentInfo | IllegalArgumentException is raised |
| User has enough money and correct paymentInfo | Return true |

Driver -> Profile Manager

|  |  |
| --- | --- |
| **profileAccess(email, psw)** | |
| **Input** | **Effect** |
| Null parameters | NullArgumentException is raised |
| An email in the database with the corresponding password | Returns the object user, corresponding to this email |
| An email in the database with an incorrect password | An invalidPasswordException is raised |
| An email not in the database | An invalidEmailException is raised |

|  |  |
| --- | --- |
| **createProfile(form)** | |
| **Input** | **Effect** |
| Null parameters | NullArgumentException is raised |
| Invalid data in the form | Return false |
| Correct data | The system creates a password, sends it by email to the user and the method returns true |
| Incomplete data in the form | Return false |

|  |  |
| --- | --- |
| **changeVehicleState(state, vehicle)** | |
| **Input** | **Effect** |
| Null parameters | NullArgumentException is raised |
| Invalid state | IllegalArgumentException is raised |
| The vehicle does not exist | NullPointerException is raised |
| Correct parameters | The vehicle in the parameters must change the state in the database and the method returns true |
| The vehicle is in used | Return false |

Driver -> Ride Manager

|  |  |
| --- | --- |
| **createRide(user, vehicle)** | |
| **Input** | **Effect** |
| Null parameters | NullArgumentException is raised |
| Invalid user | IllegalArgumentException is raised |
| Invalid vehicle | IllegalArgumentException is raised |
| Correct parameters | The system creates a new ride in the database, the vehicle changes its state in “in use” and the method returns true |
| The vehicle is in a state that is not “reserved” | IllegalArgumentException is raised |

|  |  |
| --- | --- |
| **endRide(vehicle)** | |
| **Input** | **Effect** |
| Null parameters | NullArgumentException is raised |
| The vehicle does not exist | IllegalArgumentException is raised |
| The vehicle’s state is not “in use” | IllegalStateException |
| Correct parameter | The system changes the state of the vehicle, locks the car, updates his position in the database and returns true |

|  |  |
| --- | --- |
| **calculateFee(ride)** | |
| **Input** | **Effect** |
| Null parameter | NullArgumentException is raised |
| The ride does not exist | IllegalArgumentException is raised |
| Ride with some bonuses or ride without bonuses | The methods must calculate the correct amount of money to charge to the user |

Driver -> Vehicle Manager