

Luca Scannapieco - 877145

Andrea Pasquali - 808733

Emanuele Torelli - 876210

15/01/2017

PowerEnJoy

Integration Test Plan Document

Table of contents

[1. Introduction 2](#_Toc471938883)

[1.1. Purpose and scope 2](#_Toc471938884)

[1.2. List of definitions and abbreviations 2](#_Toc471938885)

[2. Integration strategies 2](#_Toc471938886)

[2.1. Entry criteria 2](#_Toc471938887)

[2.2. Elements to be integrated 2](#_Toc471938888)

[2.3. Integration test strategies 3](#_Toc471938889)

[2.4. Sequence of component/function integration 4](#_Toc471938890)

[2.4.1. Software integration sequence 4](#_Toc471938891)

[2.4.2. Subsystem integration sequence 7](#_Toc471938892)

[3. Individual steps and test description 7](#_Toc471938893)

[3.1. ViewForClient, LoginController 7](#_Toc471938894)

[3.2. ViewForClient, SignUpController 7](#_Toc471938895)

[3.3. ReservationController, CarController 8](#_Toc471938896)

[3.4. Client app, ViewForClient 8](#_Toc471938897)

[3.5. ViewForClient, ReservationController 8](#_Toc471938898)

[3.6. ViewForClient, CarController 9](#_Toc471938899)

[3.7. CarController, DBMS 9](#_Toc471938900)

[3.8. ReservationController, DBMS 9](#_Toc471938901)

[3.9. Car app, RideController 10](#_Toc471938902)

[3.10. Car app, ViewForCar 11](#_Toc471938903)

[3.11. ViewForCar, RideController 11](#_Toc471938904)

[3.12. RideController, CityController 11](#_Toc471938905)

[3.13. CityController, DBMS 11](#_Toc471938906)

[3.14. RideController, DBMS 11](#_Toc471938907)

[4. Tools and test equipment required 12](#_Toc471938908)

[5. Program stubs and test data required 12](#_Toc471938909)

[6. Other info 12](#_Toc471938910)

[6.1. Sample documents 12](#_Toc471938911)

[6.2. Used tools 12](#_Toc471938912)

[6.3. Hours of work 13](#_Toc471938913)

[6.4. Changelog 13](#_Toc471938914)

# 1. Introduction

## 1.1. Purpose and scope

This document represents the Integration Testing Plan Document for PowerEnjoy. Integration testing is a key activity to guarantee that all the diﬀerent subsystems composing PowerEnjoy interoperate consistently with the requirements they are supposed to fulﬁl and without exhibiting unexpected behaviours. The purpose of this document is to outline, in a clear and comprehensive way, the main aspects concerning the organization of the integration testing activity for all the components that make up the system. In the following sections we’re going to provide:

* The criteria that must be met by the project status before the integration testing of the outlined elements can begin.
* A list of the subsystems and their components involved in the integration activity that will have to be tested.
* A description of the integration testing approach and the rationale behind it.
* The sequence in which components and subsystems will be integrated.
* A description of the planned testing activities for each integration step, including their input data and the expected output.
* 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.
* The stubs needed for the integration of the components and subsystems and the for the testing of the system.

## 1.2. List of definitions and abbreviations

* Component: the software level units which exploit every functionality of a subsystem (e.g. in our case a component is a controller or a view).
* Subsystem: a high-level functional element of the system (e.g. the car app, the database or the server).
* RASD: the Requirement Analysis and Specification Document provided before.
* DD: the Design Document provided before.
* ITPD: this Integration Test Plan Document.
* DBMS: database management system.
* MVC: model-view-controller, is a software design pattern for implementing user interfaces on a system.

# 2. Integration strategies

## 2.1. Entry criteria

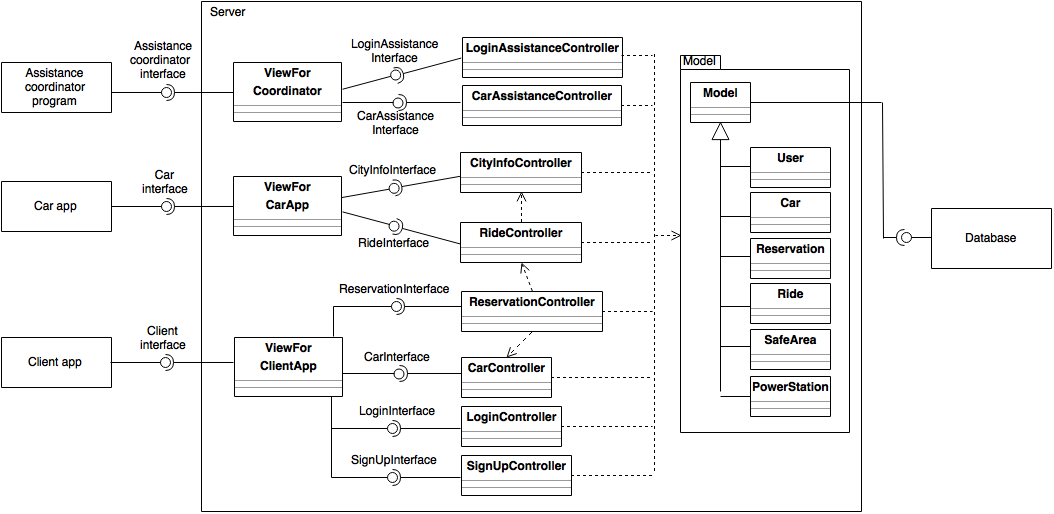
At each step of the software integration system described in the section 2.4.1 the following criteria must be met: all the functions of the components having outgoing arcs (only considering the directional arcs) must have been unit tested.

## 2.2. Elements to be integrated

As explained in the DD our system is built of five high-level components: car app, client app and assistance coordinator program for the client side; database and server for the server side.  
The server subsystem is obviously the most complex and it is, in turn, built of other components interacting among them and forming the MVC pattern. In particular the components to be integrated in the server subsystem are:

* The model reflecting the data in the database.
* The controllers that are CityInfoController, ReservationController, CarController, RideController, LoginController, SignUpController, LoginAssistanceController, CarAssistanceController.
* The three views (one for each kind of client).

For furtherly clarify the reasoning of the next two sections, we report our component diagram below (for a further clarification, see DD chapter 2.2.).



## 2.3. Integration test strategies

We are going to use an incremental approach for integration testing. In particular, we will adopt essentially a bottom-up strategy with few slight modifications.

We will use the a purely bottom-up approach in order to build the component called “server” in the high-level component diagram, that in essence represent the business layer of our application. Therefore, we will start integrating together the atomic subsystems of the server, i.e. the lower level components that do not depend on other components; then we will incrementally integrate the other subsystems that only depends on already integrated and tested components. This strategy, based on the hierarchical structure of the system, allows us to perform the integration test following the development process: as soon as components are released, we integrate them and test the integration. Furthermore, using bottom-up strategy for the server we reduce the overhead time needed to build stubs.  
In order to choose what to integrate among the atomic components we will follow the critical-module-first policy. In our case the most critical modules are the most used-one, such as the model that is the core of our MVC in the server side and therefore also the first component to be developed.   
For what concerns the client side, we can say that we violate a bit the bottom-up strategy rules. In fact we are going to test the client side components such as car app, client app and assistance coordinator program together with the server components even if the client side components use those of the server. This little modification of the strategy has the purpose of increase the parallelism of the work and consequently even the efficiency.

## 2.4. Sequence of component/function integration

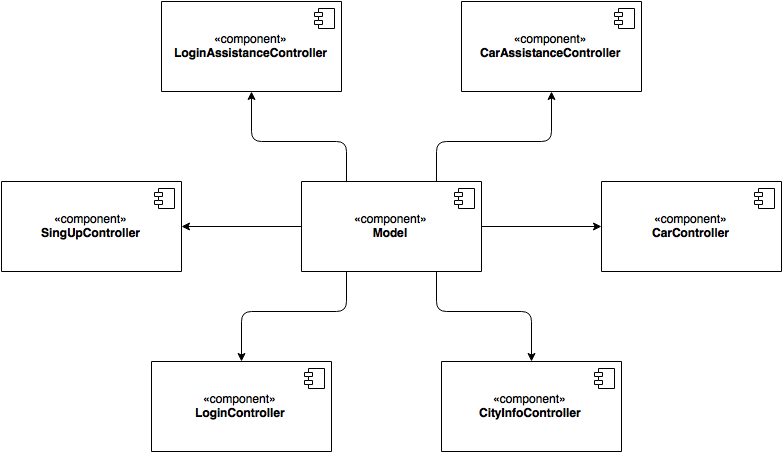
In this section we are going to describe the order of integration of the components and subsystems of PowerEnjoy. An arc going from component A to component B means that component A needs to be implemented before component B; a unidirectional arc means that there is not such a dependency.

### 2.4.1. Software integration sequence

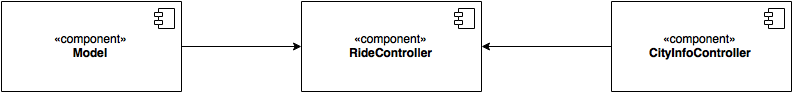
According with the critical-module-first policy described in the chapter 2.3 the first two elements to be integrated are the database and the model because they refer to the data of our system and thus they are the most used components.



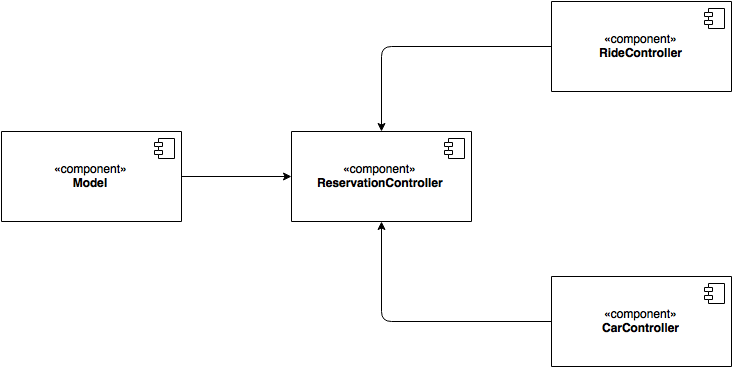
The next components to be integrated are the controllers that do not interact with other controllers such as: SignUpController, LoginController, LoginAssistanceController, CarAssistanceController, CityInfoController, CarController.



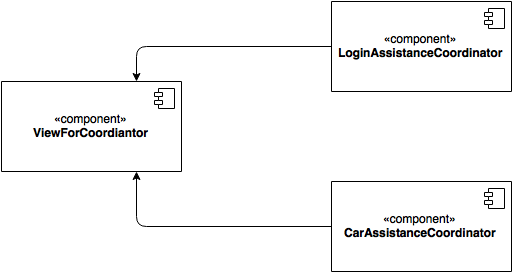
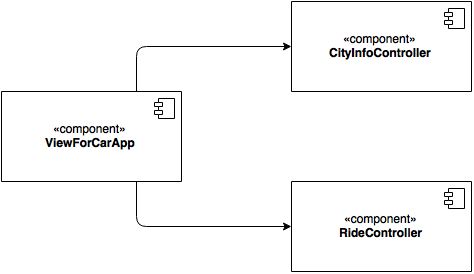
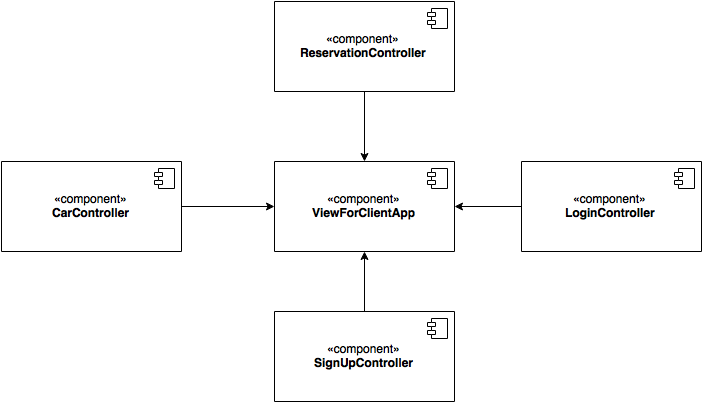
Now we can proceed adding the controllers interacting with the already implemented controllers: the next controller is RideController that only interact with CityInfoController.



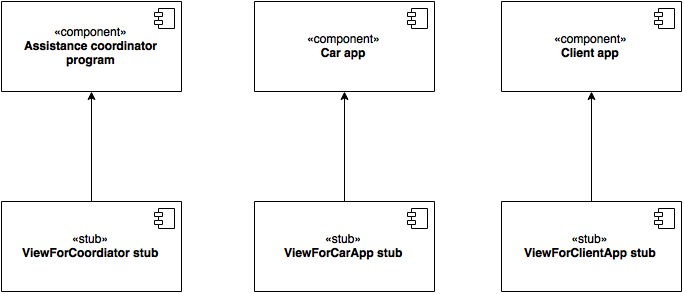
The last controller to integrate is ReservationController that uses RideController and CarController.



Once we have integrated all the controllers we can finally integrate the three views.

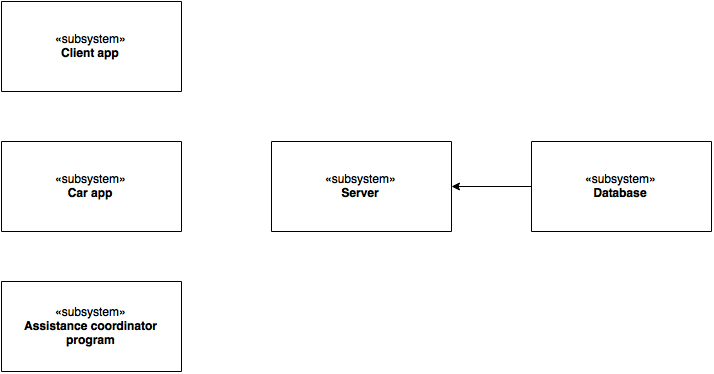


As explained in the section 2.3. we can integrate the three components of the client side in parallel with those of the server side. We only need to implement the stubs for the three views used by the three client components. These stubs will be substituted by the real views once the server subsystem will be completely integrated.



### 2.4.2. Subsystem integration sequence

The following schema shows how the integration test proceeds looking at the high-level components (or subsystems). Note that if two subsystems are not connected with any arcs it means that they can be integrated in parallel.



# 3. Individual steps and test description

## 3.1. ViewForClient, LoginController

|  |  |
| --- | --- |
| Login(email, password) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A non-registered Email | An InvalidArgumentValueException is raised |
| The parameters doesn’t correspond each other | Return false |
| The combination is valid | Return true |

## 3.2. ViewForClient, SignUpController

|  |  |
| --- | --- |
| signUp(name, surname, phoneNumber, email, address, SSN, creditCard, licenceNumber) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A non-existing Email | An InvalidArgumentValueException is raised |
| A non-existing SSN | An InvalidArgumentValueException is raised |
| A non-existing credit Card | An InvalidArgumentValueException is raised |
| A non-existing licence number | An InvalidArgumentValueException is raised |
| An already-registered email | An InvalidArgumentValueException is raised |
| An already-registered SSN | An InvalidArgumentValueException is raised |
| An already-registered credit card | An InvalidArgumentValueException is raised |
| An already-registered licence number | An InvalidArgumentValueException is raised |
| Valid credentials | Return True |

## 3.3. ReservationController, CarController

|  |  |
| --- | --- |
| CheckAvailability(car) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A car with a non-existing id | An InvalidArgumentValueException is raised |
| A valid set of parameters | Returns true or false depends on the car availability |

|  |  |
| --- | --- |
| askPosition(Car) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A car with a non-existing id | An InvalidArgumentValueException is raised |
| A valid set of parameters | Returns the position of the car |

## 3.4. Client app, ViewForClient

|  |  |
| --- | --- |
| reservationRequest(car, user) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A car with a non-existing id | An InvalidArgumentValueException is raised |
| A not-valid user | An InvalidArgumentValueException is raised |
| A not available car | Returns false |
| A valid set of parameters | Returns true |

|  |  |
| --- | --- |
| availableCarRequest() | |
| *Input* | *Effect* |
| Nothing | Returns the set of available cars |

|  |  |
| --- | --- |
| unlockRequest(position, user) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A not-valid user | An InvalidArgumentValueException is raised |
| A valid set of parameters | Returns true if the position is correct, false otherwise |

## 3.5. ViewForClient, ReservationController

|  |  |
| --- | --- |
| reserve(car, user) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A car with a non-existing id | An InvalidArgumentValueException is raised |
| A not-valid user | An InvalidArgumentValueException is raised |
| A valid set of parameters | Returns true if is possible to reserve the car, false otherwise |

|  |  |
| --- | --- |
| unlock(position, user) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A not-valid user | An InvalidArgumentValueException is raised |
| A valid set of parameters | Returns true if the user position is near to the car that is reserved by the user, false otherwise |

## 3.6. ViewForClient, CarController

|  |  |
| --- | --- |
| availableCar() | |
| *Input* | *Effect* |
| nothing | Returns the set of available car of the database |

## 3.7. CarController, DBMS

|  |  |
| --- | --- |
| getAllCars() | |
| *Input* | *Effect* |
| nothing | Returns the set of all available and not available cars |

|  |  |
| --- | --- |
| getPosition(car) (car) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A car with a non-valid Id | An InvalidArgumentValueException is raised |
| A valid parameter | Returns the position of the selected car |

## 3.8. ReservationController, DBMS

|  |  |
| --- | --- |
| getReservation(user) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A not-existing user | An InvalidArgumentValueException is raised |
| A valid user who hasn’t a not expired reservation | Returns error |
| A valid user who has a not expired reservation | Returns the not-expired reservation of that user |

|  |  |
| --- | --- |
| setCarAvailable(car) (car) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A car with a not valid Id | An InvalidArgumentValueException is raised |
| A car already available | An InvalidArgumentValueException is raised |
| A valid parameter | Set the attribute “available” to True |

|  |  |
| --- | --- |
| getCarAvailability(car) (car) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A car with a not valid Id | An InvalidArgumentValueException is raised |
| A valid parameter | Returns the Boolean value of the attribute “available” of the car |

|  |  |
| --- | --- |
| createReservation() | |
| *Input* | *Effect* |
| Nothing | Returns a new Reservation created in the DB, setting all parameters to NULL and the attribute “expired”= false |

|  |  |
| --- | --- |
| setCar(reservation,car) (Reservation) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A reservation with the attribute “car” not NULL | An InvalidArgumentValueException is raised |
| A car with a not valid Id | An InvalidArgumentValueException is raised |
| An available car | An InvalidArgumentValueException is raised |
| A reservation with the attribute “expired” =true | An InvalidArgumentValueException is raised |
| A valid set of parameters | Sets the attribute “car” of the reservation |

|  |  |
| --- | --- |
| getCar(reservation) (Reservation) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A reservation with the attribute “car” = NULL | An InvalidArgumentValueException is raised |
| A valid parameter | Returns the attribute “car” of the reservation |

|  |  |
| --- | --- |
| setReservationUser(reservation, user) (Reservation) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A non-registered user | An InvalidArgumentValueException is raised |
| A reservation expired | An InvalidArgumentValueException is raised |
| A valid parameter | Sets the attribute “user” of the reservation |

|  |  |
| --- | --- |
| setReservationExpired(reservation) (Reservation) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A reservation with attribute “expired” = true | An InvalidArgumentValueException is raised |
| A valid parameter | Sets the attribute “expired” to true |

## 3.9. Car app, RideController

|  |  |
| --- | --- |
| startRide(reservation) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A reservation already expired | An InvalidArgumentValueException is raised |
| A valid reservation | Initializes a new Ride and returns it |

## 3.10. Car app, ViewForCar

|  |  |
| --- | --- |
| endRideRequest(position, car) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A car with a non-existing Id | An InvalidArgumentValueException is raised |
| A valid set of parameters | The request is dispached to the Ride Controller |

## 3.11. ViewForCar, RideController

|  |  |
| --- | --- |
| endRide(position, car) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A car with a non-existing Id | An InvalidArgumentValueException is raised |
| A car without a running ride | An InvalidArgumentValueException is raised |
| A not-safe position | Returns error |
| A valid set of parameters | Ride Controller set the running ride to true. |

## 3.12. RideController, CityController

|  |  |
| --- | --- |
| checkPosition(position) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A valid parameter | City controller checks if the position corresponds to a safe area or not |

## 3.13. CityController, DBMS

|  |  |
| --- | --- |
| getSafeArea(position) (risponde true or false a seconda se position è in safe area o no) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A valid parameter | Returns true if the position is in a safe area, false otherwise |

## 3.14. RideController, DBMS

|  |  |
| --- | --- |
| createRide() | |
| *Input* | *Effect* |
| nothing | Return a new Ride initialized with all default parameters |

|  |  |
| --- | --- |
| getRunningRide(car) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A car with a non-valid Id | An InvalidArgumentValueException is raised |
| A valid parameter of a car that hasn’t any running ride | Returns false |
| A valid parameter of a car that has more than one running ride | Returns false |
| A valid parameter of a car that has only one running ride | Returns true |

|  |  |
| --- | --- |
| setTerminated(ride) (ride) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A ride that is already terminated | An InvalidArgumentValueException is raised |
| A ride not terminated | Set the attribute “terminated” of the ride as True |

|  |  |
| --- | --- |
| setReservation(ride,reservation) (ride) | |
| *Input* | *Effect* |
| A null parameter | A NullArgumentException is raised |
| A reservation not expired | An InvalidArgumentValueException is raised |
| A terminated Ride | An InvalidArgumentValueException is raised |
| A valid set of parameters | Set the attribute “reservation” |

3.15. RideController, Notification Helper

# 4. Tools and test equipment required

# 5. Program stubs and test data required

As explained in the chapter 2.3, we will adopt a bottom up strategy for integration test of the server side. This strategy requires the implementation of several drivers in order to invoke properly the methods of the components to be tested. The necessary drivers are:

* Model Driver
* SignUp Driver
* Login Driver
* LoginAssistance Driver
* CarAssistance Driver
* CityInfo Driver
* Car Driver
* Ride Driver
* Reservation Driver

Furthermore, we have chosen to use a sort of top-down strategy for the integration of the components of the client side. These components simply invoke the methods offered by the three views of the server; hence we need the implement the stubs for the three views.

# 6. Other info

## 6.1. Sample documents

* Assignments AA 2016-2017.pdf
* Documents previously provided:
  + PowerEnJoy – RASD.pdf
  + PowerEnJoy – DD.pdf
* Sample documents:
  + Integration testing example document.pdf
  + Sample Integration Test Plan Document.pdf
* Course slides:
  + Verification and validation, part I.pdf
  + Verification and validation, part I.pdf
  + VerificationTools.pdf

## 6.2. Used tools

* Microsoft Word 2016, for the drafting of the ITPD
* Microsoft OneDrive, to allow concurrent editing
* GitHub, to store the project in a repo
* Draw.io, for the drawing of the diagrams

## 6.3. Hours of work

For redacting and writing the Integration Test Plan Document we spent approximately 20 hours per person.

## 6.4. Changelog

No changes in the document for the moment.