

PROJECT for

SOFTWARE ENGINEERING 2

**Testing Document**

Politecnico di Milano

A.A. 2016-2017

Prof.ssa Elisabetta Di Nitto

Students:

Diego Gaboardi

Giorgio Giardini

Riccardo Giol

INDEX

* 1 Introduction
  + 1.1 Revision History
  + 1.2 Purpose and Scope
  + 1.3 Definition, Acronyms, Abbreviations
  + 1.4 Reference documents
* 2 Integration Strategy
  + 2.1 Entry criteria
  + 2.2 Elements to be integrated
  + 2.3 Integration testing strategy
  + 2.4 Sequence of component / Function integration
    - 2.4.1 Data Controller
    - 2.4.2 Client Management System
    - 2.4.3 Car Management System
    - 2.4.4 Course Management System
    - 2.4.5 Reservation Management System
    - 2.4.6 Assistant Management System
  + 2.5 Subsystem integration sequence
* 3 Individual step – Test description
* 4 Performance Analysis
* 5 Tools – Test equipment required
  + 5.1 Tools
  + 5.2 Test equipment
* 6 Stubs, Drivers and Test Data
* 7

1 INTRODUCTION

* 1. REVISION HISTORY
  2. PURPOSE AND SCOPE

This document is the Integration Test Plan Document for PowerEnjoy.

The main purpose is to describe in a clear and comprehensive way how we plan to accomplish the integration test, in order to integrate all different subsystems that make up our application, allowing in this way a correct and consistent realization of the whole project.

In particular, we’re going to provide:

* A global vision on which there are the principal subsystems and subcomponents that we want to integrate and test;
* A description of the approach that we will follow for test implementation;
* The effective order for components and subsystems integration, with a detailed description of each integration step, including the expected output for each input data;
* Some measures of expected components performance in order to completely satisfy the requirements;
* A list of all the tools and environments that will have to be used during test acivity.
  1. DEFINITIONS, ACRONYMS, ABBREVIATIONS
  2. REFERENCE DOCUMENTS

1. INTEGRATION STRATEGY
   1. ENTRY CRITERIA

In this paragraph, we are going to describe all the conditions necessary for starting the Integration Testing of our project.

First of all the Requirement Analysis and Specifications Document (RASD) must be written in order to have all the functional and non-functional requirements already defined between the customers and the developers. At the same time also the Design Document (DD) is required because it is necessary to be conscious about the components of the system and their interactions.

When the Integration Testing begins we must have already started the Coding and Unit Test phase. However these two phases can be done partially in parallel, progressively integrating modules in subsystems as soon as they are implemented.

For this reason we give an estimation of the minimum percentage of completion of the components of our system required in order to start this phase. This values are calculated reflecting the order in which components are integrated.

* 100% for the DataController component;
* At least 95% for the ClientController and CarController components;
* At least 80% for the CourseController component;
* At least 60% for the ReservationController component;
* At least 40% for the AssistantController component;
* At least 40% for the client-side components.

In this way we evaluated that the two phases would be done concurrently in the most efficient way.

* 1. ELEMENTS TO BE INTEGRATED

In this paragraph, we are going to list all the components that will be incrementally integrated during the Integration Testing.

For this purpose we took into account the Component Diagram defined in the Design Document in which we showed a high-level representation of the components of our system focusing on their relationships. However, some components will be decomposed in sub-components in order to have a model of our system with a lower level of abstraction. This will help us to define the best strategy for integrating the components in subsystems.

The components are the following:

* Data Controller: it is the component that interacts with the DBMS in order to obtain the data stored or to save new tuples.
* ClientController: it is the component that handles the client accounts and so it is decomposed in several sub-components that carry out the required task: ClientRegistrationController, CientLoginController and ClientProfileController. At the same time it also manages the current state of the client and so it is composed also by a ClientStateController.
* CarController: it is the component which handles all the information about cars: position, state and battery.
* CourseController: it is the component which manages the courses. This is one of the main functionality offered by our system and so it has to interact with other components such as the ParkingStationController, the SafeAreaController and the ChargeCalculator.
* ReservationController: this component is responsible for client reservations and interacts with almost the same components of the CourseController.
* AssistantController: it manages assistant accounts and so it has subcomponents similar to the ClientController component such as the AssistantRegistrationController, the AssistantLoginController and the AssistantProfileController.
* Some commercial components which have been already developed such as the DBMS and the PaymentSystem.
* The components on the client-side of our system and so the ClientApplication, the AssistantApplication and the CarApplication.
  1. INTEGRATION TESTING STRATEGY

In this paragraph we are going to explain the strategy that we will use for performing the Integration Testing describing the order in which components are considered.

In particular our main goal is to parallelize as much as possible this phase with the Coding phase and at the same discover bugs early in the most critical components of our system. For this reason we will partially adopt a bottom-up approach and so we will start integrating components that depends only on already developed and integrated components. This approach will be combined with a critical module testing and so we will start by integrating the most important sub-systems of components.

This is the order chosen:

* The DataController with the DBMS because these components are used in almost all the tasks. The DBMS is an external and commercial component and so we considered it already developed.
* The subsystem about clients and so the ClientController and its sub-components: the ClientRegistrationController, the CientLoginController, the ClientProfileController and the ClientStateController.
* The subsystem about the cars.
* The components relative to courses. Even if this is the most critical subsystem we decided to integrate it at this point in order to have all the components necessary for this task already developed and integrated.
* The components relative to reservations. They are considered now for the same reasons of the previous point.
* The subsystem about assistants and so the AssistantController and its sub-components: the AssistantRegistrationController, the AssistantLoginController and the AssistantProfileController.
  1. SEQUENCE OF COMPONENT / FUNCTION INTEGRATION

Following the strategy defined in the previous paragraph we are now going to describe how we integrate together our components. In order to show that a component is necessary to another one we use an arrow that connects the two components.

1. DATA CONTROLLER

As we said, we start by integrating the DataController component with the Database Management System (DBMS). This is an obvious choice because these components are required almost everywhere in our systems and so it is a direct consequence of our bottom up approach.



1. CLIENT MANAGEMENT SYSTEM

Then we consider the subsystem that manages clients. It is composed by several subcomponents which cover different aspects of the client administration and so can be integrated independently from one another. In all their functions they need to access or modify the data stored in the database and so they will be considered also with the DataController component.



So we list the groups of components that are integrated.



1. CAR MANAGEMENT SYSTEM

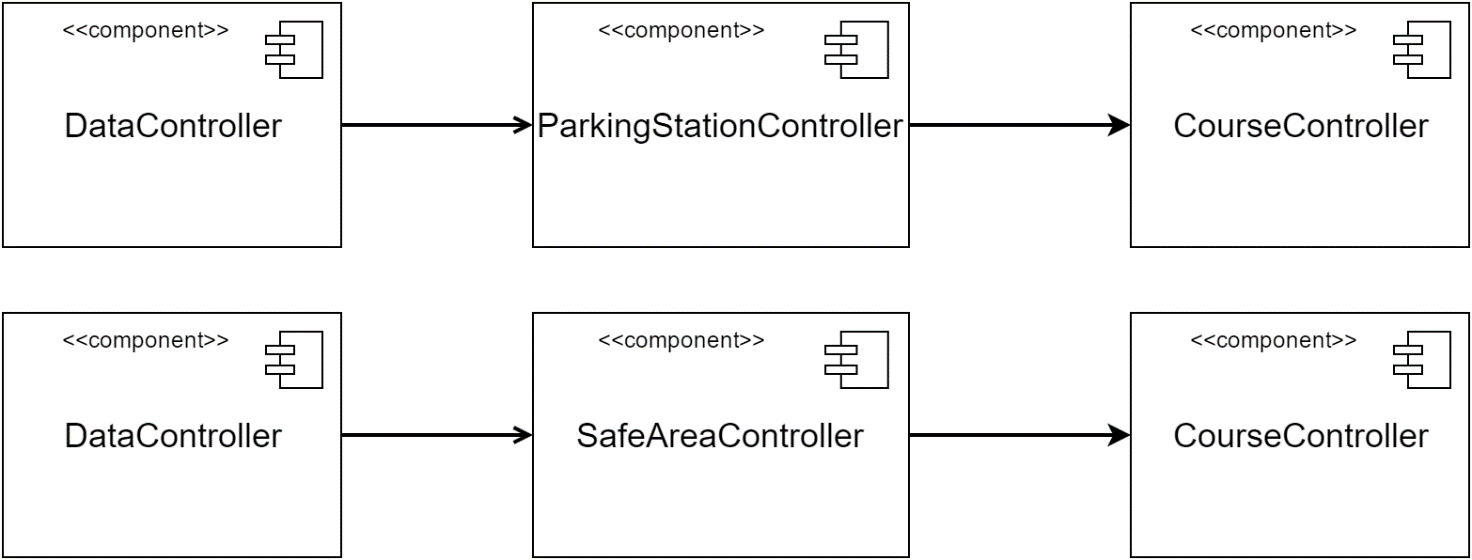
At this point the only component that can be integrated using only components that have been already developed is the CarController. It manages all the information about cars and so it has to interact with the database through the DataController, the CarStateController and the CarBatteryController.



1. COURSE MANAGEMENT SYSTEM

In this step, we have already integrated the subsystems about clients and cars and so now following the bottom-up strategy we could consider those of courses or assistants or reservation. We decided to start from the subsystem about courses because it certainly represents one of the most important functionalities of our system and so we applied a critical-module criterion.

In this subsystem there are several subcomponents that need to be integrated incrementally and the order will be described by the following diagrams. We start by considering the SafeAreaController and the ParkingStationcontroller whose functionalities are independent.



Then we consider the subcomponents that are necessary for the client payment. In this case we can use the PaymentSystem in a bottom-up approach because it is a commercial and external component that we considered already implemented and tested.



The CourseController can also interact directly with the DataController for example to insert a new Course in the database. So also these two components need to be considered.



Then the course controller has to modify the client and car state for example when the user decides to end his course. So we must integrate the CourseController also with the ClientController and the CarController.



At the end the subcomponents of this system are integrated together.

1. RESERVATION MANAGEMENT SYSTEM

Continuing to follow the critical-module approach we now consider the reservation subsystem. Reserving a car is one of the most important tasks of our system and the integration procedure has some similarities with the courses management system because some subcomponents are used in both cases.

The integration starts by considering the CarController and the ClientController which are necessary when the ReservationController needs to know or to modify the state of clients or cars. The ReservationController must interact also directly with the DataController in order to insert new reservation into the database.



At the end these components are also integrated together.



1. ASSISTANT MANAGEMENT SYSTEM

At the end we consider the subsystem with the task of managing the assistants. It has some similarities with the components that manages clients because also here there are subcomponents that handle registration, login and profile modification.

So also this subsystem is composed by subcomponents that are independent from one another and as a consequence it is not necessary to consider them all together.



* 1. SUBSYSTEM INTEGRATION SEQUENCE

In the following paragraph we provide some diagrams in which we show how the subsystems integrated in the previous section can be integrated together in order to obtain the complete application.

In the first one there are all the components that manages the services that our system offers. In order to make the diagram more readable we did not insert some low-level subcomponents such as those about client and assistant management.



In this schema, we show how the previous components interact with the client-side components.



1. INDIVIDUAL STEP - TEST DESCRIPTION

In this chapter we’ll analyse, for each pair of component that have to be integrated, the functionality that must be tested in order to guarantee a correct behaviour of the subsystem that they generate. In particular, for each possible input of the analysed functions, we’ll focus on the expected output, providing a brief description of the effects on the system.

* 1. Client management system
     1. ClientRegistrationController : DataController

|  |  |
| --- | --- |
| insertClient(clientData) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An object containing some null attributes | NullArgumentException is rised |
| An email already existing | InsertionFailedException is rised |
| An invalid credit card | InvalidCreditCardException is rised |
| An Invalid driving licence | InvalidDrivingLicenceException is rised |
| All valid data | A new client is inserted in the database, with a unique code |

|  |  |
| --- | --- |
| deleteClient(clientCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A valid and existing code | The client is removed from the database |

* + 1. ClientLoginController : DataController

|  |  |
| --- | --- |
| verifyCredential(username,password) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing username | InvalidArgumentException is rised |
| A valid and existing username, but an incorrect password | LoginFailedException is rised |
| A valid and existing username with the corresponding password | Login is successfully done |

|  |  |
| --- | --- |
| getHomeProfile(clientCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A valid and existing code | Client home is loaded |

* + 1. ClientProfileController : DataController

|  |  |
| --- | --- |
| getClientData(clientCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A valid and existing code | Client data are read from the database |

|  |  |
| --- | --- |
| modifyClientData(clientCode, newClientData) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| Invalid or incomplete new data | InvalidArgumentException is rised |
| A valid code and valid data | Client old data are replaced with new data in the database |

* + 1. ClientStateController : DataController

|  |  |
| --- | --- |
| getClientState(clientCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A valid and existing code | Client state is read from database |

|  |  |
| --- | --- |
| changeClientState(clientCode, newState) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| An invalid new state | InvalidArgumentException is rised |
| A valid code and a valid state | State of the corresponding client is updated in the database |

* 1. Car management system
     1. CarStateController : DataController

|  |  |
| --- | --- |
| getCarState(carCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A valid and existing code | Car state is read from database |

|  |  |
| --- | --- |
| changeCarState(carCode, newState, damage) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| An invalid new state | InvalidArgumentException is rised |
| A valid code and a valid state | State of the corresponding client is updated in the database, if newState is not available the state will include also the damage entity |

|  |  |
| --- | --- |
| lockCar(carCode) | |
| *Input* | *Effects* |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| unlockCar(carCode) | |
| *Input* | *Effects* |
|  |  |
|  |  |
|  |  |

* + 1. CarBatteryController : DataController

|  |  |
| --- | --- |
| getCarBattery(carCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A valid and existing code | Car battery level is read from database |

|  |  |
| --- | --- |
| setOnCharge(carCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A code corresponding to a car which is already on charge | CarAlreadyChargingException is rised |
| A valid and existing code, corresponding to a car which is not on charge | Car is setted on charge in the database |

|  |  |
| --- | --- |
| disconnectCarCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A code corresponding to a car which is already disconnected | CarAlreadyDisconnectedException is rised |
| A valid and existing code, corresponding to a car which is on charge | Car is setted not on charge in the database |

* + 1. CarController : DataController

|  |  |
| --- | --- |
| getNearestCarAvailable(position) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing position | InvalidArgumentException is rised |
| A valid and existing position with no nearby available cars | Empty list is returned |
| A valid and existing position with some nearby available cars | A list with nearest available cars to the specified position is returned |

|  |  |
| --- | --- |
| getCar(carCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A valid and existing code | Car with the corresponding code is returned |

* 1. Course management system
     1. ParkingStationController : DataController

|  |  |
| --- | --- |
| getNearestParkStationAvailable(position) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing position | InvalidArgumentException is rised |
| A valid and existing position with no nearby available park stations | Empty list is returned |
| A valid and existing position with some nearby available park stations | A list with nearest available park stations to the specified position is returned |

|  |  |
| --- | --- |
| getRechargerAvailable(parkStationCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A valid and existing park station code with no recharger available | Empty list is returned |
| A valid and existing park station code with some recharger available | A list of available recharger in the corresponding park station is returned |

|  |  |
| --- | --- |
| setRechargerOccupied(parkStationCode, rechargerCode) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing park station code | InvalidArgumentException is rised |
| An invalid or not existing recharger code in the corresponding park station | InvalidArgumentException is rised |
| All valid codes, corresponding to a recharger that is already occupied | RechergerAlreadyOccupiedException is rised |
| All valid codes, corresponding to a recharger that is not occupied | Recharger is setted occupied in the database |

|  |  |
| --- | --- |
| setRechargerFree(parkStationCode, rechargerCode) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing park station code | InvalidArgumentException is rised |
| An invalid or not existing recharger code in the corresponding park station | InvalidArgumentException is rised |
| All valid codes, corresponding to a recharger that is already free | RechergerAlreadyFreeException is rised |
| All valid codes, corresponding to a recharger that is occupied | Recharger is setted free in the database |

* + 1. SafeAreaController : DataController

|  |  |
| --- | --- |
| verifySafeArea(position) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing position | InvalidArgumentException is rised |
| A position which is not a safe area | Return false |
| A position which is a safe area | Return true |

* + 1. CourseController : PaymentSystem

|  |  |
| --- | --- |
| makePayment(paymentData) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| Invalid payment data | InvalidArgumentException is rised |
| Valid payment data | The transaction is executed |

* + 1. CourseController : ChargeCalculator

|  |  |
| --- | --- |
| calculateCharge(courseTime) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| Invalid course time | InvalidArgumentException is rised |
| Valid course time in second | Course charge is calculated proportionally to the course time and the value is returned |

|  |  |
| --- | --- |
| applyDiscount(discount[], courseCharge) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| Invalid course charge | InvalidArgumentException is rised |
| Array of discount with at least one invalid element | InvalidArgumentException is rised |
| Array of valid discounts and valid course charge | The total amount of discount is calculated and is applied to the course charge |

* + 1. CourseController : CarController

|  |  |
| --- | --- |
| createCourseCarRelation(course, carCode) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A code corresponding to a car which already have a course relation | InvalidCreationException is rised |
| A course which already have a relation with a car | InvalidCreationException is rised |
| A valid and existing code corresponding to a car with no course relation, a course with no car relation | Database is updated adding the relation between course and car |

|  |  |
| --- | --- |
| deleteCourseCarRelation(course, carCode) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A code corresponding to a car which isn’t in relation with course | InvalidArgumentException is rised |
| A course which isn’t in relation with the car | InvalidArgumentException is rised |
| A valid and existing code corresponding to a car in relation with course | Database is updated removing the relation between course and car and setting to null the corresponding fields |

* + 1. CourseController : ClientController

|  |  |
| --- | --- |
| createCourseClientRelation(course, clientCode) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A code corresponding to a client which already have a course relation | InvalidCreationException is rised |
| A course which already have a relation with a client | InvalidCreationException is rised |
| A valid and existing code corresponding to a client with no course relation, a course with no client relation | Database is updated adding the relation between course and client |

|  |  |
| --- | --- |
| deleteCourseClientRelation(course, clientCode) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A code corresponding to a client which isn’t in relation with course | InvalidArgumentException is rised |
| A course which isn’t in relation with the client | InvalidArgumentException is rised |
| A valid and existing code corresponding to a client in relation with course | Database is updated removing the relation between course and client and setting to null the corresponding fields |

,

* + 1. CourseController : DataController

|  |  |
| --- | --- |
| createCourse(dataStart, timeStart) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid time or an invalid data | InvalidArgumentException is rised |
| A valid time and a valid data | A new course with the corresponding dataStart and timeStart is created and added to the database |

|  |  |
| --- | --- |
| endCourse(course) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| A course already terminated | CourseAlreadyTerminatedException is rised |
| A course not already terminated | Time end and data end are registered to the corresponding course in the database |

* 1. Reservation management system
     1. ReservationController : CarController

|  |  |
| --- | --- |
| createReservationCarRelation(reservation, carCode) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A code corresponding to a car which already have a reservation relation | InvalidCreationException is rised |
| A reservation which already have a relation with a car | InvalidCreationException is rised |
| A valid and existing code corresponding to a car with no reservation relation, a course with no car relation | Database is updated adding the relation between reservation and car |

|  |  |
| --- | --- |
| deleteReservationCarRelation(reservation, carCode) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A code corresponding to a car which isn’t in relation with reservation | InvalidArgumentException is rised |
| A reservation which isn’t in relation with the car | InvalidArgumentException is rised |
| A valid and existing code corresponding to a car in relation with reservation | Database is updated removing the relation between reservation and car and setting to null the corresponding fields |

* + 1. ReservationController : ClientController

|  |  |
| --- | --- |
| createReservationClientRelation(reservation, clientCode) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A code corresponding to a client which already have a reservation relation | InvalidCreationException is rised |
| A reservation which already have a relation with a client | InvalidCreationException is rised |
| A valid and existing code corresponding to a client with no reservation relation, a course with no client relation | Database is updated adding the relation between reservation and client |

|  |  |
| --- | --- |
| deleteReservationClientRelation(reservation, clientCode) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A code corresponding to a client which isn’t in relation with reservation | InvalidArgumentException is rised |
| A reservation which isn’t in relation with the client | InvalidArgumentException is rised |
| A valid and existing code corresponding to a client in relation with reservation | Database is updated removing the relation between reservation and client and setting to null the corresponding fields |

* + 1. ReservationController : DataController

|  |  |
| --- | --- |
| startResCountdown(reservation) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| A reservation which already has an active reservation countdown | InvalidArgumentException is rised |
| A reservation which not already has an active reservation countdown | A one-hour reservation countdown is started on the corresponding reservation |

|  |  |
| --- | --- |
| stopResCountdown(reservation) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| A reservation which not already has an active reservation countdown | InvalidArgumentException is rised |
| A reservation which already has an active reservation countdown | The one-hour reservation countdown is stopped and reset |

|  |  |
| --- | --- |
| startCouCountdown(reservation) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| A reservation which already has an active courtesy countdown | InvalidArgumentException is rised |
| A reservation which not already has an active courtesy countdown | A five-minutes courtesy countdown is started on the corresponding reservation |

|  |  |
| --- | --- |
| stopCouCountdown(reservation) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| A reservation which not already has an active reservation countdown | InvalidArgumentException is rised |
| A reservation which already has an active reservation countdown | The five-minutes courtesy countdown is stopped and reset |

* 1. Assistant management
     1. AssistantRegistrationController : DataController

|  |  |
| --- | --- |
| insertAssistant(assistantData) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An object containing some null attributes | NullArgumentException is rised |
| An email already existing | InsertionFailedException is rised |
| A name of an assistant wich is not recognised as an employee of PowerEnjoy society | InsertionFailedException is rised |
| All valid data | A new assistant is inserted in the database, with a unique code |

|  |  |
| --- | --- |
| deleteAssistant(assistantCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A valid and existing code | The assistant is removed from the database |

* + 1. AssistantLoginController : DataController

|  |  |
| --- | --- |
| verifyCredential(username,password) | |
| *Input* | *Effects* |
| Some null parameter | NullArgumentException is rised |
| An invalid or not existing username | InvalidArgumentException is rised |
| A valid and existing username, but an incorrect password | LoginFailedException is rised |
| A valid and existing username with the corresponding password | Login is successfully done |

|  |  |
| --- | --- |
| getHomeProfile(assistantCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A valid and existing code | Assistant home is loaded |

* + 1. AssistantProfileController : DataController

|  |  |
| --- | --- |
| getAssistantData(assistantCode) | |
| *Input* | *Effects* |
| Null parameter | NullArgumentException is rised |
| An invalid or not existing code | InvalidArgumentException is rised |
| A valid and existing code | Assistant data are read from the database |

PERFORMANCE ANALYSIS

With the performance analysis we evaluate in which conditions and with which tools the system works properly. A complete analysis can be performed during the implementation of the infrastructure, monitoring how the system responds with the structures used. However, is useful perform a preliminary analysis for the infrastructures required from the performances requested.

For the correct running of the mobile application, the smartphone has to provide at least:

* The correct tracking through GPS;
* Camera with 5Mpx;
* 3,5 MB of memory for the app;
* 80 MB of RAM to be execute;
* CPU with 1GHz in one core.

All these constraints are approximate, and some more close to real values can be detected in the future, with the implementation of the infrastructure.

The application will be developed with the framework Apache-Cordova for all the platforms. Than it is tested with a specific SDK for different smartphones.

1. TOOLS - TEST EQUIPMENT REQUIRED
   1. TOOLS

The tests are all executed with some automatized tools, that decrease the writing time for the test and simplify and get more readable the solutions. It is even easier to find the error in the code, thanks to the debugging offer by the IDE Netbean. The tools used will be:

* Junit;
* Mockito;
* Arquillian;
* Apache JMeter.

For the unit test and components test will be used JUnit and Mockito, while for the subsystem test will be used Arquillian too. These three are the tools for testing the Functional Requirements.

For testing the NonFunctional Requirements will be used Apache Meter.

The roles of these tools are different, and each one is relative to a different scope:

* Junit: is a unit testing framework. Its scope is to verify the correct behaviour, and so the result of a function through assertions. With this tool can be verified the interactions between components verifying if they produce the expected result. We will use it even to verify if the methods will throw the correct exceptions with some specific parameters.
* Mockito: is a tool that permit to mock an environment for test a unit. It allows the interaction test between objects. It provides a mocking of some methods creating a scaffolding defined with stubs.
* Arquillian: is used for tests with containers. The test is similar to the Junit test, but add the container, the environment in which run the class. It also uses archives, defined using ShrinkWrap, that define the micro-environment with only the strict data necessary to the class in test to work. With Arquillian can be choose the container in which execute the class to test.
* Apache JMeter: is used to evaluate performances of a system. It has a GUI and can evaluate performances relative to any software (web applications) about heavy load on a DB or similar. It is possible to check even the behaviour of multiple test servers in a single interface. It is possible to create test plans that emulate the interaction of a user with the GUI of the application, and so evaluate the performances with many users.

(So we will evaluate the different performances in the different devices with the appropriate tools.)

* 1. TEST EQUIPMENT

Our system involves many different devices with different features, all to be tested to monitoring the performances.

For how concerns the mobile app, the possible different devices that can be involved are the smartphone with the different operative system. So the three smartphone that will be used are:

* Google Nexus 5
* iPhone 6
* Nokia Lumia 750

For the onBoard system, we will use a tablet with OS Android.

6 STUBS, DRIVERS AND TEST DATA