ITPD

Integration Test Plan Document

Version: 1.0.0

15th January 2017

Edited by:

Marzia Degiorgi - 878360

Giulia Leonardi - 877491

Valentina Ionata - 808678

Index

1. Introduction

1.1. Revision History

Version	Date	Summary
1.0.0	**15/01/2017**	Initial Release

1.2. Purpose and Scope

The ITPD document aims to describe the integration test strategy that we choose to adopt for verifying and validating the component specified in the DD document.
The main purpose is to give a fully description of the testing that we are planning to do on our "Power Enjoy" system, in order to verifying that all the future described in the previous documents behave as expected.
All the Power Enjoy components will be test as described in the body of this document.

1.3. List of Definitions and Abbreviations

1.3.1 Acronyms

- REST: Representational State Transfer
- DB: Database
- DD: Design Document
- RASD: Requirement and Specification Document
- ITPD: Integration Testing Plan Document
- API: Application Programming Interface

1.3.2 Abbreviations

Abbreviation	*Definition*
T.Y.X	Identifier of the integration test in the section number Y.X [e.g T.1.1]

1.4. List of Reference Documents

- ➢ http://www.keynote.com/resources/white-papers/testing-strategies-tactics-for-mobile-applications

  “Verification and Validation,part I”, Michele Guerriero
- ➢ “Verification and Validation,part II”, Michele Guerriero
- ➢ “Verification Tools”, Michele Guerriero
- ➢ “Integration Testing Example Document”

2. Integration Strategy

2.1. Entry Criteria

In the previous documents (RASD and DD) we have described the requirements and the design structure of our system, the interaction between the components and what are the various features that our application must have. The completeness of these two documents is very important to start the drafting of the Integration Test Plan Document.
In the testing analysis that we aim to plan in this document, it is important to consider the percentage of completion of the components; Due to that, most of *Power Enjoy* modules should be complete to be tested, the incomplete modules needed for testing can be covered by the development of Stubs or Drivers.

2.2. Elements to be Integrated

In this chapter we are going analyze how to best integrate the components of our system. We have seen previously, in the Component View of the Design Document, which are the components of the system with an higher level of abstraction; Now we're going to specify them at a lower level in order to understand how they should be integrated. Then we go to better analyze the high-level components:

User Controller: checks the correctness of the User credentials when he is logging on to the system and controls the user data entered during the registration; handles the change of the profile and takes care of payments. This high level component is composed, at a lower level, by a ***Registration Manager***, a ***Login Manager*** and a ***Modify Information Manager***.

Reservation Controller: manages everything about the reservation and is divided into two subcomponents, ***Modify Reservation Manager*** that add the new reservations in the database and do several controls and ***Available Vehicles Manager*** that find all the cars that can be reserved by the user.

Car Controller: is the component that handles all the actions that the user can do inside the car and all the operations concerning the ride. The Car Controller interacts with other low-level components: the ***Ride Manager*** that manages how to end a ride; the ***Options and Discounts Manager***, that handles the money saving option, the tourist mode and the application of discount and charges to the user; the ***Emergency Situation Manager*** that do some actions when the battery level is low, there are damages or the car is parked in an outside area.

Area Controller: manages the position of the cars and calculates the special parking areas for the money saving option.

Operator Controller: this component is used to check the login and registration data of the operator and to notify him if there is a critical situation where he has to operate. The functionalities of this component are used through the subcomponents ***Operator Registration Manager***, ***Operator Login Manager*** and ***Operations Manager***.

Model: this component, due to the calling of functions from the different components of the system, is responsible for retrieving data from the database and to make changes inside it.

There are some other pre-existing components that are external to our system and that offer some services like the ***Map Service***, the ***Payment Gateway***, the ***Push Gateway***, the ***Email Gateway*** and the ***Database.*** Then for the client and operator interface there are the ***On-board Application*** and the ***Mobile Application*** components.

2.3. Integration Testing Strategy

Before delving into the exact order in which we are going to integrate the various components of the system, in this module we are going to explain the approach selected for the integration Testing and the reasons that led us to these choices.
For our integration Testing, we decided to rely mostly to the ***bottom-up*** approach, as it seems the most appropriate to the structure of our system. The bottom-up approach

includes a test sequence that starts by the submodules and arrive successively to the main modules; if the main modules are not developed a temporary program called *Drivers* is used to simulate them. The use of this approach has several advantages:

- ➢ Tests are more realistic, since, starting to integrate the components that do not depend on other or that depend only on those already integrated, it is possible to effectively verify that all function calls take place properly, particularly if the components on which are called have already been implemented.
- ➢ The software creation processes can be performed in parallel, since to begin the testing phase is not necessary to have implemented all the system classes, but is enough to have implemented the main ones. This means greater efficiency and less time to complete the realization of the software.
- ➢ Starting the test by subcomponents, critical components are tested before the others and this allows, in case of errors or mistakes, to find them early in the process.

2.4. Sequence of Component/Function Integration

In this section is illustrated, step by step, the sequence of integration of the various components of the system, starting from the low level components and forming subsystems, which will be later integrated to the high level ones.
The direction of the arrows represents the order of integration: the components from which the arrows come are needed by the other components to whom are connected, because they contain the functions that are used by the second ones, so are to be completed first.

2.4.1. Software Integration Sequence

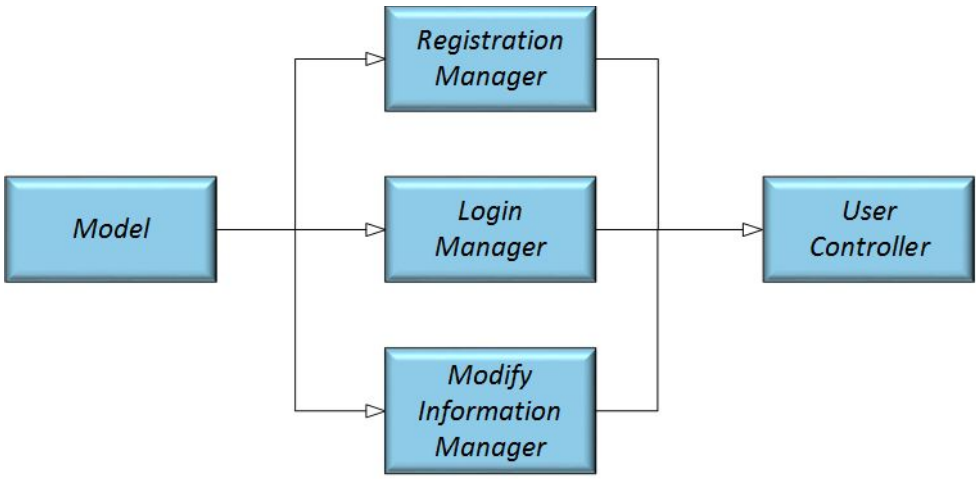
The first components that we are going to integrate together will be:

- ➢ The ***Model*** class with the ***Database***, which is the first class to be integrated in importance according to the critical module first approach, since it contains all the data relevant to the system.

- ➢ The components that represent the various functionalities of the main controllers with the ***Model***, in order to form the main subsystems which will then form the high-level components.

● User Subsystem

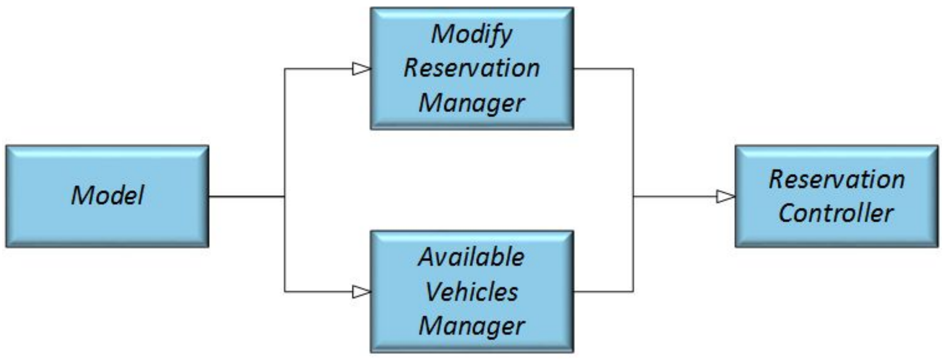
The User subsystem represents all the features that the user can make interacting with our system, such as registering, accessing the application, modifying the personal data. In order to do this we are going to integrate first the ***Model*** with the ***Registration Manager***, the ***Login Manager*** and the ***Modify Information Manager*** and next to integrate the last ones with the ***User Controller***.



Subsystem 1: User Controller

● Reservation Subsystem

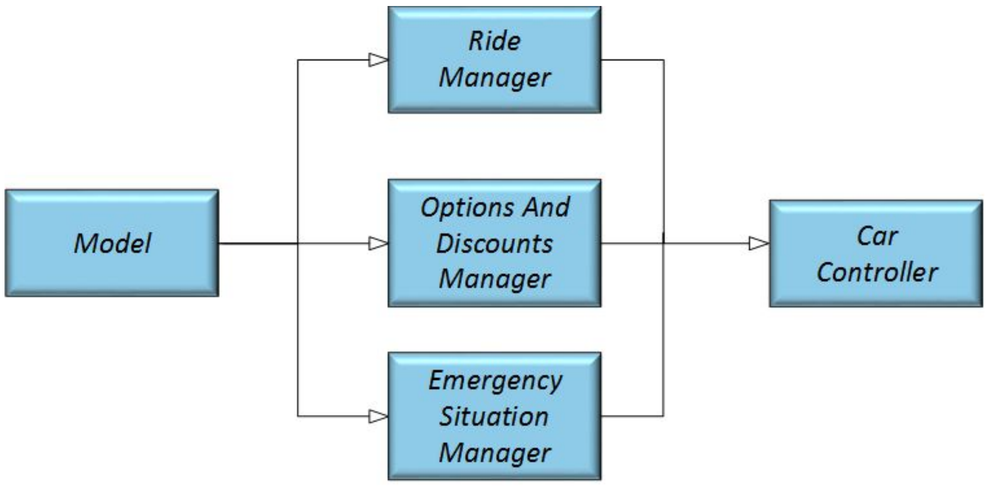
The Reservation subsystem combines the functionality that allow you to reserve a car, to cancel a reservation, to modify its data and find all the vehicles available to the user. It is composed by integrating the ***Modify Reservation Manager*** and the ***Available Vehicles Manager*** with the ***Model*** and than with the ***Reservation Controller***, following the bottom-up approach.



Subsystem 2: Reservation Controller

- **Car Subsystem**

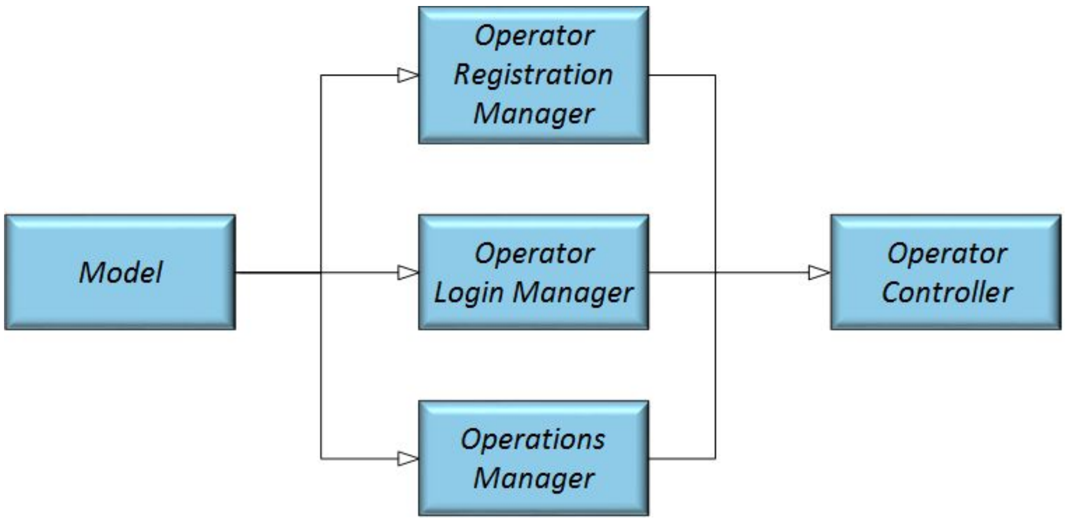
The Car Subsystem is obtained integrating the ***Model*** with the three functions of the car that use it: the ***Ride Manager*** that represents all the operations that can be performed relative to the race, the ***Options And Discounts Manager***, which is responsible for applying the discounts, to charge the User for the ride and to enable the navigation options and the ***Emergency Situation Manager*** that manages conditions of the car when it is in a state of emergency. This three classes are then integrated to the ***Car Controller*** to form the subsystem.



Subsystem 3: Car Controller

- ## Operator Subsystem

The Operator Subsystem offers the same functionalities of the User Subsystem of registration and login through the classes ***Operator Registration Manager*** and ***Operator Login Manager***, and there is also an other component, the ***Operations Manager***, that handle to notify the operator if there are critical situations. This component are integrated first with the ***Model***, next with the ***Operator Controller*** like the previous modules.



Subsystem 4: Operator Controller

- ## Area Subsystem

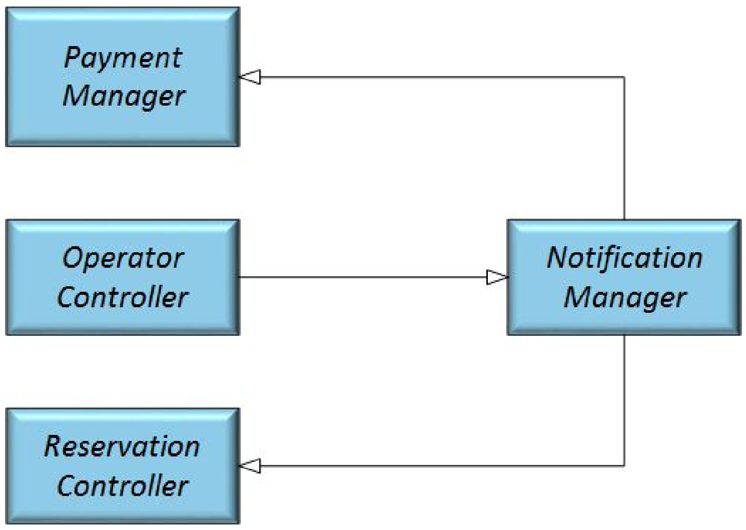
The Area Subsystem manage the position of cars and parking areas for the money saving option and is simply an integration of the ***Model*** and ***Area Controller*** components.

Model
Area Controller

Subsystem 5: Area Controller

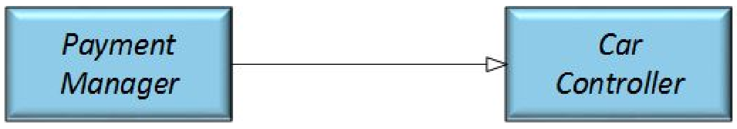
The next step is to integrate some support classes with the subsystems that we have just obtained from integration of low-level components:

- The ***Notification Manager*** must be integrated with all controllers that should receive notifications, such as the ***Operator Controller***, that takes care to notify the operator of emergency situations, the ***Reservation Controller***, which receives notification about the reservation and the ***Payment Manager***, that takes care of payment alerts.



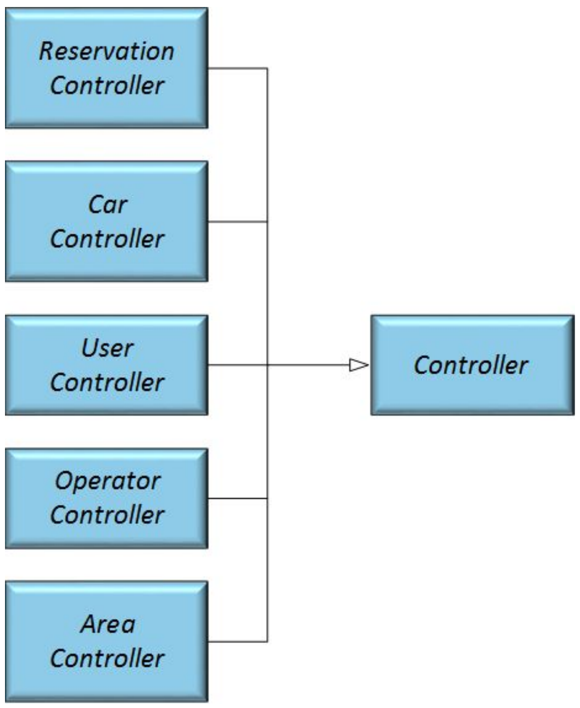
System Integration 1:Notification Manager

- The ***Payment Manager,*** that must charge the cost of the ride to the User, have to be integrated to the ***Car Controller***, that deals with the discounts and the payment of the ride.

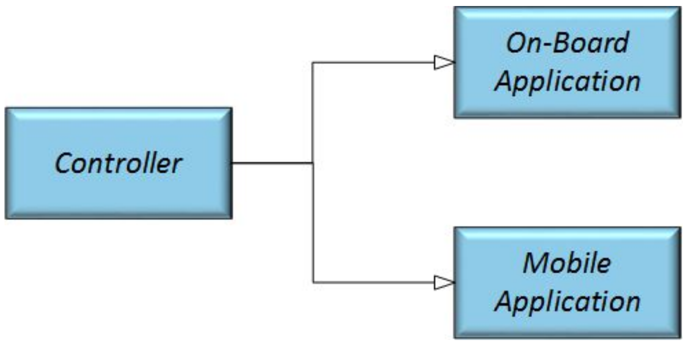


System Integration 2: Payment Manager

Finally, the ***Controller***, which handles the system's interaction with the interface, has to be integrated first with the main subsystems and then with the ***On-board Application*** and the ***Mobile Application*** of the operator and of the User.



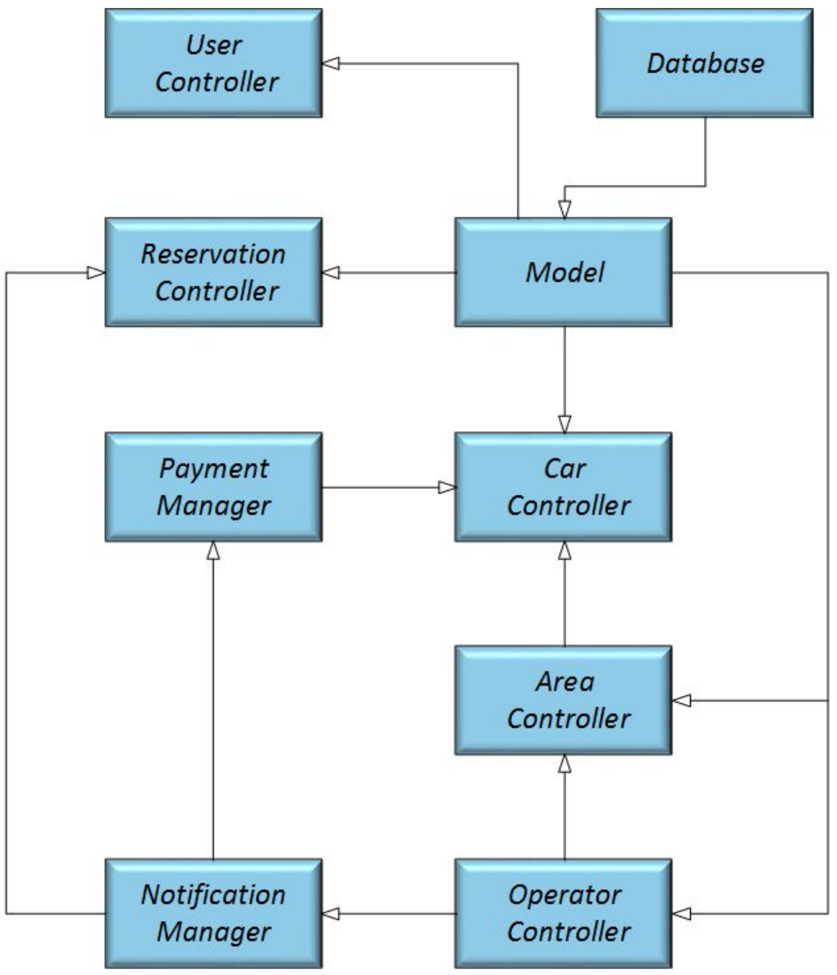
System Integration 3.1: Controller, Components



System Integration 3.2: Controller, Applications

2.4.2. Sub System Integration Sequence

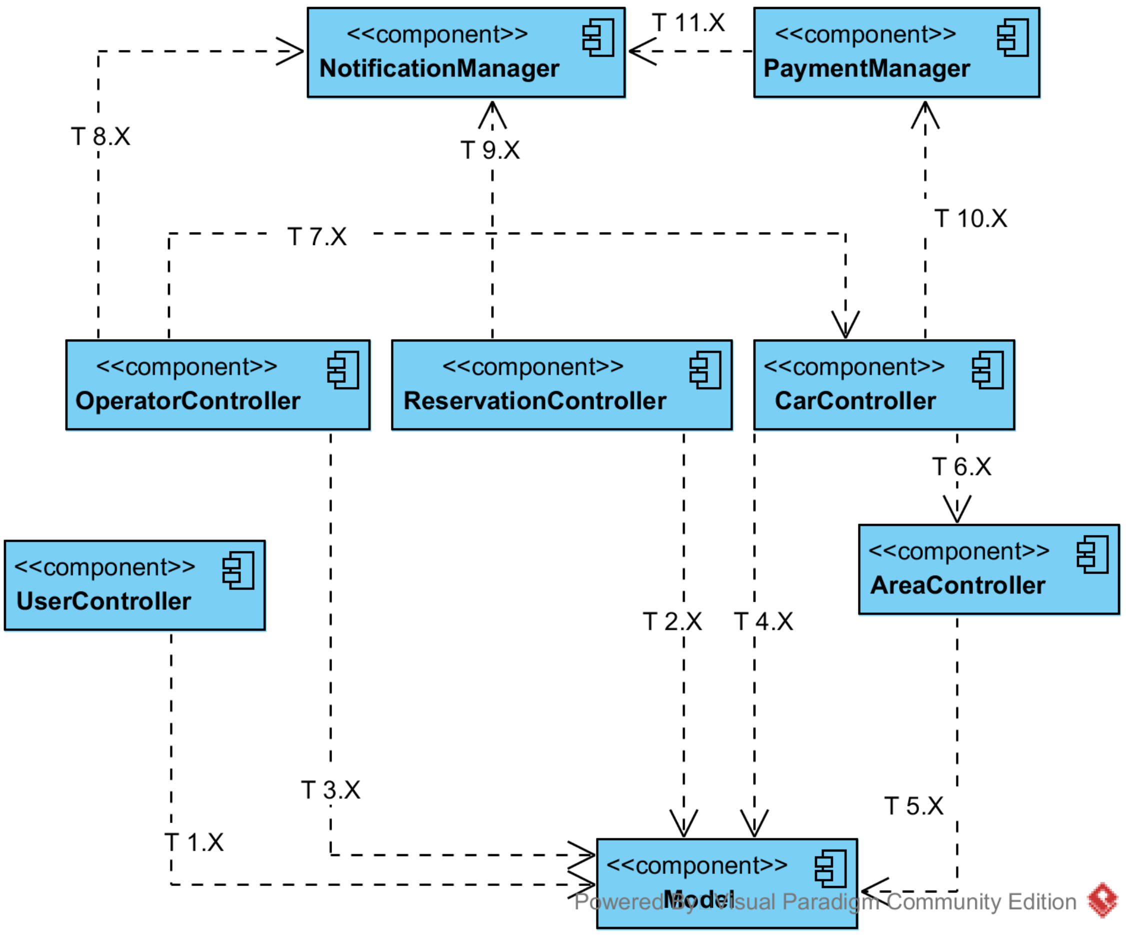
In the following chart we show, into a more general view, the sequence in which the main components are integrated, leaving aside the previously treated low-level components. In this representation we omit the client-side components that are simply connected to the various controllers through a router.



System Integration Sequence 1: Complete Integration of the System

3. Individual Steps and Test Description

In this section it is explained the description of every single step during the testing phase, that involves different high level modules. In the figure below it is specified all the test case in the component view, taken from the DD document.



<<component>> NotificationManager
T 11.X
<<component>> PaymentManager
T 8.X
T 9.X
T 10.X
T 7.X
<<component>> OperatorController
<<component>> ReservationController
<<component>> CarController
T 6.X
<<component>> UserController
<<component>> AreaController
T 2.X
T 4.X
T 3.X
T 5.X
T 1.X
<<component>> Model
Powered By Visual Paradigm Community Edition

Test 1 : Component View with Test description

Every Test is specified in the diagram above,and it is described in the following tables. Starting from the model and moving to higher level components, it is indicated all the methods to test and it is used the format *<Caller,Called>* .

3.1. T 1.X :User Controller, Model

T.1.1_ *checkLogin(username,password)*	
Input	***Output***
Null parameter	NullArgumentException
Wrong Username or password	InvalidCredentialException
Valid arguments	Return a session cookie and the user status is updated in the DB

T.1.2_ *getStatus(user)*	
Input	***Output***
Null parameter	NullArgumentException
The user does not exist	InvalidArgumentValuesException
Valid arguments and the user is logged	Return true
Valid arguments and the user is not logged	Return false

T.1.3_ *checkRegistration (username, password,name,surname, email,licence,creditcard)*	
Input	***Output***
Null parameter	NullArgumentException
Invalid arguments inserted or the User is already registered	InvalidArgumentValuesException
Valid arguments	The User is created and inserted in the

	DB with the following datas: credit card, name and surname, id, licence number, email address

T.1.4_ *updateUserStatus(userList)*	
Null parameter	NullArgumentException
Empty array or an array that contains null values/ non existing users	InvalidArgumentValuesException
Valid argument	The user status is updated in the DB

T.1.5_ *getUserList()*	
Input	***Output***
Nothing	Get the list of all the registered users.

3.2. T 2.X : Reservation Controller, Model

T.2.1_ *CreateReservation (user, car)*	
Input	***Output***
Null parameter	NullArgumentException
Exist another reservation associated to the same User in the DB	InvalidArgumentValuesException
Valid arguments	The reservation is created and inserted in the DB with the useful datas: user,car, reservation id

T.2.2_ *DeleteReservation (user)*	
Input	***Output***
Null parameter	NullArgumentException
Reservation expired	The reservation is fully deleted in the DB and the User is charged
User doesn’t exist	InvalidArgumentValuesException
Valid arguments	The reservation is fully deleted in the DB, the associated car is set available again

T.2.3_ *getReservationList()*	
Input	***Output***
Nothing	Get the list of all the active reservations.

T.2.4_ *UpdateReservationList (reservationList)*	
Input	***Output***
Null parameter	NullArgumentException
Valid arguments and some reservation expired	The completed and expired reservation are removed, and the the user associated to an expired reservation are charged
Empty array or an array that contains null values/ non existing reservations	InvalidArgumentValuesException
Valid arguments and no reservation expired	The completed reservation are deleted and the list is updated in the DB

3.3. T 3.X : Operator Controller, Model

T.3.1_ *checkOperatorLogin(operatorID,password)*	
Input	***Output***
Null parameter	NullArgumentException
Invalid id or password	InvalidArgumentValuesException
Valid argument	The operator status is set available in the DB

T.3.2_ *getOperatorStatus(operator)*	
Input	***Output***
Null parameter	NullArgumentException
Invalid operator id	InvalidArgumentValuesException
The operator is assigned to an emergency	Return false
The operator is available	Return true

T.3.3_ *updateStatus(operatorList)*	
Input	***Output***
Null parameter	NullArgumentException
Empty array or an array that contains	InvalidArgumentValuesException

null values/ non existing operators	
Correct parameters	The status of operators is updated in the DB

T.3.4_ *getOperatorList()*	
Input	***Output***
Nothing	Returns the list of all operators available in the DB.

3.4. T 4.X : Car Controller, Model

T.4.1_ *displayAvailableCars (location)*	
Input	***Output***
Null parameter	NullArgumentException
Invalid address or location	InvalidArgumentValuesException
No cars available in a certain radius distance from the selected area	EmptyAreaException
Valid parameters	The available cars in the inserted area are taken from the DB

T.4.2_ *updateLocation (listCars)*	
Input	***Output***
Null parameter	NullArgumentException
empty array, or array with null objects or non existing cars	InvalidArgumentValuesException
Valid parameters of the car	The position of the cars is updated in the DB.

T.4.3_ *getCarList()*	
Input	***Output***
Nothing	Return the list of all cars

3.5. T 5.X : Area Controller, Model

T.5.1_ *getAreaList()*	
Input	***Output***
Nothing	Return all the areas with cars

T.5.2_ *getSafeAreas()*	
Input	***Output***
Nothing	Return all the Safe Areas

T.5.3_ *getFreeParking(area)*	
Input	***Output***
Null parameter	NullArgumentException
Non existing area	InvalidArgumentValuesException
Correct parameter	Return the number of free parking in that area

3.6. T 6.X : Car Controller, Area Controller

T.6.1_ *getArea(car)*	
Input	***Output***
Null parameter	NullArgumentException
Not valid car parameter	InvalidArgumentValuesException
Correct parameter	The coordinates of the location area of the car is returned

T.6.2_ *checkMoneySavingMode(car)*	
Input	***Output***
Null parameter	NullArgumentException
Not valid car parameter	InvalidArgumentValuesException
The car is in an another area	Returns false
The final location of the car is in a money saving safe area	Returns true

T.6.3_ *checkOutsideArea (car)*	
Input	***Output***
Null parameter	NullArgumentException
The car is in an outside area	Returns false
The car is in a safe area	Returns true

3.7. T 7.X : Operator Controller, Car Controller

T.7.1_ *UpdateCarStatus (operator)*	
Input	***Output***
Null parameter	NullArgumentException
Non existing operators	InvalidArgumentValuesException
Valid arguments and no reservation expired	The emergency is completed, the car assigned to that operator is set available again.

3.8. T 8.1 : Notification Manager, Operator Controller

T.8.1_ notifyOperator(operator)	
Input	***Output***
Null parameter	NullArgumentException
No Existing parameter	InvalidArgumentValuesException
Correct parameter	Send a notification to the operator specified

3.9. T 9.1 :Reservation Controller, Notification Manager

T.9.1_ confirmReservation(user)	
Input	***Output***
Null parameter	NullArgumentException
No Existing parameter	InvalidArgumentValuesException
Correct parameter	Send a notification to the user to confirm the reservation

3.10. T 10.1 : Car Controller, Payment Manager

T.10.1_ *endRide (car)*	
Input	***Output***
Null parameter	NullArgumentException
Invalid car parameter	InvalidArgumentValuesException
Valid parameters of the cars and final charge calculated	The reservation is set as completed and the Payment Manager charges the User controlling the area where the ride ended.

3.11. T 11.1 : Payment Manager,Notification Manager

T.11.1_ notifyUser*(User, string)*	
Input	***Output***
Null parameter	NullArgumentException
Invalid car parameter	InvalidArgumentValuesException
Correct parameter	Send a notification to the user to confirm the payment, or to notify something went wrong

4. Tools and Test Equipment Required

4.1. Tools

In this section we are going to describe which tools we will use to test our “Power Enjoy” system, and provide an explanation of why and how we are going to handle them.
First, in order to test all the **functional requirements** described into the RASD document,we chose the following tools:

- **JUnit Framework** is perfect to verify each component of the system and partially their interaction, both through unit test. In fact, we can see if the values returned or exceptions are the expected ones. With JUnit we can test each methods of the written code, that allows us to find problems early but not to test component strictly depends to other that are not implemented yet.
- **Mockito Framework** allows to abstract the dependencies between each component and to check the interaction also with external service, or not yet implemented classes. This choice covers the non tested part of the code. The main reason to choose this framework is because provides the environment for interaction testing and it is very simple to use.

Instead, for the **non functional requirements** defined in the RASD document, in order to test performance of the system we are going to use the following tool:

- **Apache JMeter** is optimal to verify all the aspects of the system that are not directly related to its behavior. It can be used to simulate group of servers, networks or objects in order to perform test on: Database, TCP, and REST services. That allows us to define a Test plan through JMeter to see if the system match the real needs of the User.

Also the **Manual Testing** is not less important, in fact we can adopt this technique when the test needs to run only few times and not repetitively.

At the end we will integrate, also, the ***User acceptance Testing*** to effective validate the system and see if it matches the users' needs.

4.2. Test Equipment

As we fully described in the DD document, the "Power Enjoy" system is run on a native mobile application. In order to testing it, we have to verify all the process from the download to the execution of the application. We have also to check the interaction between the front-end and the back-end of our system.

In order to do that, it is essential to test on physical devices supporting the application. Due to the extremely huge quantity of different devices and the continued challenge of them, to cover the major number of Users we decide to use an **"hybrid approach"**.
First, we require an **emulated environment** to take advantage of the speed and mobile diversity, also if the accuracy of pixel is not perfect. This choice allow to test every single devices on the market, now and in the future, with an affordable cost.
Then, we mix it with **real devices** to validate that the application functioning as expected, so that the requirement matches the real needs of Users.

At the end, the back-end system testing requires, as we already specified in the Tools description, Apache JMeter. That it is used to test RESTful API to know how efficiently it works and how many concurrent users the server can handle. JMeter is also required to test the DB.

5. Program Stubs and Test Data Required

5.1. Program Stubs and Drivers

In the bottom-up approach that we are going to adopt for the "Power Enjoy" Testing strategy, the submodules strictly depend on the main module as shown below.

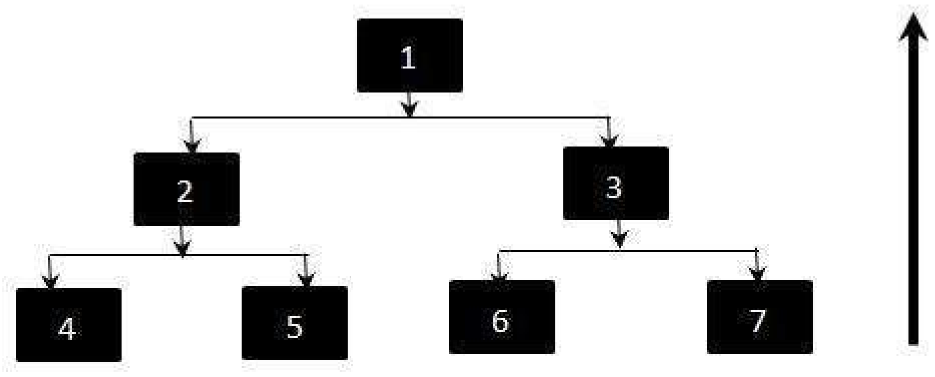


Figure 1: Bottom-up approach

Due to these dependency is extremely important to develop **drivers** that temporary simulate missing pieces in our system. The driver allows to invoke the unit to be tested, so we can conduct JUnit test until the final module is integrated because it permits to get the "inputs" needed for testing.
We are going to show below the dependencies existing between the different system components, and also what are the drivers needed.

Drivers	Test Involved	Description
Reservation Controller Driver	T.2.X	It will invoke methods in the model component
Operator Controller Driver	T.3.X	
Area Controller Driver	T.5.X T.6.X	it will invoke methods of the model and test the interaction with the Car Controller component
Car Controller	T.4.X T.10.1	It will invoke methods of the model component and also of the Payment Manager component
Notification Manager Driver	T.10.1 T.11.1 T.8.1	It is needed to simulate the interaction with the following components: Payment Manager, Reservation Controller, and Operator Controller

5.2. Test Data

The amount of data necessary to the integration testing phase should be suitable to perform the test described in this document.
In particular, the data that we need are the following:

Data	Boundary Case	Tested Component
User information (username, password, licence number, email, credit card..)	● Null Object ● Null fields ● Invalid Licence number ● Invalid email ● Invalid credit card ● Invalid Login ● Invalid password	User Controller
Operator Information (ID, password)	● Null Object ● Null fields ● Invalid ID ● Invalid password	Operator Controller
Reservation Request	● Null Object ● Null fields ● Invalid Location ● No cars in that Area	Reservation Controller Notification Manager
Area Information (about GPS location)	● Null Object ● Null fields ● Area outside the allowed zones	Car Controller Area Controller Operator Controller Notification Manager
Payment Information (at the end of a ride)	● Null Object ● Null fields ● Not valid payment method ● Not valid final location	Car Controller Operator Controller Area Controller Payment Manager Notification Manager

6. Effort Spent

6.1 Giulia Leonardi

- 27/12/16: 2h
- 28/12/16: 2h
- 29/12/16: 4h
- 11/01/17: 3h
- 12/01/17: 2h
- 13/01/17: 2h
- 13/01/17: 3h

6.2 Marzia Degiorgi

- 27/12/16: 2h
- 29/12/16: 2h
- 30/12/16: 1h
- 06 /01/17:3h
- 07/01/17: 3h
- 10/01/17: 3h
- 11/01/17: 3h
- 12/01/17: 2h

6.3 Valentina Ionata

- 29 /12/16: 2h
- 31/12/16:1h
- 5/01/17: 4h
- 10/01/17: 2h
- 11/01/17: 2h
- 12/01/17: 4h
- 13/01/17: 2h
- 14/01/17: 2h