POLITECNICO
MILANO 1863

“PowerEnjoy”

Integration Test Plan Document

Version 1.0 *(15/01/2017)*

Giorgio Marzorati (876546)
Aniel Rossi (877018)
Andrea Vaghi (877710)

INDEX OF CONTENTS

INTRODUCTION

Purpose and scope

This document (Integration Testing Plan Document) describes the entire process of integration and testing of all the components in which our system can be decomposed. The principal purpose is to test the functionality of the interaction and functionality between them, and for that reason every team member who cooperates in the integration and testing should read this document. This phase plays a key role in the development process, because it guarantees that each component interoperate consistently, fulfil the requirements and don't exhibits unexpected behaviours. To make this in a clear way, we will show all the subsystems that must be tested and the subcomponent in which they can be decomposed. Also we will expose the criteria that the status of the project must satisfy before the beginning of integration testing.

In this document we will describe:

- An overall description of the testing approach used;
- A sequence of integration of the components;
- A description (specifying input and expected output data) for each test activity;
- Performance measures;
- The list of the tools used in testing activities;

Glossary

- SUBCOMPONENT: each low level component that realizes one or more functionality of a subsystem;
- SUBSYSTEM: high-level functional unit of our system that must provide some functionalities in order to make operative our entire project;
- DD: Design Document;
- DBMS: Database Management System;
- RASD: Requirement Analysis and Specification Document;
- GPS: Global Positioning System.

Reference documents

- Assignments AA 2016-2017
- Requirement analysis and specification document : RASD.pdf
- Design Document: DD.pdf
- Integration Testing example documents: Integration testing example document.pdf and Sample Integration Test Plan Document.pdf

INTEGRATION STRATEGY

Entry criteria

In this section we will present the required condition about the progress of the project in order to have meaningful results from the integration testing.

First of all, the documents concerning the requirement analysis (RASD) and the architecture design (DD) must be fully written. In this way we have a panoramic visual on our system and in particular on the component diagram, that is a core element in this phase.

The integration process has some constraint on the percentage of completion on the main components. The percentages we are going to show are not a lower bound in order to consider a component ready for the integration test process but it refer to the status of the entire project. These percentages are calculated also based on our sequence of integration in order to begin this phase with almost fully developed components.

- 100% Data System subsystem
- 90% or greater for Car Management System subsystem
- 70% or greater for Account Manager subsystem
- 50% or greater for Client Mobile Application

Elements to be integrated

This chapter is dedicated to the description of all components that we need to integrate together and test.

As it could be seen in the DD, we can think that our system is made of many high-level components interacting with each other and with external ones. Each one of these high-level components, called *subsystem*, implements specific functionalities that our application need to offer. Like we have already mentioned, one subcomponent is composed by subcomponents. A subcomponent is on an higher level of abstraction respect to a subcomponent. From this, during the integration phase, we will consider the two different levels of abstraction.

We will show further in the document that we will adopt a bottom-up strategy of integration. This means that we will concentrate first on low-level components, and then building the higher level ones.

There are also components that are not part of a real subsystem because their functionality is independent from other components. In our case these components work as “interfaces” for already existing commercial components, like for positioning or payment functionalities.

Components on the Server-Side:

1. Account Management System (subsystem)
    a. Registration Manager
    b. Login Manager
2. Car Management System (subsystem)
    a. Reservation Manager

    b. Identification Manager
    c. Ride Manager
    d. Car Manager
3. Data System (subsystem)
    a. DBManager
    b. Data Model
4. Notification Manager (component)
5. Position Manager (component)
6. Payment Manager (component)
7. Assistance Manager (component)

On the Client - Side there is only the Mobile Application.

On the Car there is an embedded software developed by an external company.

Integration testing strategy

The strategy adopted in this phase is a mixture of bottom-up and critical-module-first approaches. This is because in some steps of the integration, using a bottom – up approach, we have subcomponents or subsystems that are not meant to be integrated one before the other. In these cases we decide which one to integrate first reasoning about their level of criticality and then considering how much this component impacts on our system. This is because we can concentrate first on components that, in case of problems, could be a critical danger for the entire system and we have to be sure of their behaviour before going on with the process. The main reasons of the choice of the bottom-up strategy is that:

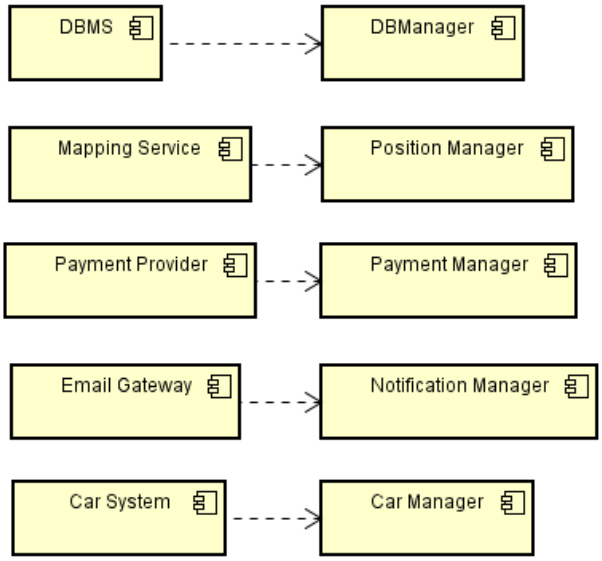
- we have in our system some important components which role is to manage the interaction with commercial and already built components;
- we are able to test always on an already tested part of our software and work on already semi-constructed structures;

Sequence of component/function integration

Software integration sequence

The sequence of integration of the components of the systems is based, as already mentioned, on a bottom-up approach. The focus at the beginning is on the components that interact with other components or systems that are taken from the commercial world and that are ready to use. In particular:

- DBManager -> interacts with the DBMS
- PositionManager -> interacts with Mapping Service
- PaymentManager -> interacts with Payment Provider
- NotificationManager -> interacts with Email-Gateway
- CarManager -> interacts with the Car System present on the physical car



Explaination: A → B

A is needed for the integration of B

In this case A are all commercial components already working

Once these components are tested and we are sure that interact the right way, we can proceed to integrate the other components that rely on these just mentioned.

Data System integration:

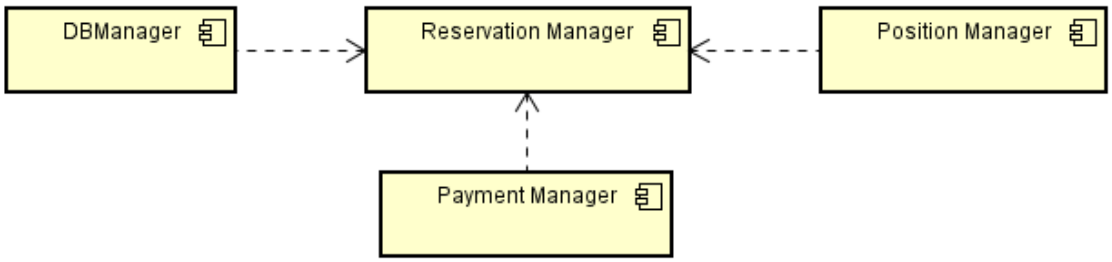
First we begin integrating the subcomponents of the Data System subsystem because it provides the access and the communication with the Database, which plays a fundamental role for all the components of the system.

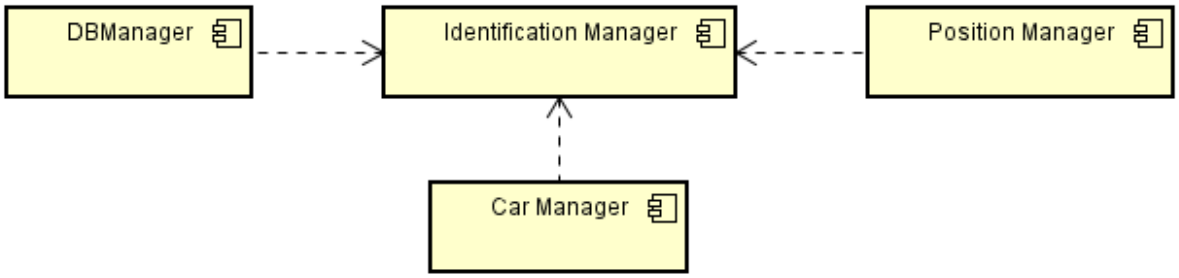
Car Management System integration:

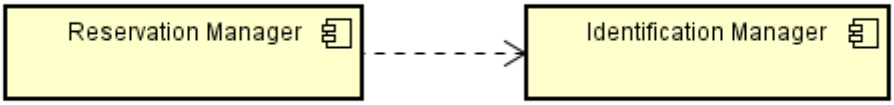
The second step in the integration process is to appropriately connect the subcomponents implementing the Car Management System. This choice comes from the critical-module-first approach, because Car management is the most important functionality of PowerEnjoy.

The first component we decide to integrate is the Reservation Manager, because it relies only on already integrated components.

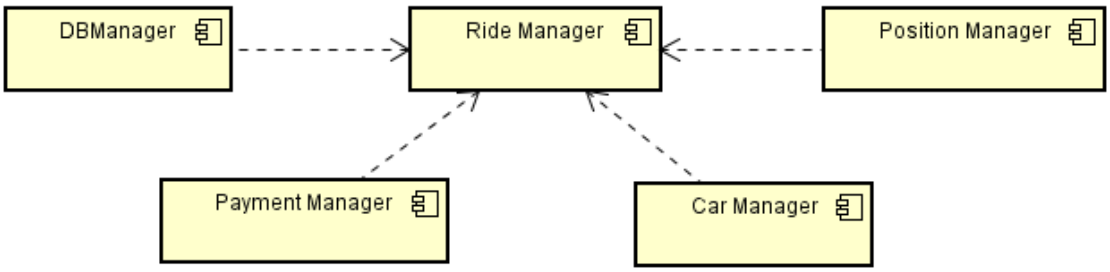
The second step is to integrate Identification Manager. It relies not only on already configured components, but also on Reservation Manager and Car Manager.



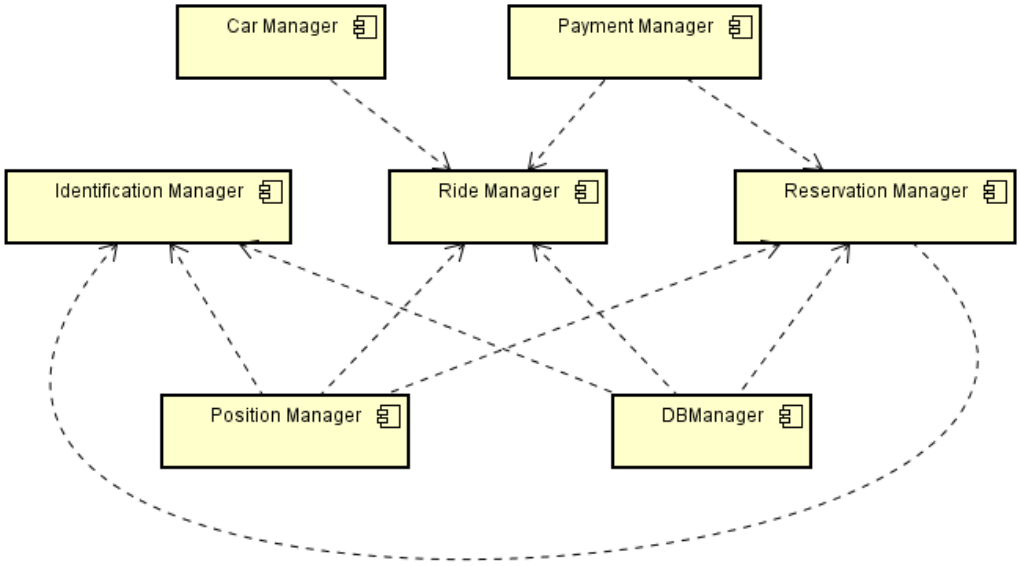




The last component to be integrated is the Ride Manager.



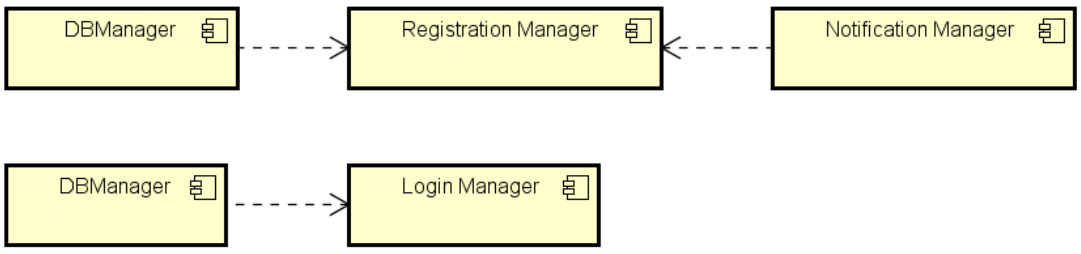
Overview of the Car Management System dependencies of integration:



Account Management integration:

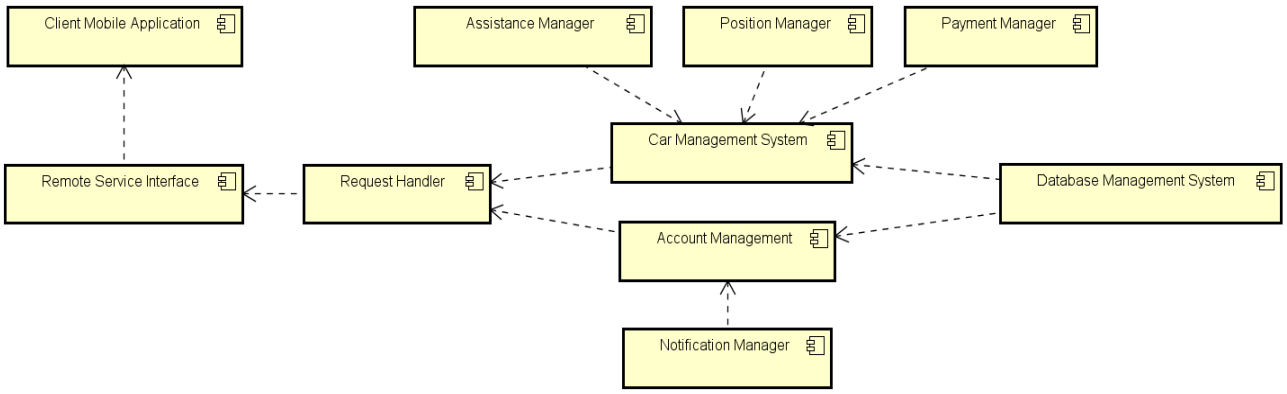
It should be noted that, like the Car Management System, also the subcomponents of Account Management are loosely coupled together as they cover different operations that can be performed on accounts. So we can integrate them without any particular functional constraint and we decide to integrate in order:

- Registration Manager
- Login Manager



Subsystems integration sequence overview

In the following diagram we provide a general overview of how the various high-level subsystems and components are integrated together to create the full PowerEnjoy infrastructure.



INDIVIDUAL STEPS AND TEST DESCRIPTION

In this section, we will provide a detailed description of the test to be done on each pair of components we mentioned in the chapter "Elements to be integrated". In each description we will provide first the *caller* component, than the *called* component, on which the *caller* can invoke the methods.

Reservation Manager

Reservation Manager -> DB Manager

insertReservation(reservation)	
Input parameters	***Effects***
Null parameter(s)	NullArgumentException thrown
Reservation with an ID already existing in the database	InvalidArgumentValueException thrown
Valid parameter	A Reservation data is created with the current timestamp, the driving license of the user who wants to perform the reservation and the license plate of the car to be reserved
makeReservationExpired(reservationId)	
Input parameters	***Effects***
Null parameter	NullArgumentException thrown
Reservation Id not in the present in the database	InvalidArgumentValueException thrown
Valid parameter	The reservation data is set as expired and it is no more valid.
createFee(drivingLicense, reservationId)	
Input parameters	***Effects***
Null parameter(s)	NullArgumentException thrown
drivingLicense not in the present in the database	InvalidArgumentValueException thrown
Reservation Id not in the present in the database	InvalidArgumentValueException thrown
Valid parameters	If the reservation is expired (one hour expiration) a fee is added to the Database.
getAvailableCars(userPosition)	
Input parameters	***Effects***
Null parameter	NullArgumentException thrown
Valid parameter	Returns the list of all available cars depending from the clientPosition

Reservation Manager -> Position Manager

calculateDistance(userPosition, carPosition)	
Input parameters	***Effects***
Null parameter(s)	NullArgumentException thrown
Valid parameters	Returns the distance between two positions

Reservation Manager -> Payment Manager

sendFee(user)	
Input parameters	***Effects***
Null parameter	NullArgumentException thrown
Valid parameters	A fee is sent to the user through his payment method

Identification Manager

Identification Manager -> DB Manager

checkUser(user)	
Input parameters	***Effects***
Null parameter	NullArgumentException thrown
Invalid PIN for the email	InvalidArgumentValueException thrown
Invalid email for PIN	InvalidArgumentValueException thrown
Valid parameters (email, PIN)	The client can make an identification
checkExistingAndAvailableCar(licensePlate)	
Input parameters	***Effects***
Null parameter	NullArgumentException thrown
Invalid license plate	InvalidArgumentValueException thrown
Valid parameter	The car can be identified
identifyCar(userId, licensePlate)	
Input parameters	***Effects***
Null parameter(s)	NullArgumentException thrown
User Id not present in the database	InvalidArgumentValueException thrown
License plate not present in the database	InvalidArgumentValueException thrown
Valid parameters	The user is associated to the car and the car is set as unavailable in the Database

Identification Manager -> Car Manager

unlockCar(licensePlate)	
Input parameters	***Effects***
Null parameter	NullArgumentException thrown
License Plate not existent or not valid	InvalidArgumentValueException thrown
Valid parameter	The Car Manager sends a request to the Car System to unlock the car

Identification Manager -> Reservation Manager

disactiveReservation(reservationId)	
Input parameters	***Effects***
Null parameter	NullArgumentException thrown
Reservation Id not existent	InvalidArgumentValueException thrown
Valid parameters	The reservation is set no longer active

Ride Manager

Ride Manager -> DB Manager

createRide(ride)	
Input parameters	***Effects***
Null parameter	NullArguentException thrown
Ride with an ID already existing in the database	InvalidArgumentValueException thrown
Valid parameter	A Ride data is created with the current timestamp, the driving license of the user who wants to perform the reservation and the license plate of the car to be reserved
finishRide(ride)	
Input parameters	***Effects***
Null parameter	NullArguentException thrown
Invalid ride	InvalidArgumentValueException thrown
Valid parameter	The Ride data is modified adding the final timestamp and the final position
checkFinalPosition(currentPosition)	
Input parameters	***Effects***
Null parameter	NullArgumentException thrown

Valid parameter	The car's current position is checked among the list of Safe Areas
getSafeAreas()	
Input parameters	***Effects***
No parameters	Returns the list of safe areas stored in the database
saveSensorsValues(licensePlate,sensorData)	
Input parameters	***Effects***
Null parameter(s)	NullArgumentException thrown
License Plate not present	InvalidArgumentValueException thrown
Valid parameters	Stores the values of the sensors of a car in the Database

Ride Manager -> Payment Manager

sendPayment(user, ride)	
Input parameters	***Effects***
Null parameter(s)	NullArguentException thrown
Invalid user	InvalidArgumentValueException thrown
Invalid ride	InvalidArgumentValueException thrown
Valid parameters	The user is charged for the selected ride

Car Manager -> Ride Manager

sendSensorsValues(licensePlate,sensorData)	
Input parameters	***Effects***
Null parameter(s)	NullArgumentException thrown
Valid parameters	Sends sensor values to the Ride Manager

Car Manager -> Assistance Manager

sendAssistanceRequest(assistanceRequest)	
Input parameters	***Effects***
Null parameter(s)	NullArgumentException thrown
Valid parameters	Sends an Assistance Request to the Assistance Manager with the problem details

Registration Manager

Registration Manager ->DB Manager

registerUser(user)	
Input parameters	***Effects***
Null parameter	NullArgumentException thrown
Existing user (email or drivingLicense)	InvalidArgumentValueException thrown
Valid parameter	The account is registered in the Database

Registration Manager -> Notification Manager

emailToUser(user)	
Input parameters	***Effects***
Null parameter	NullArgumentException thrown
Valid user	The user (already inserted in the Database) is notified with his personal PIN

Login Manager

Login Manager -> DB Manager

login(user)	
Input parameters	***Effects***
Null parameter	NullArguentException thrown
Invalid PIN for the email	InvalidArgumentValueException thrown
Invalid email for PIN	InvalidArgumentValueException thrown
Valid parameters (email, PIN)	The user is logged in the system and can use it

Assistance Manager

Assistance Manager -> DB Manager

addAssistanceRequest (assistanceRequest)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Valid parameters	Adds to the Database the assistance request
setRequestSolved (assistanceRequest)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Valid parameters	It modifies the assistanceRequest data setting the Boolean solved to true

General interactions with Position Manager

getUserPosition(idDrivingLicense)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid drivingLicense	InvalidArgumentValueException thrown
Valid parameters	It returns the position of the specified user
getCarPosition(licensePlate)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid licensePlate	InvalidArgumentValueException thrown
Valid parameters	It returns the position of the specified car

General interactions with DBManager

These are all general methods that DBManager offers to the components that send their requests.

getUser(idDrivingLicense)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Driving license not existent	InvalidArgumentValueException thrown
Valid parameters	It returns the user entity with the specified driving license id
getUser(PIN)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid pin	InvalidArgumentValueException thrown
Valid parameters	It returns a user data from with the specified pin
getReservation(user)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid user	InvalidArgumentValueException thrown
Valid parameters	It returns reservation data if active of the specified user
getReservation(licensePlate)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid licensePlate	InvalidArgumentValueException thrown
Valid parameters	It returns reservation data if active of the specified licensePlate
getReservation(car)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid car	InvalidArgumentValueException thrown
Valid parameters	It returns reservation of the specified car

getRide(licensePlate)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid licensePlate	InvalidArgumentValueException thrown
Valid parameters	It returns ride of the specified licensePlate
getRide(drivingLicense)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid car	InvalidArgumentValueException thrown
Valid parameter	It returns ride of the specified drivingLicense
getPayment(idRide)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid idRide	InvalidArgumentValueException thrown
Valid parameters	It returns the paymentdata of the ride connected to the specified idRide
getPayment(drivingLicense)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid drivingLicense	InvalidArgumentValueException thrown
Valid parameters	It returns all payments data of all the rides connected to the specified drivingLicense
getPaymentCharge(payment)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid payment	InvalidArgumentValueException thrown
Valid parameters	It returns the charge data of the payment specified
addPayment(payment)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid payment	InvalidArgumentValueException thrown
Valid parameters	It adds to the Database the payment data

getCar(licensePlate)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid licensePlate	InvalidArgumentValueException thrown
Valid parameters	It returns the car data
getAssistanceRequest(licensePlate)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid licensePlate	InvalidArgumentValueException thrown
Valid parameters	It returns the assistance request data
getCarIdentification(idDrivingLicense)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid drivingLicense	InvalidArgumentValueException thrown
Valid parameters	It returns the car identification data
GetCarIdentification(licensePlate)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid licensePlate	InvalidArgumentValueException thrown
Valid parameters	It returns the car identification data
getSafeArea(idSafeArea)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid idSafeArea	InvalidArgumentValueException thrown
Valid parameters	It returns the safe area data
getSafeArea(position)	
Input parameters	***Effetcs***
Null parameter	NullArguentException thrown
Invalid position	InvalidArgumentValueException thrown
Valid parameters	It returns the safe area data by position

PERFORMANCE ANALYSIS

Since there are some components that can be tested in isolation, we can perform some preliminary system's performances measurements before the full analysis of the infrastructure, which will be executed during the system integration phase.

We can first assure that the client application will run properly for all the mobile platforms, which means it will use a reasonable usage of CPU and RAM. We consider that the minimum smartphone requirements in order to run the application properly are the following:

- Since the system is structured in order to achieve a thin-client architecture, we suppose that the mobile application should occupy between 5-10 Mb of disk space just for the app installation. In order to run correctly and considering the trends in the display resolution of newest devices, a 50 Mb of free space is required (for a correct representation of all images)
- The average RAM usage must be less than 128 Mb (obviously we can't control how many application the client will run concurrently, so we assume a reasonable amount of memory)
- The application must run correctly on devices with single core processor (we can consider 1 GHz clocks as lower bound) and more powerful

As illustrated in the RASD, there are also some constraints on smartphones hardware equipment:

- GPS sensor
- Internet Connectivity

This data could be reconsidered during the actual development phase, considering more technical constraints and improvements in devices technology occurring meanwhile. The actual performances measurements will be performed with dedicated software tools, which will be illustrated in the dedicated section of this document.

TOOLS AND TEST EQUIPMENT REQUIRED

Tools

In order to have more precise results from testing various components we are going to use different tools concerning business logic components running in Java Enterprise Edition runtime environment, starting from Arquillian and JUnit.

Arquillian integration testing framework enables us to test the correctness of the interactions between a component and its surrounding execution environment, in particular in our case it helps us to test if the connections with Database are properly managed. This tool is also really useful to test all the relations between components that interact with our Java application server.

For what concern unit testing activities, our choice for the dedicated tool is JUnit. We will use it to test if the interaction between strictly related components are producing expected results (for example, if the correct objects are returned after the invocation of a method, if appropriate exceptions are raised when invalid arguments are passed and so on).

In order to achieve the result illustrated before, we will focus in the manual creation of specific test units that will cover the most important aspect of the application, monitoring the corresponding behaviour in the different testing cases. Also, the test process will imply considerable effort in manually arranging the system data in order to recreate specific situations.

As explained in the previous chapter of this document, we are also going to control the actual hardware resources requested by the mobile application in order to work correctly on all devices and platform. To do so, we are going to use some performances analysis tool:

- Since we will develop the app as an Hybrid Application using the framework Cordova, which uses web technologies such as Javascript, HTML5 and CSS3 for the actual development, we will operate a first analysis during the development with the different browser's developer tools (Chrome for Android and Safari for iOS) and with Visual Studio for Windows Phone
- Once the application will be developed and deployed, its hardware performances (CPU, RAM usage) will be tested with the specific tool depending of the specific platform: the Android Monitor of Android Studio, the XCode IDE or the Windows Performance Analyzer toolkit of the Windows ADK

Test equipment

In this section we are going to illustrate what kind of equipment we need to perform reasonable tests able to better simulate the actual environment of the deployed system.

For what concerns the client side of the architecture here is the list of devices we are going to use to test the mobile application:

- At least one Android device (smartphone or tablet) for each resolution size:

Phones:	Tablets:
320 x 480	1024 x 600
480 x 800	1280 x 800
480 x 854	1920 x 1080
540 x 960	2048 x 1536
1280 x 720	2560 x 1440
1920 x 1080	2560 x 1600

- At least one iOS device for each iOS product family (smartphones & tablets) (except the iPhone 3 & 3s)

- At least one Windows Phone smartphone for each resolution size:

  800 x 480
  1280 x 768
  1280 x 720
  1920 x 1080

This list of devices can be of course updated in case of new products coming in the market from the moment of this document writing.
Regarding the backend section, our considerations lead us to think that all the components concerning the business logic should be deployed and tested on an already configured cloud infrastructure.

Specifically, the testing infrastructure must be almost the same to the one that will host our application; in particular it must run the same operating system, the same Java Enterprise Application Server, interfaces and the same DBMS.

We assume to use the Red Hat OpenShift cloud infrastructure, but the actual final implementation could be changed according to future considerations.

REQUIRED PROGRAM STUB AND TEST DATA

Program stubs and drivers

Since we will adopt a bottom-up approach to perform the component integration and testing, we will need some drivers in order to perform the invocation of necessary methods on component that we need to test. As can be easily extrapolated this part will be performed using the Junit framework.

Here we list all the drivers that need to be developed as part of the integration testing phase:

- **DBManager Driver**: this module will invoke the methods exposed by DBManager in order to test the interaction with DBMS.
- **Position Manager Driver**: this module will invoke the methods exposed by Position Manager in order to test the interaction with Mapping Service.
- **Payment Manager Driver**: this module will invoke the methods exposed by Payment Manager in order to test the interaction with External Payment Provider.
- **Reservation Manager Driver**: this module will invoke the methods exposed by Reservation Manager in order to test the interaction with DBManager, Position Manager and Payment Manager.
- **Identification Manager Driver**: this module will invoke the methods exposed by Identification Manager in order to test the interaction with DBManager, Position Manager.
- **Ride Manager Driver**: this module will invoke the methods exposed by Ride Manager in order to test the interaction with DBManager, Payment Manager and Position Manager.
- **Login Manager Driver**: this module will invoke the methods exposed by Login Manager in order to test the interaction with DBManager.
- **Registration Manager Driver**: this module will invoke the methods exposed by Registration Manager in order to test the interaction with DBManager and Notification Manager.
- **Request Handler Driver:** this module will invoke the methods exposed by Request Handler in order to test the interaction with Car Manager System and Account Manager subsystems.

Due to the use of a bottom up approach, we don't need stubs. The reason is because in all the cases we use drivers to test the interaction with already integrated components. The only stub that we use is a stub that plays the role of the Client, since it is the last element to be integrated and during all the sequence we want to test also the validity of the information that are sent him from the system.

Test data

Since we are going to perform a complete battery of tests, in this section we expose a set for each Driver specified in the previous chapter. So we need to have:

- A set of valid and invalid Reservations in order to test properly the Reservation Manager component. The set should also contain instances with some problems like :
  - Null object
  - Null fields
  - idDrivingLicense not conform with legal format
  - licensePlate not conform with legal format
  - Non valid fields
- A set of valid and invalid Rides in order to test properly the Ride Manager component. The set should also contain instances with some problems like :
  - Null object
  - Null fields
  - idDrivingLicense not conform with legal format
  - licensePlate not conform with legal format
  - Non valid fields
- A set of valid and invalid Users and Cars in order to test properly the IdentificationManager component. The set should also contain instances with some problems like :
  - Null object
  - Null fields
  - idDrivingLicense not conform with legal format
  - licensePlate not conform with legal format
  - Non valid fields in both tables
- A set of valid and invalid Users in order to test properly the Registration Manager component. The set should also contain instances with some problems like :
  - Null object
  - Null fields
  - idDrivingLicense not conform with legal format
  - Non valid fields
- A set of valid and invalid Users in order to test properly the Login Manager component. The set should also contain instances with some problems like :
  - Null object
  - Null fields
  - idDrivingLicense not conform with legal format
  - Non valid fields
- A set of valid and invalid Assistance Requests in order to test properly the Assistance Manager component. The set should also contain instances with some problems like :
  - Null object
  - Null fields
  - licensePlate not conform with legal format
  - Non valid fields

EFFORT SPENT

Giorgio Marzorati

03/01/2017 - 2h
04/01/2017 - 1h
06/01/2017 - 2h
07/01/2017 - 3h
09/01/2017 - 2h
10/01/2017 - 1h
11/01/2017 - 2h

Aniel Rossi

27/12/2017 - 2h
29/12/2017 - 2h
03/01/2017 - 2h
04/01/2017 - 1h
06/01/2017 - 2h
07/01/2017 - 3h
09/01/2017 - 2h
10/01/2017 - 1h
11/01/2017 - 2h
13/01/2017 - 3h
14/01/2017 - 4h
15/01/2017 - 7h

Andrea Vaghi

23/12/2016 - 2h
06/01/2017 - 1h
11/01/2017 - 2h
13/01/2017 - 3h
14/01/2017 - 4h
15/01/2017 - 7h

CHANGELOG

V1.0 - First release