ITPD
Integration Test Plan Document

Luca Marzi
Valeria Mazzola
Federico Nigro

January 16, 2017

Contents

1 Introduction

1.1 Hours of work

- Luca Marzi: 10h
- Valeria Mazzola: 10h
- Federico Nigro: 10h

1.2 Purpose and Scope

This document is the Integration Testing Plan for the PoweEnJoy system.

The term "PowerEnJoy system" refers to the set of all the software artifacts (developed or bought) that cooperates in order to provide all the functionalities which are specified in the RASD document describing the PowerEnJoy services.

The purpose of this document is to outline, in a clear and comprehensive way, the main aspects concerning the organization of the integration testing activity for all the components that make up the system.

In the following section we are going to provide:

- An exhaustive description of the integration process that we intend to follow with respect to the architecture that has been proposed in previous documents.
- A description of the integration testing approach and the rationale behind it.
- A description of the test cases and test suites that we intend to perform on the main interfaces and modules interactions.
- A description of the main perfomance evaluation that we want to perform against the system.
- A list of tools that we are going to use to perform the whole Verification & Validation analysis.

It's important to remark the fact that the Integration Testing Plan is intimately connected with the Build Plan (or Integration Plan): the effective development of components and interfaces will be guided mainly by the criteria defined for the Integration Testing.

At the end, the idea is to follow the "develop and test" approach, applied not only to software units but also to entire subsystems.

1.3 List of Reference Document

- RASD Document
- DD Document

- RASD: Requirements Analysis and Specification Document
- GPS: Global Positioning System
- SDK: Software Development Kit
- DBMS: Database Management System

2 Integration Strategy

This document aims to describe a consistent strategy should allow us to obtain an overall suboptimal quality with respect to all the functional and non functional requirements that have been specified in previous documents.

For this reason, both the RASD and the DD documents must be completed before proceeding with the integration phase that is eventually described in this document; the idea is that all the major aspects about the software architecture and the componet design, as weel as all the major functionalities, shall be available for a further analysis from the point of view of the integration that has to be made in order to obtain the complete system behind the PowerEnJoy services.

All the important conditions that have to be satisfied in order for this document to be applicable are explained below.

2.1 Entry Criteria

Before the beginning of the Integration Testing some criteria have to be satisfied:

- Every logical and functional component of the System is well defined and well connected with each other. In particular, the interactions among entities presented in the Architecture Schema and in the Sequence Diagrams are in their final version, as well as correctly implemented through the specify tools (see the Design Document).
- The edges of the System with respect to the remaining part of the Environment are well defined: there is nothing more to add to the System in terms of functionalities and there is nothing less to subtract to it (see requirements and goals in the RASD).
- Every interface, among entities and regarding the interaction with the User, is defined and in a util version for the next test analysis. In other words, it is not yet necessary a final implementation in this case, but it has to be clear how the transictions works and what precisely the User can receive and send via the particular interface (for instance the case of the Mobile App and the On Board System).
- In addition to the inputs to give to the System by the Integration Test, it is not less important to have a clear view of what every functional part does within the System (with a non final, but still well detailed documentation for each important component of the functional part object of study).

- Referring to the point above, the White Box Test has to be done for all the functional parts already implemented and analized in the Design Document and in the RASD previously. This means that, referring to the software components of the System described in the Design Document especially, all the portions of code regarding the UserController, the PaymentManager, the AdministrationHelper and the CarController were correctly and rationally deconstructed and eventually rewritten in some of their parts, for avoiding unreachable lines of code and typical conceptual errors. The BusinessManager is analyzed in the same way, paying a particular attention to the single links with the other components inside the whole schema (in general testing the links among logic units is important for their influences to themselves).

2.2 Elements to be integrated

In this document and specificly in this paragraph, we long to inspect a set of elements which is very useful for the test analysis. The elements covered in the chapter, are considered as some units, designed to guide the different cases of test. The connections of the single element with each other, as well as, its further decomposition, will be showed up later. Here there is a quick list of all the outlined elements:

- User Component
- "First Level Check Unit" (Referred to the Mobile App)
- "Second Level Check Unit" (Referred to the Server Java)
- Database Component
- Data Manager
- Database Manager
- Autentication/Registration Decision Manager
- Notification Manager
- Switching Menu Manager
- Profile Management Manager
- GPS Service Manager
- Unlocking Car Manager
- Locking Car Manager
- Car System Component
- Cancel Reservation Manager

- Fees Manager
- Payment Manager
- Payment System
- Assistance Manager
- Assistance Component
- Payment Instance Manager
- Exception Manager
- Log Manager
- Communication Manager
- Operator/Administrator Component

2.3 Integration Testing Strategy

There are several possibile Integration Testing Strategies in the literature; our purpose is to choose to one that better fits with the Build Plan that emerges from the designes choices described in the DD.

Here is a brief recap about the most populare Integration Testing Strategies described in the literature with some considerations about their applicability in the context of this project.

- Top-down: the top-down integration stategy starts from the highest level of the archicture and proceeds with the integration testing of lower level (with respect to the "use" or "include" relations) components and subsystems. Typically, this approach requires the development of good quality stubs to emulate low level components. With respect to the context of the core our project, this approach doesn't offer any particular advantage: the core of our system is composed by highly-customized low level software components (such as the ones that are deployed into the physical cars) that would be too much difficult to emulate throgh good quality stubs.

- Bottom-up: the bottom-up integration stategy starts from the lowest level of the architecture and proceeds towards the top of it. Typically this approach requires the developement of good quality drivers for those subcomponts whose integration has to be tested. With respect to the contet of the core of our project, this approach seems to be the best one: it allows us to focus on those components that are at the core of the architecture and at its lowest levels.

- Thread: the thread integration strategy consists of testing several modules (or part of them) that offer a user-visible subset of functionalities. With respect to the context of our project, this strategy seems suitable

to iteratively provide an increasing number of functionality to the driver that emulates the user interaction.

- Critical modules: this strategy consists of testing the riskiest modules first (provided a risk function). With respect to the context of our project, we have already mentioned the idea of "core" subsystems that lie at the bottom of the system's architecture. This strategy may become useful when evaluating the next low-level interaction that has to be considered for testing purposes.

Given all the considerations above, here are the guide lines for the Integration Testing Strategy for the project: we decide to follow the Bottom-up strategy as Integration Testing Strategy of the entire system with risk evaluation of all those low level critical compoments that are highlighted eventually in this document.

Although some combination of the Bottom-up approach with the Top-down approach might be useful in terms of early testing of some of the some user functionalities (such as the GUI), we believe that such a combination would result in a too much difficult integration plan.

We believe that the best approach for integrating and testing consists of the developement and testing of a solid API on the top of which all the user (and administration) can be developed and tested.

2.4 Sequence of Component/Function Integration

2.4.1 Software integration sequence

S1 User Interactions Subsystem

- In this section the subsystems related to all the User's aspects towards the System will be analyzed. A particular attention for all the relevant inputs of the User is the guideline to cover appropriately all the Test Cases. Every division of the section is referred to a particular situation between the entities, the references to the various situations are located in the Sequence Diagrams paragraph in the DD document and the various Scenarios, previously showed up in the RASD document.
  - **S1.1: Autentication and Registration Subsystem**
    It corresponds to the description of what happens at the moment of the very first access of the User to the Mobile App. Hence, She/He shall communicate to the Mobile App and, forwardly, to the Server, if the aim is a new registration or just a logging on to the service.
    - **Operations to test:**
      1. **the particular choice of the User (if she/he wants to sign in or sign up)**
      2. **Fields to be compiled by the User during the sign up operation:**
         - **Mandatory fields:**

(a) name
(b) surname
(c) e-mail
(d) address
(e) country
(f) nationality
(g) credit card number

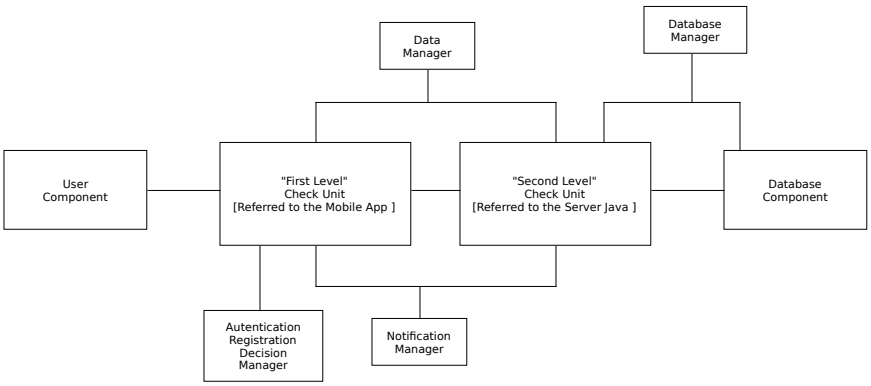
· **Not Mandatory fields:**

(a) phone
(b) mobile
(c) nickname

3. **Fields to be compiled by the User diring the sign in operation:**

(a) a unique field in which a string composed by the name surname sequence, or the e-mail or the nickname (this field identifies the particular User)
(b) the password associated to the User specified by the field above

∗ **Components:**



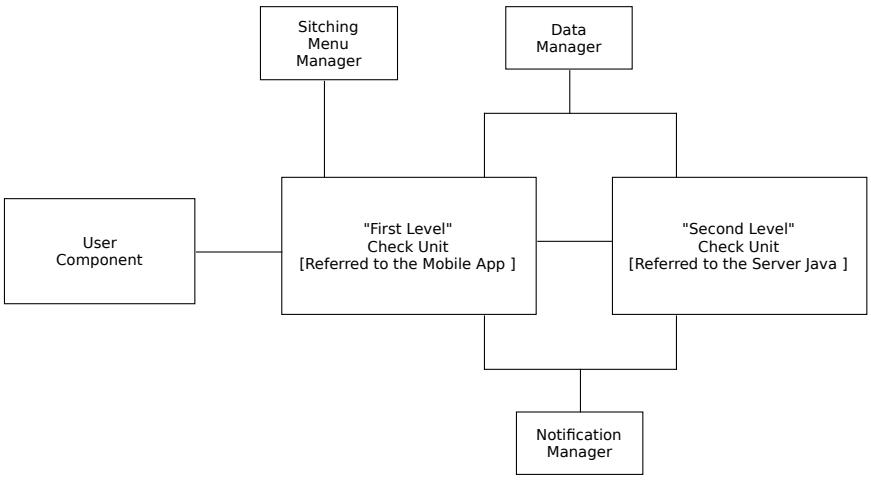
– **S1.2: Choice Menu Subsystem**

It corresponds to the description of the switching options available for the User within the context of the functionalities and the set of windows of the Mobile App.

∗ **Operations to test:**

· User experience with the integration of the commands and the interactions of the views of the App.

∗ **Components:**



  * The presence of the "Second Level Check Unit" is required for the case because of the recovery of some information from the Server. This information, regarding for instance the positions of the User and the Car, as well as the datas of the User for the particular view (see the User Interface Design paragraph in the DD document for a visive reference), are displayed after a correct call to the server.

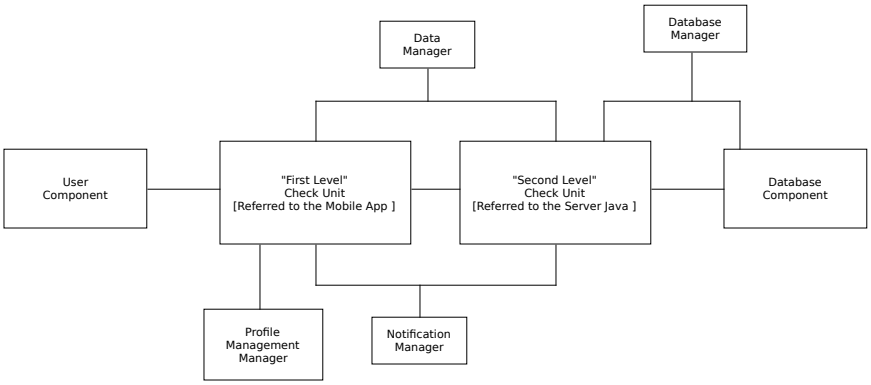
- **S1.3: Profile Management Subsystem**

  * **Operation to test:**

In the table below some changings of parameters due to the User actions in her/his own profile are figured out with respect to a certain grade of priority. In fact is important to understand for this section, how much modifiations by the User will affect directly or not, other internal of the software:

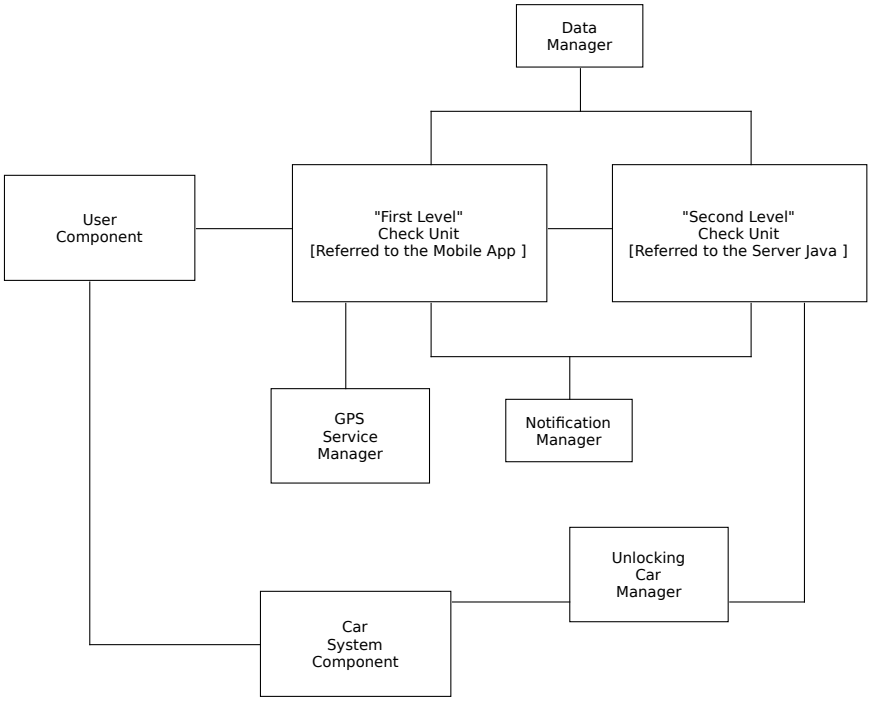
Operation	**Priority**
changing the number of the credit cart used for the payment	High
changing the password for accessing to the service	High
changing the e-mail	High
changing the country	Medium
changing the address	Medium
changing the mobile/phone number	Medium
changing the nickname	Low
changing the profile picture	Low

  * **Components:**



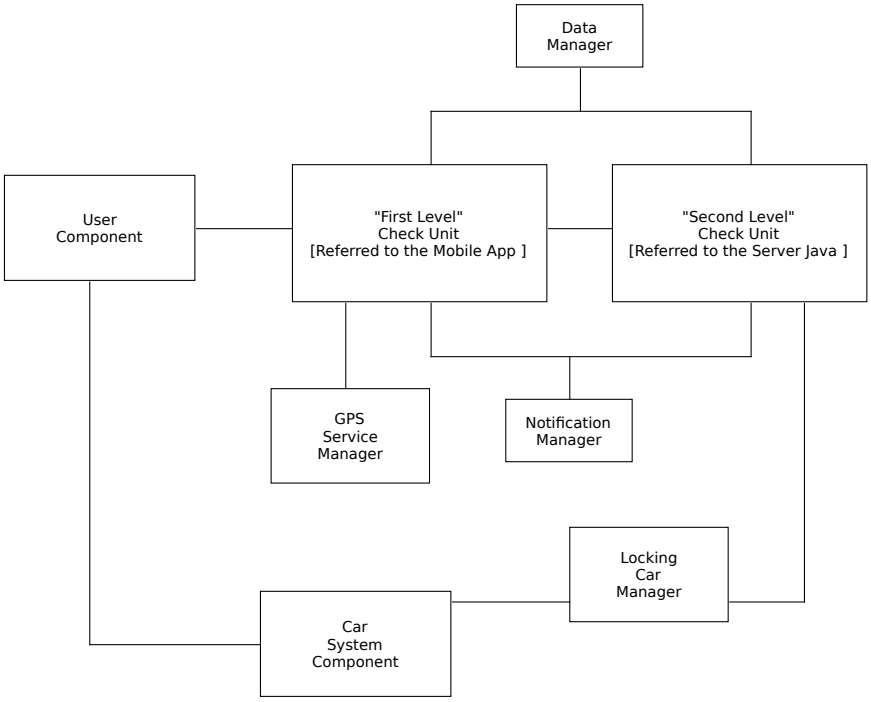
– **S1.4: Unlocking the Car Subsystem:**

  * **Operations to test:**
    1. The User proximity to her/his reserved Car after the reservation (periodically verified in any case)
    2. The User's expressed will to unlock the Car (and so, the process of reponse and reply of the Mobile App and the Server, see the Sequence Diagram in the DD document for discovering more)
    3. The conditions of proximity during the will expressed by the User (and so all the cases in which there can be a negative or a positive reponse from the Server)
    4. The unlock of the Car process (as directly consequence of the points above).
  * **Components:**



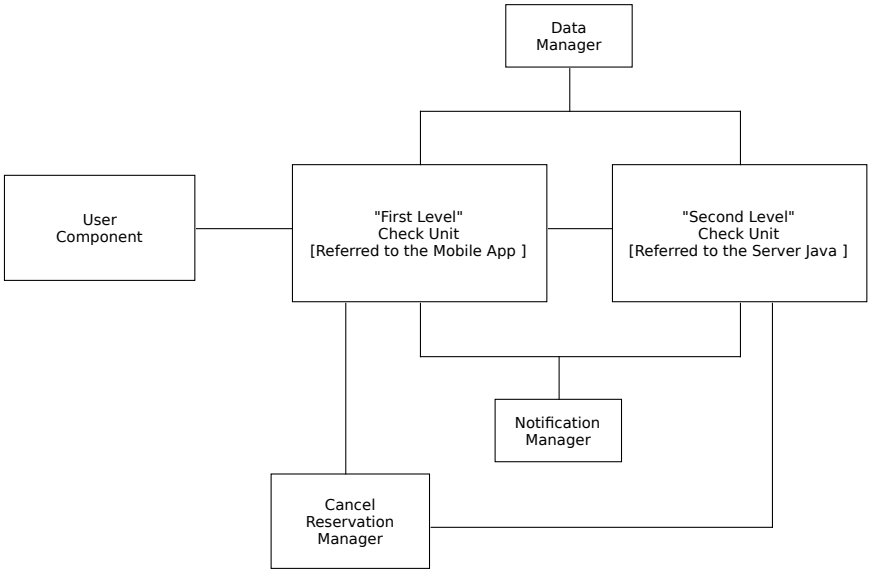
– **S1.5: Locking the Car Subsystem:**

  * **Operations to test:**
    1. The User proximity to her/his reserved Car after the end of the ride, as well as the User condition of being out of the vehicle (the Server makes this analysis thanks to the interaction with the Car's System and indirectly the sensors of the Car, in addition to the usual interaction with the Mobile App. See the Sequence Diagrams in the DD document for discovering more)
    2. The condition of closure of the Car by the User
    3. The lock of the Car process (as directly consequence of the points above).
  * **Components:**



– **S1.6: Cancel a reservation Subsystem:**

  * **Operations to test:**
    1. The Active instance of the service for the User who decides to cancel the current reservation
    2. The deactivation process (it means all the operations in the two ways interaction which have the reservation cancelled as a final result).
  * **Components:**



– **S1.7: Fees and Discounts Subsystem:**
The meaning of this subsystem is to control and properly respond to the various conditions which bring the User in some of the situations to pay some additional fees to the System. In the table below there is a brief recapitulation of the set of fees to pay in the different particular cases (but if you want to know more about it, in the RASD and DD documents the topic is further illustrated):

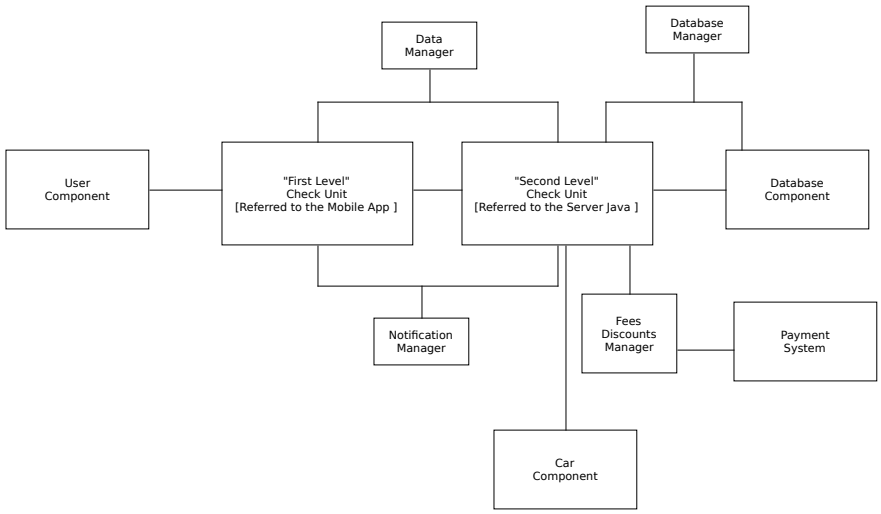
Discount or Fee to be applied to the Payment	**Situation**
+1 euro	The time reservation of 1 hour expires
-10%	The System detects two other passengers in the Car
-50%	The Car has more than 50% of the battery full
-30%	The Car is left in the special parking area and plugged to the power grid after the ride
+30%	The Car is left at more than 3 KM from the nearest power grid station
+30%	The Car is left with more than 80% of the battery empty
(-30%)	If the money saving option is enabled and instructions are correctly followed

∗ **Operations to test:**
1. If for each of the situations just showed all the points of the

column “entered condition” are effectively consumed

2. The process of alerting the User of the exception raised and of displaying her/him the total amount to pay for the particular situation
3. The process of correctly calculating the fee for the particular situation
4. The process of opening correctly an instance between the Transactional System and the User (verifying the correct production of the new object containing all the necessary User's datas necessary to correctly conclude the step)
5. The final result produced by the verification of the sentences above: the System has to open another instance and so repeating the phases if the transaction is incorrectly made, or close the whole circumstance in the opposite case.

∗ **Components:**



– **S1.8: Payment Subsystem:**

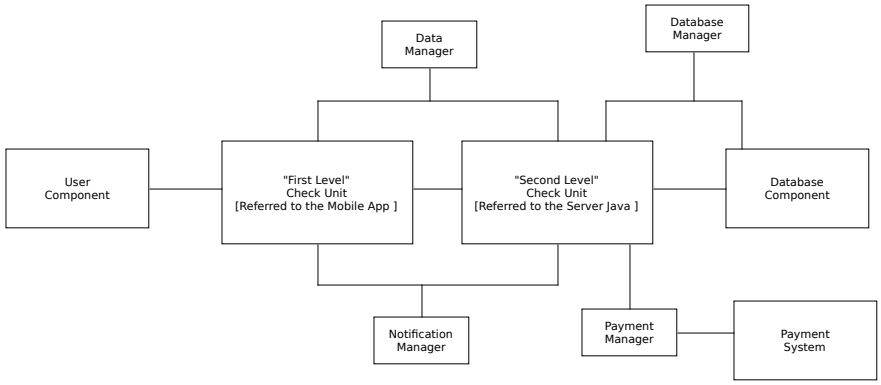
It is very similar to the Fees Subsystem, but in this case the instance between User and Transactional System is opened not for the raising of particular exception due to particular “entered condition”, but just to correctly manage the payment after the end of a ride.

∗ **Operations to test:**

1. The inputs that highlight the end of the current ride of the particular User (see the RASD and DD documents for a detailed description):
   (a) the Car is correctly parked with the engine switched off
   (b) All the passengers, User included, got off the Car

2. The process of calculating the total amount of money to pay by the User.
3. The process of alerting the User of the end of the ride and of displaying her/him the total amount to pay, calculated and verified in the previous step
4. The process of opening correctly an instance between the Transactional System and the User (such as in the Fees Subsystem, but changing some entry parameters)
5. The final result produced by the verification of the sentences above: the System has to open another instance and so repeating the phases if the transaction is incorrectly make, or close the whole circumstance in the opposite case.

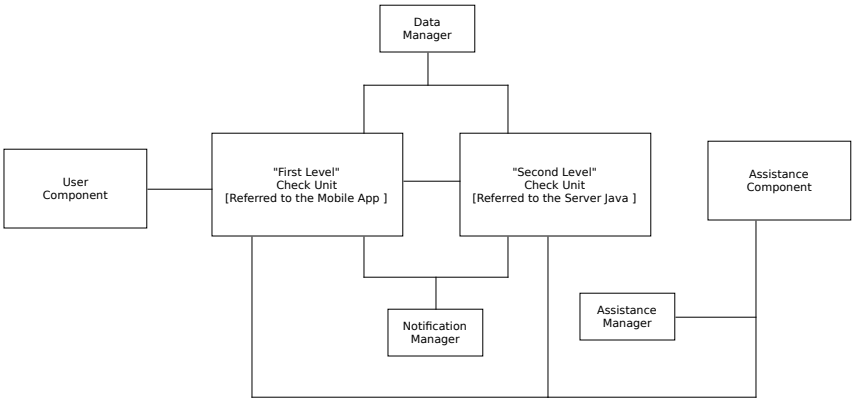
* **Components:**



– **S1.9: Assistance Subsystem**

This section aims to rule out some tests to be made on the communication between the User and an Operator if needed.

* **Operation to test:**
  1. The choice of either talking or chatting with the Operator. The decision is made by the User and communicated to the Server via the Mobile App or the Car's System
  2. The opening and handling of the connection powered by the Server
  3. The closure of the connection once the communication is finished
  4. The management of the relative notification to the User and the Operator if needed.
* **Components:**

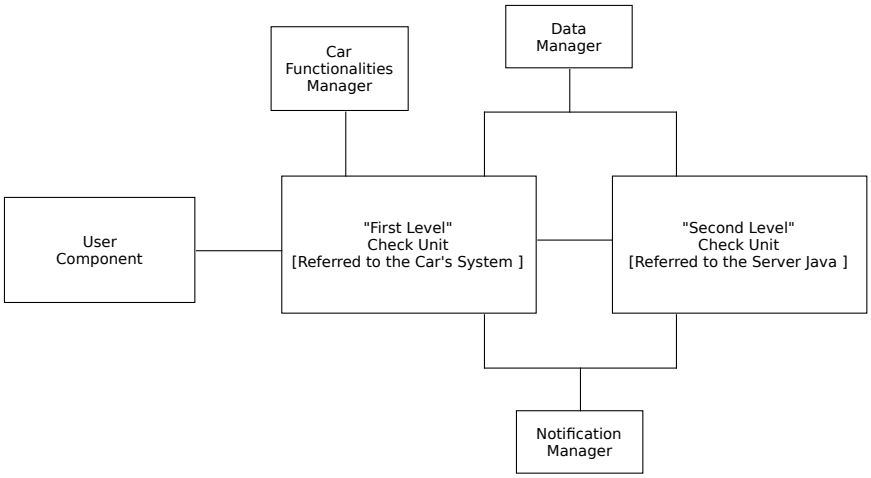


– **S1.10: Car communication Subsystem**
This section expresses a set of extra functionalities designed for the User who starts the ride and outlined by the Car's System. For some of them the presence of the Server is necessary, for other ones the Car's System can completely handle the request and shows the outputs to the User.

* **Operations to test:**
  · The Car's System active condition (it shall be effectively active once the System unlocks the Car)
  · The User Experience related to the Menu Switch, offered to the User as long as she/he gets on the Car (this option is similar to the Mobile App's one)
  · The activation of the money saving option
  · The GPS Functionalities:
    1. The decision of the User to give a particular path to follow for the journey
    2. The indication of the nearest Stations from the path indicated by the User
    3. The commutation of the "actual stations", hence the stations which appear on the road to the User during the drive
    4. The signalations of obstacles, closed road, limits of speed, and so on, as a common navigator
  · The link to a traffic news service if activated
  · The communication with an Assistance Operator if needed by the User or by the System
  · Access to a manual of instructions and dispositions
  · Information about the health and property of the Car
  · Information about the current statistics of the Car (the amount of power left enter in this category)

- A notification manager for the handling of the following message:
  1. The starting of the ride
  2. The finishing of the ride
  3. The actual amount of money to pay
  4. The statistics and summary of the end of the ride:
     (a) Information correlated to the physical aspects of the ride (such as the kilometers of the completed ride, energy spent etc.)
     (b) Information correlated to the final amount of money to pay, eventually with fees and discounts already calculated
     (c) The option to be connected directly from the Car to the Transactional System for the transaction or to be received it as a notification on the Mobile App
- The interactions with the Car sensors for understanding how many other passengers more than the User are entering/leaving the Car
- The interactions with the lock/unlock device of the Car
- The calls and responses made/received to/by the Server in relation of all the situations discussed above
- The Car's System sleeping condition (It shall be effectively in a sleeping mode after the locking required by the Server)

* **Components:**

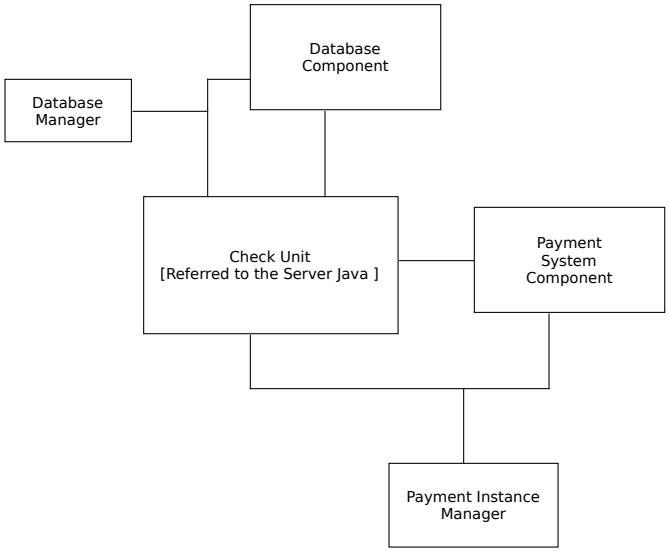


S2 Interaction with the Payment Subsystem

- This section wants to provide a better and brief description of how the System manages the different interactions with the Payment System (if

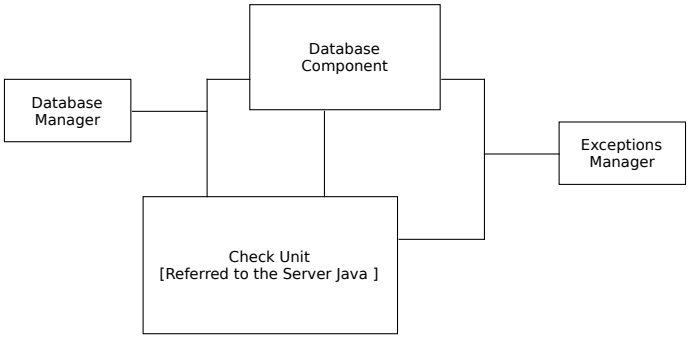
you need a better description of the nature of it, you can refer to the RASD and the DD documents). The System, in particular, has to build a precise instance of an object with some inputs statements whose correctness in the form has to be clear to the Transactional System. On the other way, it will be the Transactional System to return typed values from the corresponding function calls that the System can understand and correcly handle for next steps of the operations. This concepts have to be clear indipendently from the implementation details and all the exceptions in the middle has to be managed by the System. In details:

- **Operations to test:**
  1. **A correct analysis of all the inputs parameters to be delivered to the Transactional System (this means making them correctly in the previous operations and verifying them just before the opening of the communication with the Transactional System):**
     (a) The Object describing the User and her/his specifics (such as name, surname, number of credit card)
     (b) The Object describing the amount of payment to be paid by the User and that the System expects after the call
     (c) The Object describing the nature of the payment (if it regards either an end of the ride instance or a particular fee)
  2. **A correct analysis of all the outputs parameters returned by the Transactional System:**
     (a) The Object regarding the id of the payment (all the ids are stored momentaneously for the current active transactions by the Server) and if there is a correspondence with one in the temporary list
     (b) The Object which contains the informations about the payment and the User (hence the Server can start to open a new communication with the User to alert)
     (c) The Object regarding the result of the transaction:
        i. if the transaction is incorrect the System has to repeat the whole operation
        ii. if the transaction is correct the System has to inform the User and correctly close the communication for the particular case with it and with the Transactional System.
- **Components:**



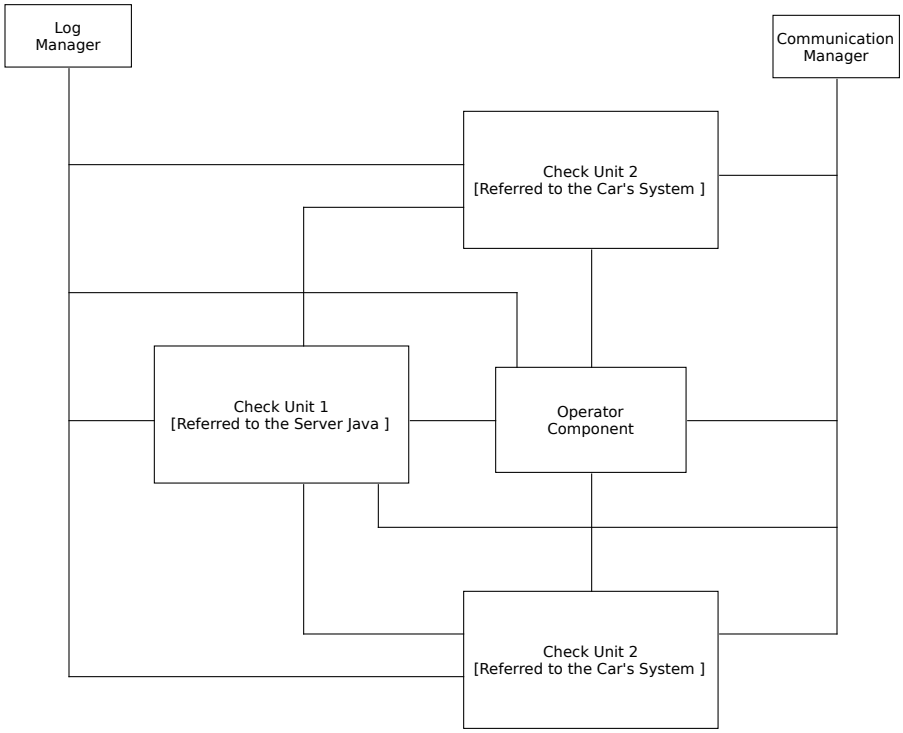
S3 Interaction with the Database Subsystem

- Such as for the previous section, some interactions with the Database shall be controlled by the Server and some of the exceptions handled for avoiding in general a crash of the application.
  - **Operations to test:**
    - **If the exceptions affecting tables of the Database are handled in a way that allows the System to complete the analized operations anyway, this is important for all the different kind of operations which regard the interaction with the System and the Database and, deeper:**
      - Exceptions on null values
      - Exceptions on inserts
      - Exceptions on updates
      - Exceptions on deletes
      - Exceptions on particular conditions of the Database
    - **Components:**



S4 Management Unit Subsystem:

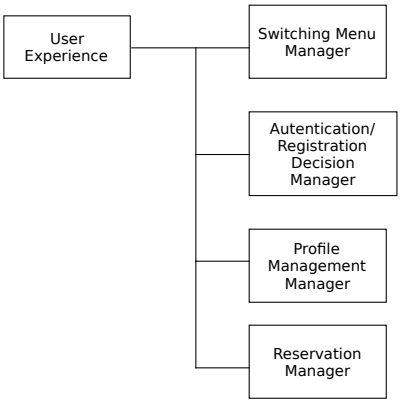
- For better understanding the purpose of this section, you can think about all the Subsystems described in the **S1** but focalized in an "inverse manner". In other words, now it is not the User the centre of the topic, but the Administrators and Operators which have the important role to govern the entire Service thanks to some tools offered by the System.
  - **Operations to test:**
    - **The mapping between the set of actions taken by the User in the application in one of its particular context and the log file produced for the Administrator**
      The actions of the Users for the different scenarios are stated inside a log file, the test regards checking if the actions are univoquely mapped inside it and updated with respect to the current date and time. This is important because in exceptional and common case the Administrator can always relies on this kind of documentation.
    - **If the channels of communications between the User and the Operator are in functions**
      This set of channels regards in details:
      - A direct communication with a chat or with a call
      - An "injection of code" which allows the operator to diplay to the User some particular statements of message from the application (the Mobile App or the Car's System) at the occurence.
  - **Components:**



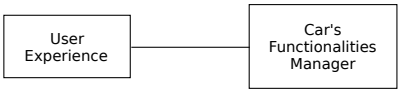
2.4.2 Subsystem integration sequence

Following the bottom up strategy and the specifics already pointed out in the previous analysis of all the Subsystems forming the test strategy, the approach to integrate all the cases of test will be designed in the section. The used metodologies take place paying attention to the entities and the actors "at the border" of the System. At first position of all of them we can find the User: for her/him it is intended the whole Service. But details are important for other parts too: such as the various Operators and Assistent who are "behind the scene", the Payment System and the Database. Their absence seriously compromise the final role of the entire Application. Finally, the hypothesis for a good integration is that everything related to the implementation was already been finalized. The modules analyzed below, are considered as "containers" for all the cases of interactions between the entity A and the entity B. All these interactions will be covered by some tests for the purpose. In those processes in which also the integration of an entity C is required, particular "markers" can be placed and retrieved at the right moment of the testing procedure. In other words we can say that the test is suspended making some conjectures for going on in the analysis, hence formulating a correct reply without passing from entity C, but signing the inputs for the future tests covered in the appropriate module.

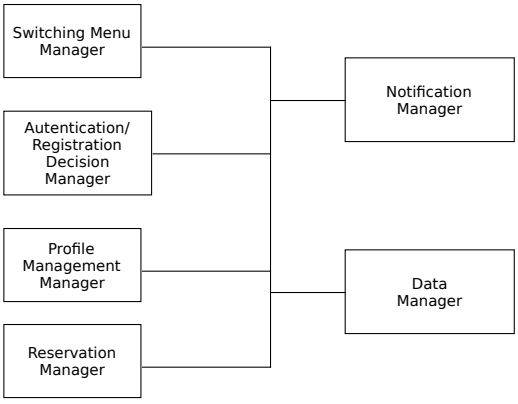
S5 **Integration between the functionalities offered to the User and the Mobile App**



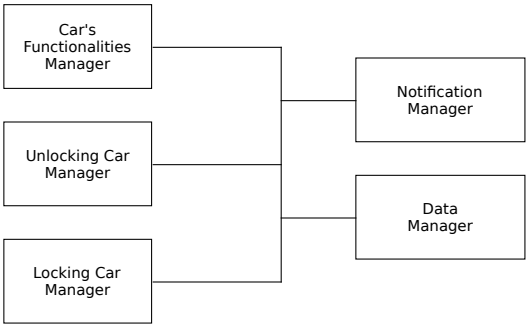
S6 **Integration between the functionalities offered to the User and the Car's System**



S7 **Integration between the Mobile App's calls and the Server Java**

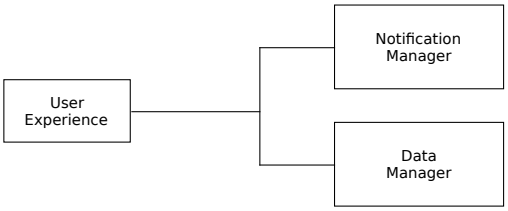


S8 **Integration between the Car's System's calls and the Server Java**



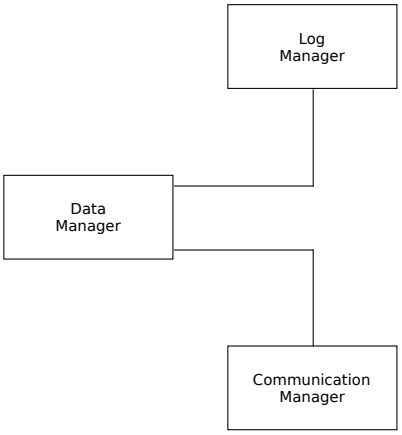
S9 **Integration between the User and the Server Java**

Once the points S5, S6, S7, S9 are correctly managed, everything which brings the transictions procedure between the Server and the User can be "hide" for this kind of tests (in other words the User and the Server are seen as the end points of the communication for this particular situation).

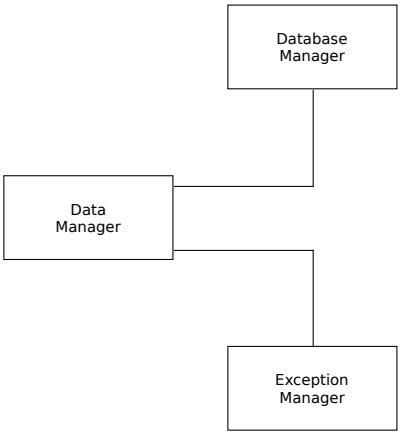


S10 **Integration between the functionalities required from the Payment System and the Server Java**

S11 **Integration between the functionalities offered to/required by the Assistance Operator/Administrator and the Server Java**



S12 **Integration between the functionalities offered to/required by the Database and the Server Java**



S13 **Final tests integrations**

This final work include a set of examples which are more similar of real situations. Topics for the section are very near to the scenarios illustrated in the RASD document in their definitions. It is important to make a final integration of all the units in this way, because also the minimal error can be discovered and fixed and also if it is impossible to reach an ideal margin of error equal to 0, it can be reduced enough to have a final, running version of the software, ready to be competitive on the market.

3 Individual Step and Test Description

In this chapter we provide a detailed description of the the test cases that have to be performed with respect to the specific integration of the components (integration steps are described in the previous chapter).

The interfaces names used in this section are as defined in the Design Document.

3.1 Car Management System

3.1.1 CarEventAPI

The following table defines the actions the Car Controller takes upon receiving each type of event message from a car.

Message type	Effect
startEngine	Car Controller executes the function call startEngine(carID, time); where: • carID is the ID of the current car • time is the timestamp contained in the message
stopEngine	Car Controller executes the function call stopEngine(carID, time); where: • carID is the ID of the current car • time is the timestamp contained in the message
lock	Car Controller executes the function call lock(carID, time); where: • carID is the ID of the current car • time is the timestamp contained in the message

3.1.2 CarStatusServices

The following tables detail the desidered effect of every method exposed by the Car State Manager to the Car Controller.

startEngine(carID, time)	
Input	Effect
(null, null)	Car State Manager exception: NPE
(null, valid timestamp)	Car State Manager exception: NPE
(valid ID, null)	Car State Manager exception: NPE
(valid ID, time <= starting time of the current Ride)	Car State Manager exception: invalid timing
(valid ID, time > starting time of the current Ride) && the current car state is not Reserved with Ride	Car State Manager exception: invalid activation
(valid ID, time > starting time of the current Ride) && the current car state is Reserved with Ride	• Car State Manager sets the car's state to Reserved with Active Ride • Car State Manager notifies the Reservation Manager of the start of an Active Ride for the current Reservation with the function call startActiveRide(carID, time)

stopEngine(carID, time)	
Input	Effect
(null, null)	Car State Manager exception: NPE
(null, valid timestamp)	Car State Manager exception: NPE
(valid ID, null)	Car State Manager exception: NPE
(valid ID, time <= starting time of the current Active Ride)	Car State Manager exception: invalid timing
(valid ID, time > starting time of the current Active Ride) && the current car state is not Reserved with Active Ride	Car State Manager exception: invalid deactivation
(valid ID, time > starting time of the current Active Ride) && the current car state is Reserved with Active Ride	• Car State Manager sets the car's state to Reserved with Ride • Car State Manager notifies the Reservation Manager of the end of the current Active Ride with the function call endActiveRide(carID, time)

lock(carID, time)	
Input	Effect
(null, null)	Car State Manager exception: NPE
(null, valid timestamp)	Car State Manager exception: NPE
(valid ID, null)	Car State Manager exception: NPE
(valid ID, time <= starting time of the current Ride)	Car State Manager exception: invalid timing
(valid ID, time > starting time of the current Ride) && the current car state is not Reserved with Ride	Car State Manager exception: invalid lock
(valid ID, time > starting time of the current Ride) && the current car state is Reserved with Ride	• Car State Manager sets the car's state to Reserved • Car State Manager notifies the Reservation Manager of the end of the current Ride with the function call endRide(carID, time)

3.1.3 CarControlProtocol

The following methods are exposed by the Car Controller to the Car Manager in order to issue commands to the cars.

unlock(carID)	
Input	Effect
Invalid CarID	Car Controller exception: invalid car ID
Valid CarID	The Car Controller sends a message of type unlock to the correct car.

reserve(carID)	
Input	Effect
Invalid CarID	Car Controller exception: invalid car ID
Valid CarID	The Car Controller sends a message of type reserve to the correct car.

free(carID)	
Input	Effect
Invalid CarID	Car Controller exception: invalid car ID
Valid CarID	The Car Controller sends a message of type free to the correct car.

invalidate(carID)	
Input	Effect
Invalid CarID	Car Controller exception: invalid car ID
Valid CarID	The Car Controller sends a message of type invalidate to the correct car.

retrieve(carID)	
Input	Effect
Invalid CarID	Car Controller exception: invalid car ID
Valid CarID	The Car Controller sends a message of type retrieve to the correct car.

3.1.4 CarControlAPI

The following table defines the actions the car's on board system must take upon receiving each type of command message from the server.

Note: the car's state mentioned in the table is the car's internal representation of its state, not the server's.

Message type	Effect
unlock	• The car's state is set to Reserved with Active Ride • The car's door unlock • An ACK message is sent back to the server
reserve	• The car's state is set to Reserved • An ACK message is sent back to the server
free	• The car's state is set to Available • An ACK message is sent back to the server
invalidate	• The car's state is set to Out Of Order • An ACK message is sent back to the server
retrieve	• The car's state is set to Available • An ACK message is sent back to the server

3.1.5 CarAccessAPI

The following methods are exposed by the Car Manager to the Reservation Manager in order to:

1. issue car state changes caused by user action;
2. allow for queries on car sets.

reserve(carID, time)	
Input	Effect
(null, null)	Car State Manager exception: NPE
(null, valid timestamp)	Car State Manager exception: NPE
(valid ID, null)	Car State Manager exception: NPE
(valid ID, valid timestamp) && the current car state is not Available	Car State Manager exception: invalid reservation
(valid ID, valid timestamp) && the current car state is Available	• Car State Manager sets the car's state to Reserved • Car State Manager calls the Car Controller's reserve(carID) method

free(carID, time)	
Input	Effect
(null, null)	Car State Manager exception: NPE
(null, valid timestamp)	Car State Manager exception: NPE
(valid ID, null)	Car State Manager exception: NPE
(valid ID, valid timestamp) && the current car state is not Reserved	Car State Manager exception: invalid reservation deletion
(valid ID, valid timestamp) && the current car state is Reserved	• Car State Manager sets the car's state to Available • Car State Manager calls the Car Controller's free(carID) method

getCarsByPosition(GPSPosition)	
Input	Effect
(null)	Car Pool Manager exception: NPE
(valid GPS position)	Return value: a set of <carID, GPSPosition> tuples

getCarsByAddress(address)	
Input	Effect
(null)	Car Pool Manager exception: NPE
(a string not matching any address in the database)	Car Pool Manager exception: invalid address
(valid address)	Return value: a set of <carID, GPSPosition> tuples

3.1.6 ReservationServices

The following methods are exposed by the Reservation Manager to the Car Manager in order to allow a correct computation of a reservation's cost.

startRide(carID, time)	
Input	Effect
(null, null)	Reservation Manager exception: NPE
(null, valid timestamp)	Reservation Manager exception: NPE
(valid ID, null)	Reservation Manager exception: NPE
(valid ID, valid timestamp) && there isn't any Reservation for the car	Reservation Manager exception: invalid ride start
(valid ID, valid timestamp) && there is a rReservation for the car	The Reservation's information in the database is updated with the presence and starting time of the Ride

endRide(carID, time)	
Input	Effect
(null, null)	Reservation Manager exception: NPE
(null, valid timestamp)	Reservation Manager exception: NPE
(valid ID, null)	Reservation Manager exception: NPE
(valid ID, valid timestamp) && there isn't any Reservation for the car	Reservation Manager exception: invalid ride end
(valid ID, valid timestamp) && there is a Reservation, but there isn't any ride for the car	Reservation Manager exception: invalid ride end
(valid ID, valid timestamp) && there is a Reservation, a Ride and an Active Ride that hasn't ended yet	Reservation Manager exception: invalid ride end
(valid ID, valid timestamp) && there is a Reservation, a Ride and no Active Rides for the car	The Ride's information in the database is updated with the ending time of the Ride

startActiveRide(carID, time)	
Input	Effect
(null, null)	Reservation Manager exception: NPE
(null, valid timestamp)	Reservation Manager exception: NPE
(valid ID, null)	Reservation Manager exception: NPE
(valid ID, valid timestamp) && there isn't any Reservation for the car	Reservation Manager exception: invalid activation
(valid ID, valid timestamp) && there is a Reservation, but there isn't any ride for the car	Reservation Manager exception: invalid activation
(valid ID, valid timestamp) && there is a Reservation, a Ride and an Active Ride that hasn't ended yet	Reservation Manager exception: invalid activation
(valid ID, valid timestamp) && there is a Reservation, a Ride and no Active Rides for the car	The Ride's information in the database is updated with the presence and starting time of the Active Ride

endActiveRide(carID, time)	
Input	Effect
(null, null)	Reservation Manager exception: NPE
(null, valid timestamp)	Reservation Manager exception: NPE
(valid ID, null)	Reservation Manager exception: NPE
(valid ID, valid timestamp) && the current car state is not Reserved	Reservation Manager exception: invalid deactivation
(valid ID, valid timestamp) && there is a Reservation, but there isn't any ride for the car	Reservation Manager exception: invalid deactivation
(valid ID, valid timestamp) && there is a Reservation, a Ride and an Active Ride that hasn't ended yet	The Ride's information in the database is updated with the ending time of the Active Ride
(valid ID, valid timestamp) && there is a Reservation, a Ride and no Active Rides for the car	Reservation Manager exception: invalid deactivation

The badPark method is called my the Car Manager when it detects a car has been parked in a NonSafeArea at the end of a Ride.

badPark(carID)	
Input	Effect
(null)	Reservation Manager exception: NPE
(valid carID)	The Ride's information in the database is updated with the information concerning the bad behaviour of the user.

The farPark method is called my the Car Manager when it detects a car has been parked in an Area too far from the charging stations, or with a battery level too low.

farPark(carID)	
Input	Effect
(null)	Reservation Manager exception: NPE
(valid carID)	The Ride's information in the database is updated with the information concerning the bad behaviour of the user.

The rechargingPark method is called my the Car Manager when it detects a car has been parked in a ChargingArea at the end of a Ride.

rechargingPark(carID)	
Input	Effect
(null)	Reservation Manager exception: NPE
(valid carID)	The Ride's information in the database is updated with the information concerning the virtuous behaviour of the user.

The goodPark method is called my the Car Manager when it detects a Ride has ended with the car's battery level not more than 50% empty.

goodPark(carID)	
Input	Effect
(null)	Reservation Manager exception: NPE
(valid carID)	The Ride's information in the database is updated with the information concerning the virtuous behaviour of the user.

3.2 User Management System

3.2.1 AuthenticationServices

registerNewUser(newUserParamaters)	
Input	Effect
A Null parameter	A NullPointerException is raised
new user's email is already associated to another user	An InvalidRegistrationException is raised with an explanation of the problem with the email
new user's driving license is already associated to another user	An InvalidRegistrationException is raised with an explanation of the problem with the email
invalid new user's personal information fields	An InvalidRegistrationException is raised with an explanation of the problem with the fields.
valid new user's informations	the User object representing the newly created user

login(userLoginParameters)	
Input	Effect
A Null parameter	A NullPointerException is raised
invalid email	An InvalidLoginException is raised with an explanation of the problem with the email
invalid password	An InvalidLoginException is raised with an explanation of the problem with the password

deleteUser(user)	
Input	Effect
A Null parameter	A NullPointerException is raised
a valid user	boolean true confirming the result of the operation

3.2.2 ReservationServices

findCar(position)	
Input	Effect
A Null parameter	A NullPointerException is raised
invalid position	An InvalidPosition exception is raised
valid position object	Return value: a set of <carID, GPSPosition> tuples, where all the cars listed are in the Available state

findCar(address)	
Input	Effect
A Null parameter	A NullPointerException is raised
invalid address	An InvalidAddress exception is raised
valid address	Return value: a set of <carID, GPSPosition> tuples, where all the cars listed are in the Available state
insertReservation(user, car)	
Input	Effect
A Null parameter	A NullPointerException is raised
The user is incompatible with the specified car with respect to the business logic	A ReservationErrorException is raised
listReservation(user)	
Input	Effect
A Null parameter	A NullPointerException is raised
invalid user object	An InvalidParameterException is raised
valid user object	A Reservation object which represent the reservation made by the specified user
deleteReservation(reservation)	
Input	Effect
A Null parameter	A NullPointerException is raised
A reservation that has already been canceled	InvalidParameterException
valid and active reservation	boolean true

4 Tools and Test Equipment Required

4.1 Tools

Here is a brief explaination of all the tools that we want to use to perform all the main passages of the Integration Testing described in previous chapters of this document.

- JUnit: is the mainstream suite for testing software components and units written in Java. We aim to take advantage of all the main functionalities provided by JUnit to test software units; here are some of the advantages that we can obtain by using this tool:
  - JUnit tests allow you to write codes faster, which increases quality.

- JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
- JUnit tests can be organized into test suites containing test cases and even other test suites.
- JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented.

Java code will be used mainly for those software components that are related to the mobile application for the Android platform.

The Android SDK provides additional tools that can be used to verify Java the correctness code and to test the performance of the overall application on different types of device.

These tools are:

- the Memory Profiler
- the Memory Monitor and Allocation Tracker
- the Battery Profiler

Talking about mobile applications, there are software tools (provided by Apple) for automatically checking correctness and performance of iOS applications on several versions of virtual or physical iOS devices. We want to mention some of them such as:

- the Performance Profiling Tool
- the Activity Monitor
- the MallocDebug

With respect to the Business Logic, being it developed within the Java 2E framework (as described in previous documents), we aim to use:

- Arquillian integration testing framework: this tool enables us to execute tests against a Java container in order to check that the interaction between a component and its surrounding execution environment is happening correctly.

4.2 Test equipment

All the integration tests have to be performed in within a consisten context that better represents the real context that the system will face after the production phase.

For what concerns the mobile application context, all SDKs from the major platforms like Android and iOS come with a set of simulators of different devices or versions of the same type of device.

Although the possibility given by simulation, we aim to exploit this technique only to reproduce the context of devices that are not mainstream among the smartphone market.

The mobile application will be tested using physical devices that represent the mainstream of the smartphone market (also with respect to the expected users); with respect to our analysis of the smartphone market, these are devices we aim to use for testing purposes.

- iPhone 6
- iPhone 6 plus
- Samsung Galaxy Note II