POLITECNICO
MILANO 1863

PowerEnJoy

Integration Test Plan Document

Luca Scannapieco - 877145
Andrea Pasquali - 808733
Emanuele Torelli - 876210

15/01/2017

Table of contents

1. Introduction

1.1. Purpose and scope

Integration testing is an important phase during the development of PowerEnJoy system, since its aim is to guarantee that all the components and subsystems interoperate correctly among them with respect to the requirements they are supposed to fulfil and without exhibiting unexpected behaviours. The purpose of this Integration Test Plan Document is to explain the criteria with which the components that build up the system will be integrated and how the integration testing activities will be organized. In the following sections we're going to provide:

- The status that is supposed to be achieved by the project before the integration of the outlined elements can begin.
- The identification of the subsystems and their components that must be tested during the integration activities.
- 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 tests that have to be performed during the integration activities, including the input data and the expected output.
- A description of the tools that will be used during the testing activities and the reason why we decided to adopt them.
- The stubs or drivers needed for the integration of the components and subsystems, according to the adopted integration approach.

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 as much as possible, according to the functionalities that each component is supposed to ensure. For example, it's reasonable to suppose that the core business components of the server, such as the views, the controllers and the model, must have at least 90% tested. On the other hand, since the client side components contain few application logic but many graphical elements, the required percentage of testing has to be at least 70%. Of course these percentages are supposed to increase as long as the integration phase goes on.

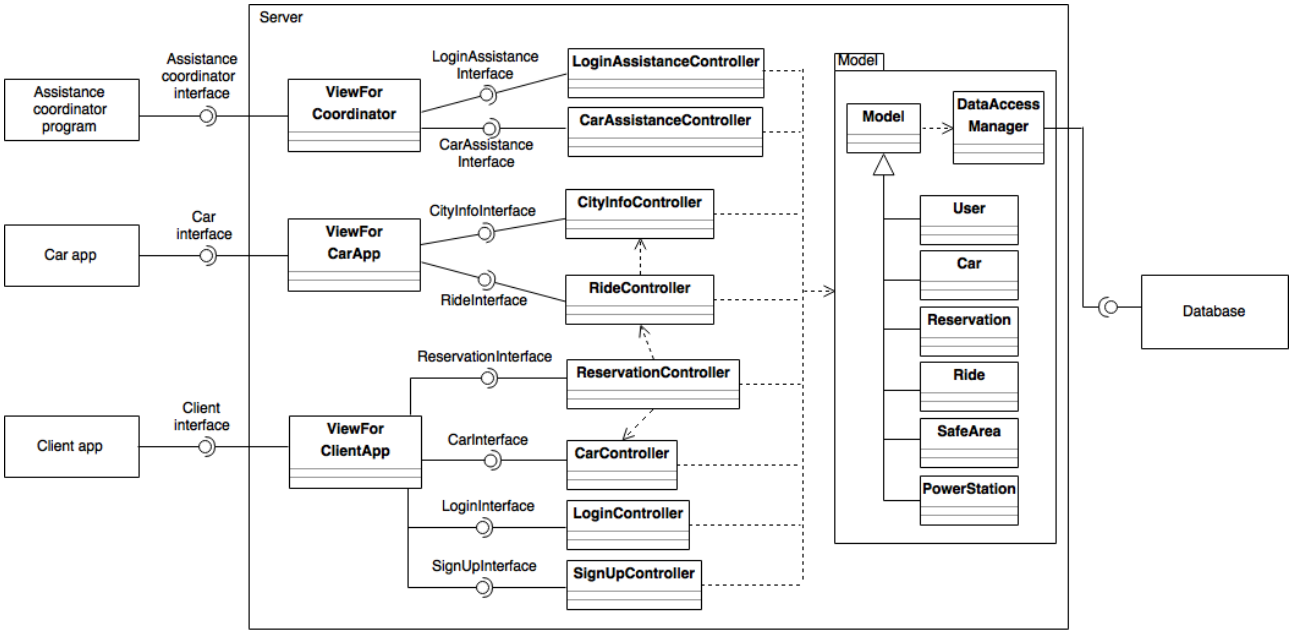
2.2. Elements to be integrated

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).

As explained in the DD, the names of the components of the server follow the camel case notation (e.g. "RideController" instead of "ride controller"), with a remarkable similarity with the Java notation for classes names, in order to emphasize the fact that they are software components. The external components to the server, instead, don't follow this notation.

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 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 ones, 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 if we would have strictly followed the bottom-up approach we would integrate the client side as the last component. Instead, 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 the efficiency as well.

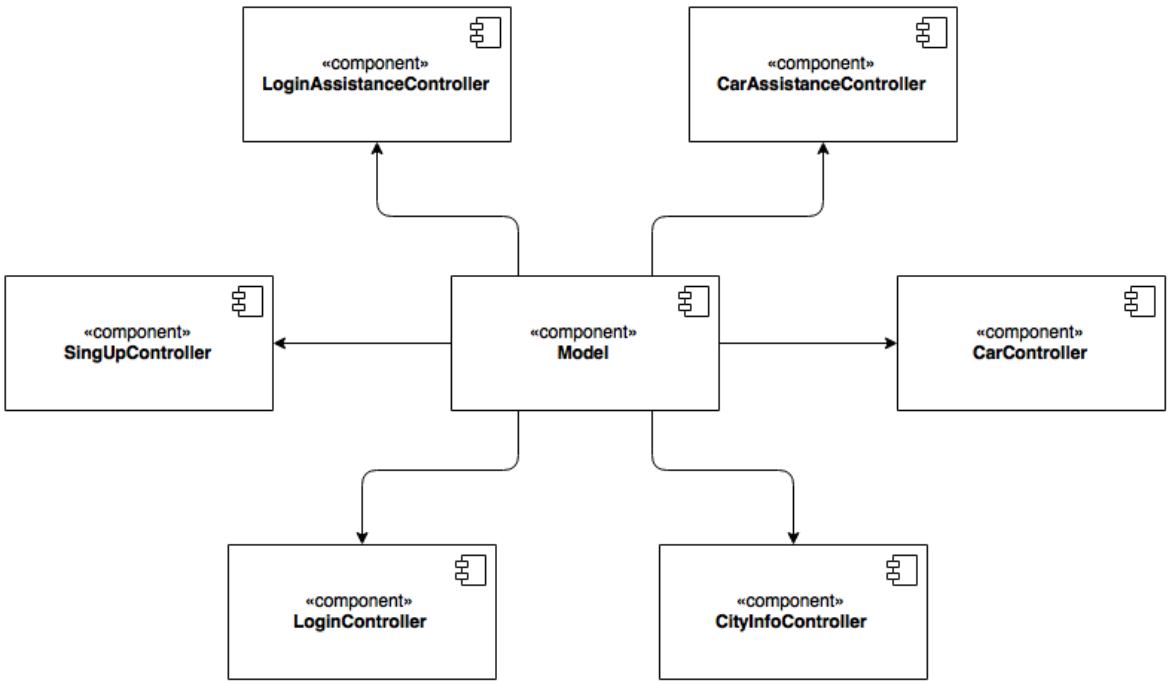
2.4. Sequence of component/function integration

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

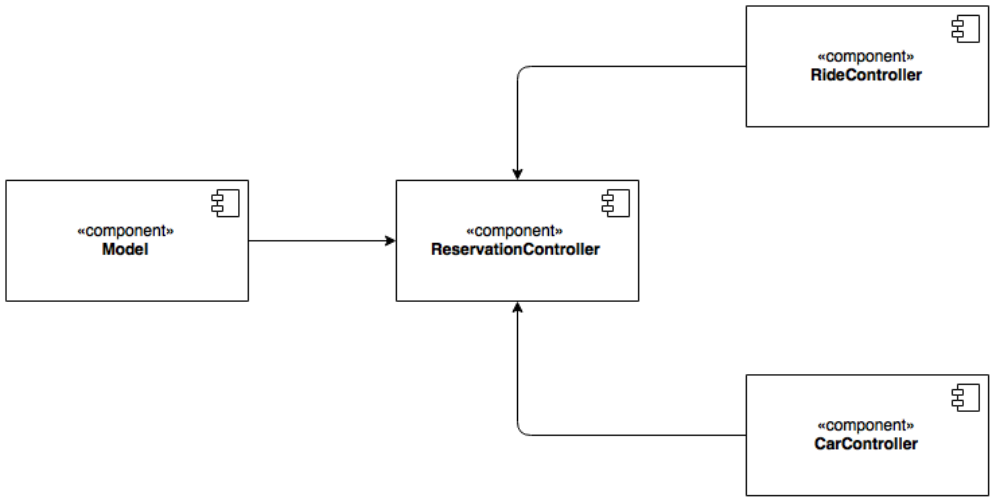
According with the critical-module-first policy described in the chapter 2.3 the first two elements to be integrated are the database and the model because they refer to the data of our system and thus they are the most used components.

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

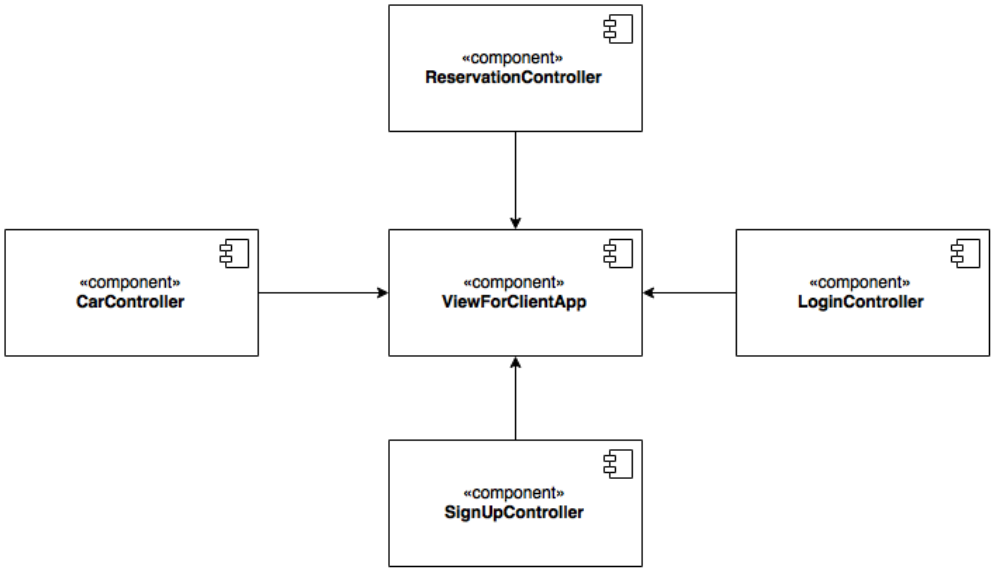


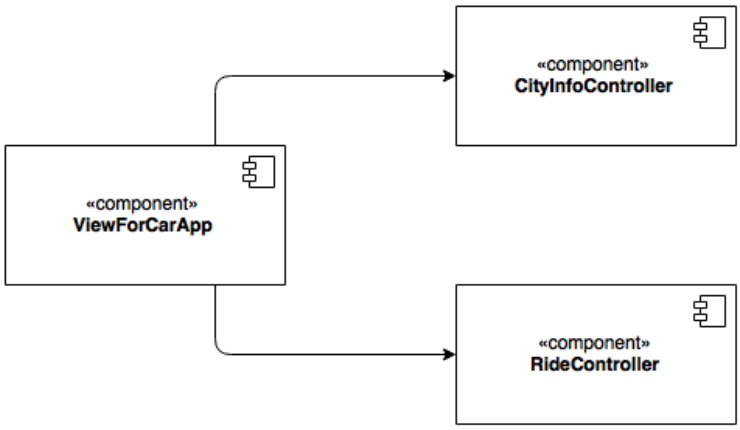
Now we can proceed adding the controllers interacting with the already implemented controllers: the next controller is RideController that only interact with CityInfoController.

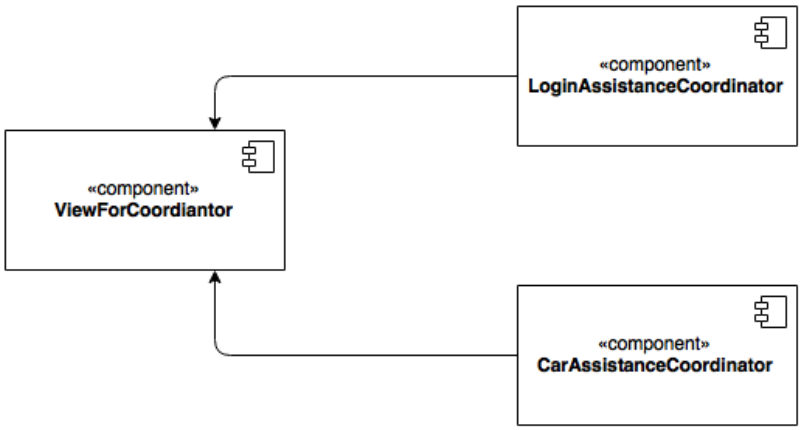
The last controller to integrate is ReservationController that uses RideController and CarController.



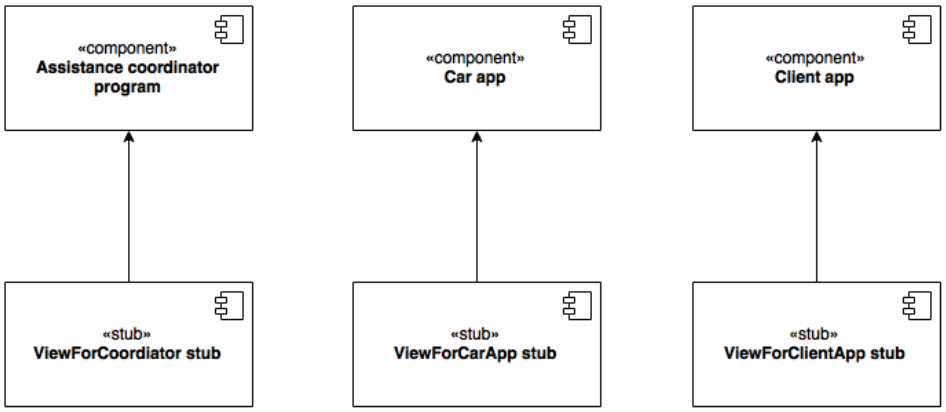
Once we have integrated all the controllers we can finally integrate the three views.





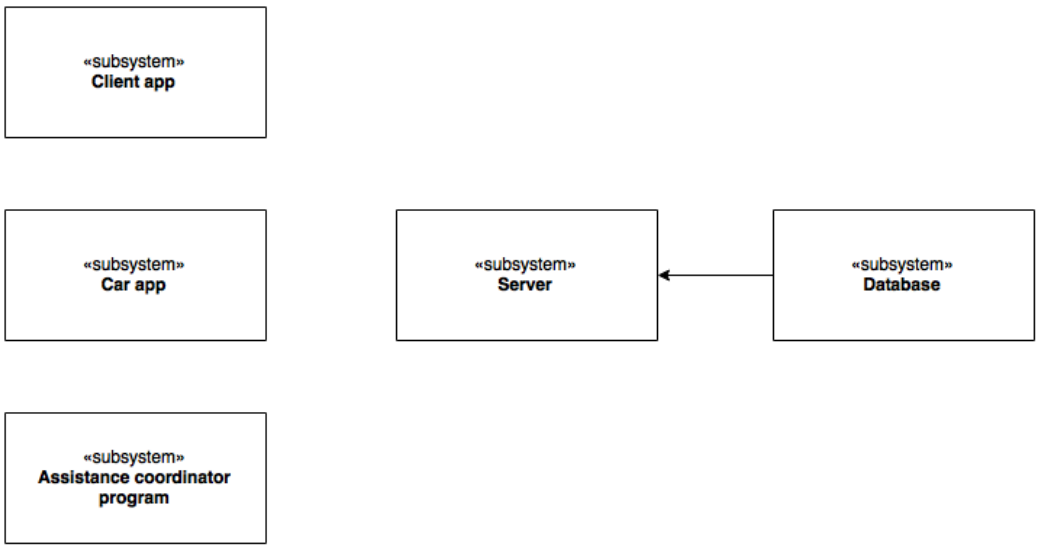


As explained in the section 2.3. we can integrate the three components of the client side in parallel with those of the server side. We only need to implement the stubs for the three views used by the three client components. These stubs will be substituted by the real views once the server subsystem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

In this section the majority of the integration tests are described. For the sake of simplicity, some redundant tests have been cut off, in our case those concerning the integration with the views, because they involve the same parameters that are subsequently routed to the controllers or to the client side applications.

In the following subsections we always refer to the model taking into account that, thanks to the DataAccessManager component, it contains all the logic necessary to retrieve and manage the data from the database. Furthermore, we won't report the tests related to the database queries since we suppose that the DataAccessManager component has already been exhaustively tested.

3.1. Client app, LoginController

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

3.2. Client app, 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. LoginController, Model

checkCredentials(email,password)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
An email non-registered	Returns false
An email that correspond to its password of a registered user	Returns true

3.4. SignUpController, Model

CreateNewUser(name, surname, phoneNumber, email, address, SSN, creditCard, licenceNumber)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
	An InvalidArgumentValueException is raised
A valid set of parameters	Returns true or false depends on the car availability

CreateNewUser(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	Create a new object “user” with all parameters set with the inputs.

DeleteUser(email)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
An non-registered email	An InvalidArgumentValueException is raised
A valid email	Returns true and delete the object “user” corresponding to the input email.

3.5. 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

3.6. Client app, ReservationController

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

3.7. CarController, Model

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

getPosition(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, Model

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)	
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)	
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

setReservationCar(reservation,car)	
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)	
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)	
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)	
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(car)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Acar with a non-existing Id	An InvalidArgumentValueException is raised
A valid car	Ride controller will initialize a new Ride setting the right reservation as its attribute and will return true.

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.

3.10. 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

3.11. CityController, Model

getSafeArea(position)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
A valid parameter	Returns true if the position is in a safe area, false otherwise

3.12. RideController, Model

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	An InvalidArgumentValueException is raised
A valid parameter of a car that has only one running ride	Returns true

setTerminated(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

setReservationRide(ride,reservation)	
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".

3.13. ReservationController, RideController

readyToStart(reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
A non-existing reservation	An InvalidArgumentValueException is raised
A non-expired reservation	An InvalidArgumentValueException is raised
An expired reservation	The ride controller will launch the welcome() method to the car app.

3.14. RideController, Car app

welcome()	
Input	*Effect*
nothing	The car app will show the welcome message on the screen.

3.15. Car app, RideController

activateMoneySavingOption(car, targetPosition)	
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

startRide(car)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
A car with a non-existing Id	An InvalidArgumentValueException is raised
A car with a non-expired reservation	An InvalidArgumentValueException is raised
A valid parameter	Returns true and the ride controller will create a new Ride

finishRide(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 valid parameter	The ride controller will set True the attribute “terminated” of the ride

3.16. RideController, Model

finishRide(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 valid parameter	The ride controller will set True the attribute “terminated” of the ride

activateMoneySavingOption(car, targetPosition)	
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 valid set of parameters	Returns true and changes the attribute “moneySavingOptionActivated” to true and “targetPosition” the input targetPosition.

3.17. Assistance coordinator program, LoginAssistanceController

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

3.18. Assistance coordinator program, CarAssistanceController

getAllCars()	
Input	*Effect*
nothing	Returns the set of all available, not available and out of order cars

changeCarStatus(car, status)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
A car with a non-existing Id	An InvalidArgumentValueException is raised
A status different from “available”, “not available”, “out of order”	An InvalidArgumentValueException is raised
A valid set of parameters	Returns true

3.19. CarAssistanceController, Model

changeCarStatus(car, status)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
A car with a non-existing Id	An InvalidArgumentValueException is raised
A status different from “available”, “not available”, “out of order”	An InvalidArgumentValueException is raised
A valid set of parameters	Returns true and changes the attribute “status” of the car

4. Tools and test equipment required

In order to test PowerEnJoy applications, we are going to use mainly the JUnit framework to implement unit tests in Java. It is a good instrument to check that methods and classes work in the correct way and are producing the right results. As showed in the previous section, we have to check that every method responds with the correct effect to a determined input parameter, returning the right object or raising the expected exception when parameters are not valid.

To test interaction between objects we will use also Mockito, a framework that supports, really useful for our parallel integration strategy in which we will test client side together with the server component, so we need to implement the stubs for the three views used by the three client components (assistance coordinator program, car app and client app). Mockito, with its scaffolding, gives us the possibility to define stubs, in order to test the components that we cannot test in isolation.

Finally, to test our applications performances and that it will support at least 500 users access at the same moment, we will use JMeter, simulating the minimum number of users request that we want to ensure.

As for the tests regarding the mobile applications, we can rely on various tools that can be integrated to the most popular IDEs for the development of mobile applications, specifically Xcode for iOS, Android Studio for Android, and Visual Studio for Windows.

Finally, there is still the need of performing a significant amount of manual operations, despite the usage of the described automated testing tools, especially because many of the planned testing activities require the identification the appropriate set of testing data.

To be sure that the mobile application and the website will be responsive and adaptable to all mobile devices of many brands and dimensions, it is necessary to test it directly on each model. So, these devices (or, at least, the corresponding emulators) are required:

- A model of each iPhone starting from iphone 4.
- A model of an Android device for each display size from 3’5’’ to 5’5’’.
- A model of a Windows Mobile device for each display size from 3’5’’ to 5’5’’.
- A model of iPad for each display size.
- A model of an Android tablet for each display size.

Instead, for what concerns the website, it’s enough to perform tests on a computer with a modern web browser installed, like Google Chrome, Mozilla Firefox, Safari or Microsoft Edge, because there are not specific requirements concerning PC performances or display dimensions.

It will also be necessary a device that reflects those that will be installed on every car, in order to test the car app.

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

The tools used for testing the integration of the components are described in section 4.1.

6.3. Hours of work

For redacting and writing the Integration Test Plan Document we spent approximately 25 hours per person.

6.4. Changelog

No changes in the document for the moment.