Politecnico di Milano
MSc Computer Science and Engineering

Software Engineering 2
Academic Year 2016-2017

Integration Test Plan Document

PowerEnJoy

Authors:

Melloni Giulio 876279
Renzi Marco 878269
Testa Filippo 875456

Reference Professor:
Mottola Luca

Release Date: January 15th, 2017
Version 1.0

Table of Contents

List of Figures

1 | Introduction

1.1 Revision History

Version	Date	Summary
1.0	15/01/2017	First release of the document

1.2 Purpose and Scope

The Integration Test Plan Document (later on simply referred as *ITPD*) covers the whole process of integration of the components of the system; in particular it explains in a detailed way how these ones will be integrated and the specific sequence that must be followed.
The document firstly tackles the integration plan issue starting from some assumptions, as listed in the *Entry Criteria* section; after that the component diagram presented in the *DD* is split up in different macro areas that will be useful to carry out the bottom-up approach (*Elements to be integrated* section).
Then the strategy that leads the integration plan is stated and, obviously along with it, its rationale. Following the strategy, the sequence of integration of the components and the single steps are derived.
The main goal of the integration test plan is firstly to ensure that all the developed components properly combine together in order to accomplish all the functionalities and secondly to spot any undesirable behaviour of the system.

1.3 List of Definitions and Abbreviations

PowerEnJoy is the name of the system that has to be developed.

System sometimes called also *system-to-be*, represents the application that will be described and implemented. In particular, its structure and implementation will be explained in the following documents. People that will use the car-sharing service will interact with it, via some interfaces, in order to complete some operations (e.g.: reservation and renting).

Renting it is the act of picking-up an available car and of starting to drive.

Ride the event of picking-up a car, driving through the city and parking it. Every Ride is associated to a single user and to a single car.

Reservation it is the action of booking an available car.

Car a car is an electrical vehicle that will be used by a registered user.

Not Registered User indicates a person who hasn't registered to the system yet; for this reason he can't access to any of the offered function. The only possible action that he can carry out is the registration to get a personal account.

Registered User interacts with the system to use the sharing service. He has an account (which contains personal information, driving license number and payment data) that must be used to access to the application in order to exploit all the functionalities.

Employee it's a person who works for the company, whose main task is to plug into the power grid those cars that haven't been plugged in by the users. He is also in charge of taking care of the status of the cars and of moving the vehicles from a safe area to a charging area and vice versa if needed.

Safe Area indicates a set of parking lots where the users have to leave the car at the end of the rent; the set of the Safe Areas is pre-defined by the system management. These areas are spread all over the city.

Plug defines the electrical component that physically connects the car to the power grid.

Charging Area is a special *Safe Area* that also provides a certain number of plugs that connect the cars to the power grid in order to recharge the battery.

Registration the procedure that an unregistered user has to perform to become a registered user. At the end, the unregistered user will have an account. To complete this operation three different types of data are required: personal information, driving license number and payment info.

Search this functionality lets the registered user search for available cars within a certain range from his/her current position or from a specified address.

RASD is the acronym of *Requirements Analysis and Specification Document*

DD is the acronym of *Design Document*

ITPD is the acronym of *Integration Test Plan Document*

1.4 List of Reference Documents

- Project Assignments 2016-2017
- RASD v1.1
- DD v1.1

2 | Integration Strategy

2.1 Entry Criteria

There are some criteria that impose some conditions on the project testing phase.
Firstly some considerations on the level of completion of the components with respect to their functionalities:

- The **Dispatcher** must have been fully implemented in order to route the simulated requests.
- Controllers like the **ReservationManager**, the **RegistrationManager**, the **StateManager**, the **LogInManager** and the **RideManager** have to expose sufficiently developed interfaces in order to be able to test the requests management.
- Components like the **Payment Manager** and the **MapController** that are to be linked with third-party components (**Payment System** and **MapServce**) must have been fully developed in order to use the external APIs.

Secondly, the *Requirements Analysis and Specification Document* and the *Design Document* must have been written.
Thirdly, the components must have been individually tested (unit testing is not intrinsically part of this testing phase) in order to ensure that bugs from the upcoming integration tests will be caused exclusively by the iterations among these components and not by any kind of internal problem.

2.2 Elements to be Integrated

In order to build the full *PowerEnJoy* system all its components have to be properly integrated. In this section the focus is on which components are selected and how these are aggregated.
Let us consider the component diagram of the *Design Document* to refer to the components to be integrated. For the integration testing purpose it is useful to organize the components into logical **Macro Areas** that will support the testing process as explained in the *Integration Test Strategy* section:

- **Input Area** includes *ViewRender* and *Dispatcher* components. This pair of modules should be tested together to ensure that all input requests are properly received by the system.

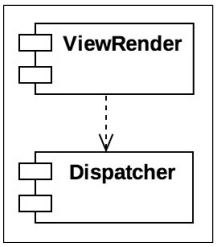

Figure 2.1: Input Area

- **Management Area** includes the *ReservationManager*, the *RegistrationManager*, the *StateManager*, the *LogInManager*, the *MapController*, the *RideManager* and the *Dispatcher*. These modules are responsible for the business logic of the application and consequently should be tested together.

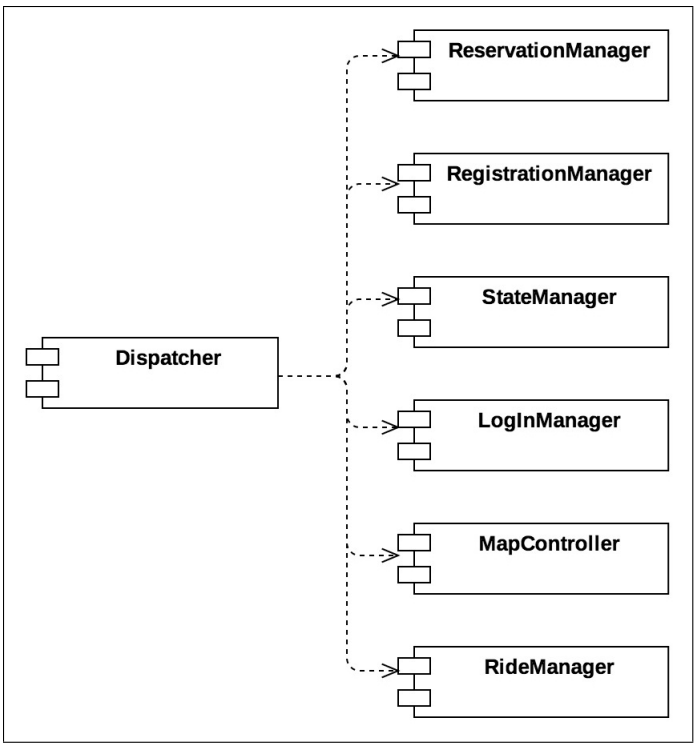

Figure 2.2: Management Area

- **Render Area** is the set made of the *ViewRender* and of the *ReservationManager*, the *RegistrationManager*, the *StateManager*, the *LoginManager*, the *MapController*, the *RideManager*. This logical area has to be tested in order to ensure that all managers can update the view of the application without bugs.

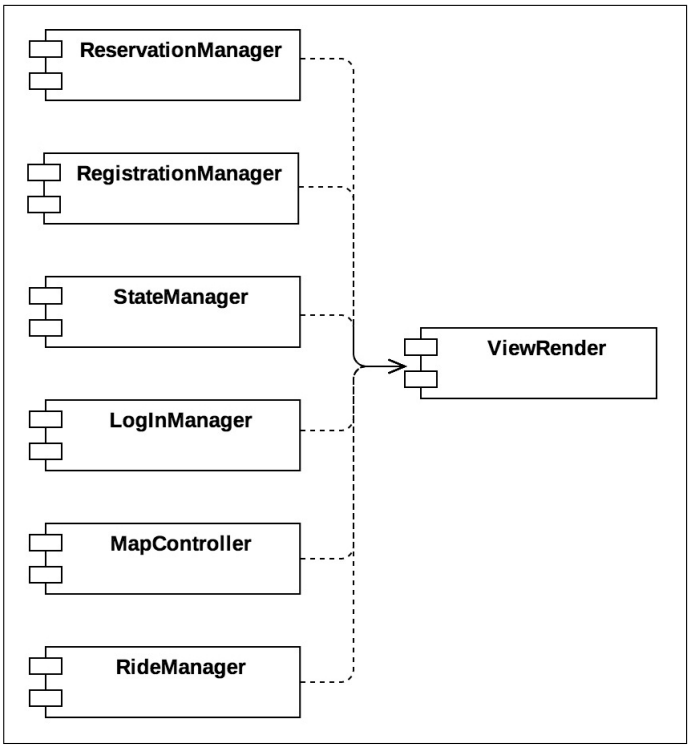

Figure 2.3: Render Area

- **Ride Area** includes *RideManager*, *MapController*, *RideCostCalculator* and *PaymentManager*. The tests on this area is crucial because it is responsible of the costs computation and of the payment process.

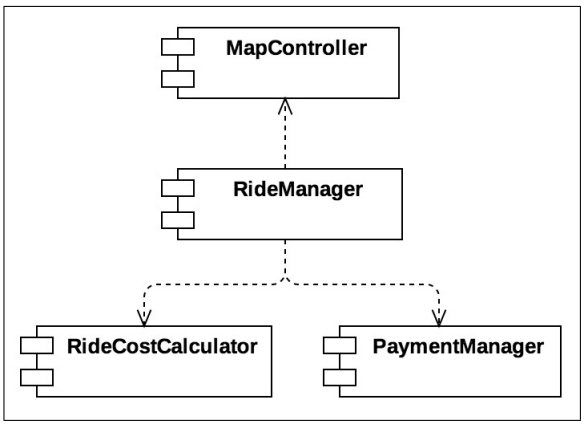

Figure 2.4: Ride Area

- **Data Area** is the group of components that deal with the *Model* and the external *DBMS*. This is made of the *Model* itself and of the *ReservationManager*, the *RegistrationManager*, the *StateManager*, the *LoginManager*, the *MapController*, the *RideManager*. Tests in this area aims at verifying the correctness of data through the various operations that the system has to perform on them.

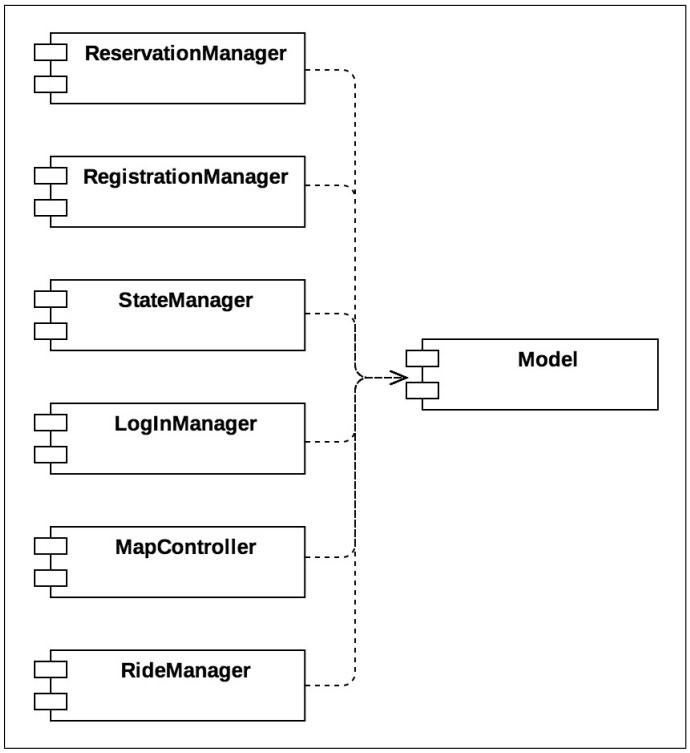

Figure 2.5: Data Area

- **CarCommunication Area** is the pair of *ServerCommunicationManager* and *CarCommunicationManager*. Here the tests have to ensure that flow of information in both directions is feasible and consistent.

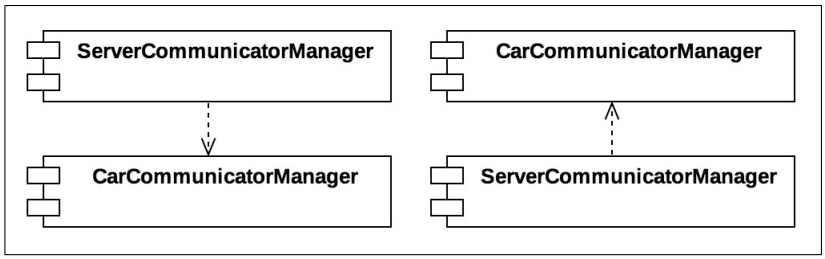

Figure 2.6: CarCommunication Area

- **Car Area** is the logical set of components that have to be tested on the car. *CarCommunicationManager*, *CentralUnit* and *ScreenManager* are part of the Built-in sw for the car.

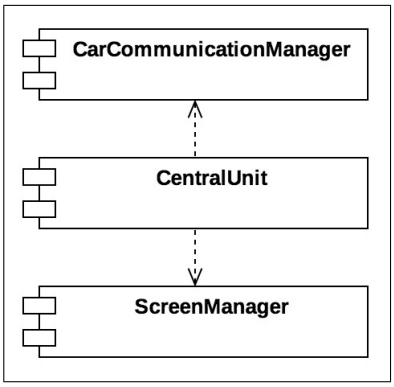

Figure 2.7: Car Area

Please note that the given groupings do not represent a partition of the set of components of the system (some components are shared by more than one macro area) but just a logical division that is convenient to carry out the integration testing because each macro area is responsible of a sub-set of the system functionalities. Finally, a remark on the external components (*MapService*, *Payment System* and *DBMS*): they are already available for integration testing and they only require that the internal components that are faced with are fully developed.

2.3 Integration Test Strategy

The integration testing process will be carried out with both bottom-up and threads approaches. In particular, the bottom-up testing will be executed between modules that belong to the same macro area (as defined in the *Elements to be Integrated* section) throughout their development process while the threads analysis will be eventually performed among modules of different areas when the previous internal tests are successfully passed.
This testing strategy is incremental by construction because it follows the the development of the components and consequently it makes it easier to spot possible errors during the implementation. As portions of components are added to the existing ones, the integration testing will be triggered on the new parts making use of suitable drivers in order to simulate the calls from one caller component to the called one that has to be tested.
This continuous iteration of the bottom-up approach guarantees the testing coverage of all the possible interactions of the components.
As previously mentioned, a thread analysis has to be performed too. This testing phase aims at verifying that the chains of function calls among components of different macro areas produce correct actions. The threads testing approach is chosen because it simulates the standard behaviour of the system, in terms of user requests. It could be considered a means to study the system performances too in this sense.

2.4 Sequence of Component Integration

This section will illustrate how the two different testing approaches, bottom-up and thread, are carried out. The order of presentation (that is bottom-up first and thread second) reflects the fact that to do a thread analysis on a set of components one must have performed the bottom-up testing on the same items first. A set of tables and figures will help to describe these two strategies.

2.4.1 Bottom-Up Integration Strategy

The bottom-up integration testing will be adopted within each *Macro Area*. More precisely, for each pair of components that have function calls from one to the other a proper test suite will take care of testing all the possible interactions. After the testing of a pair of components, a third component which is in relation with one of this two tested modules is added and again its calls are checked. The same reasoning applies to the remaining components that belong to the same area.
Let us introduce an example. Consider the components of the *Input Area* as defined in the *Elements to be Integrated* section:

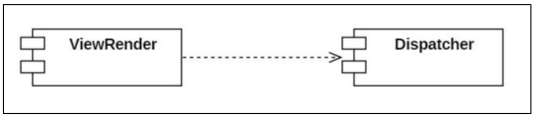

Figure 2.8: Bottom-up test example

where the following notation is adopted to state that Component 1 calls a function exposed by Component 2:

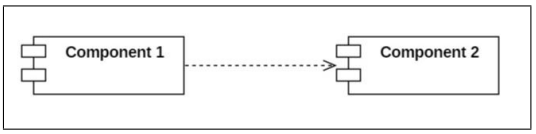

Figure 2.9: Bottom-up notation

In the running example the bottom-up approach states that the already implemented part of the **ViewRender** *drives* the already implemented portion of the **Dispatcher** component because the former calls all the methods exposed by the latter.
The list of all the most significant methods invocations involved in the bottom-up testing for each pair of components of each *Macro Area* is listed in the *Individual Steps and Test Description* section.

2.4.2 Thread Integration Strategy

The thread testing is performed in order to verify that chains of function calls among modules of different *Macro Areas* lead to correct executions. This type of analysis is carried out when the components that it involves have a sufficient level of completion to support one specific functionality. The following figures show the most relevant threads analysis for the main functionalities of the system.
Let us consider the *Login Thread* as an example:

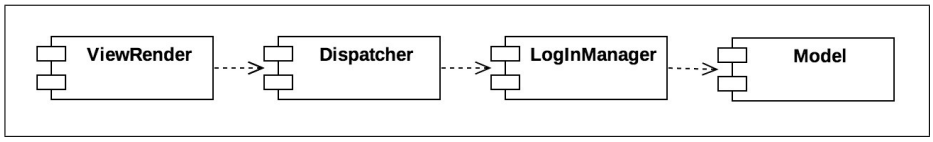

Figure 2.10: Login Thread

In order to complete the login operation the *ViewRender* takes the request from the client and addresses it to the *Dispatcher*. This last component realizes that this specific request

should be handled by the *LogInManager* which in turn needs to query the *Model* to retrieve the client information.
Please note that in these diagrams the purpose is only to show which components are needed to assess a specific functionality and not the entire control flow (see the sequence diagrams in the *Design Document* for that). As such, some call links are omitted (for example the arc from the *LogInManager* to the *ViewRender* meaning that a web page is displayed at the end of the process is not present because it is not of interest).
The following figures can be read with the same reasoning.

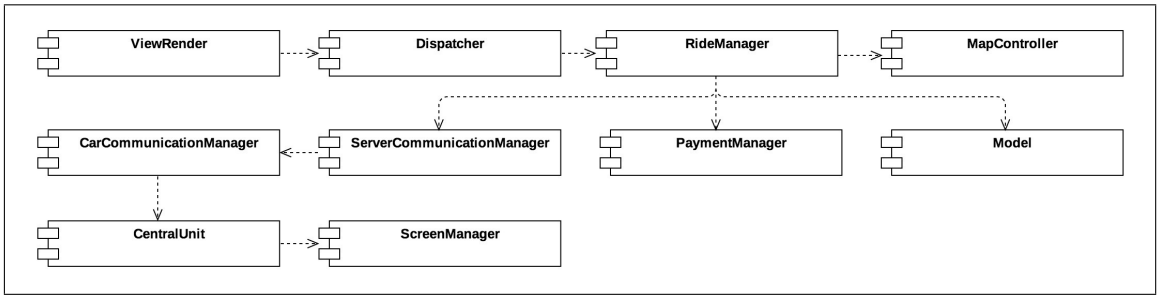

Figure 2.11: Ride Start Thread

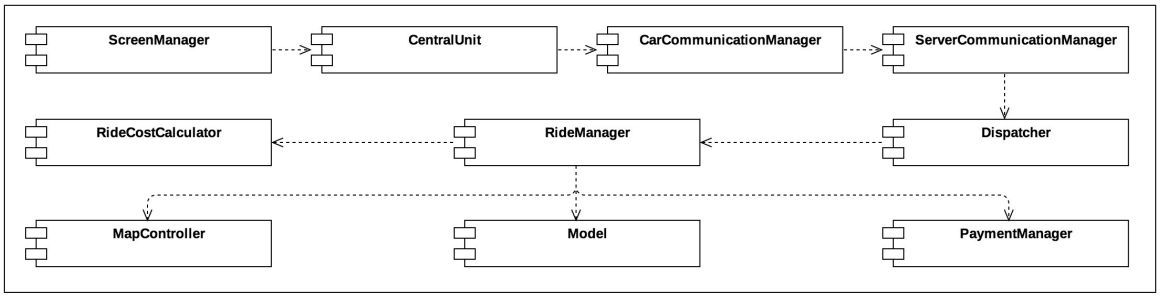

Figure 2.12: Ride Stop Thread

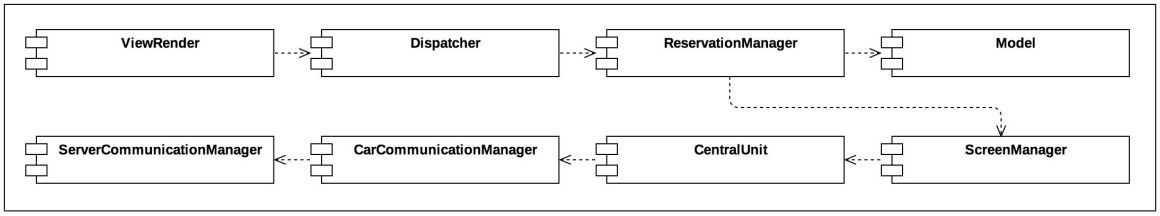

Figure 2.13: Reservation Thread

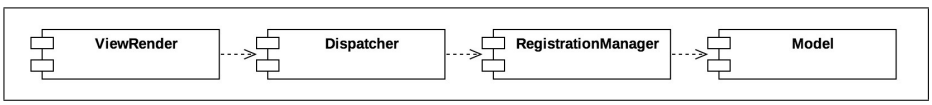

Figure 2.14: Registration Thread

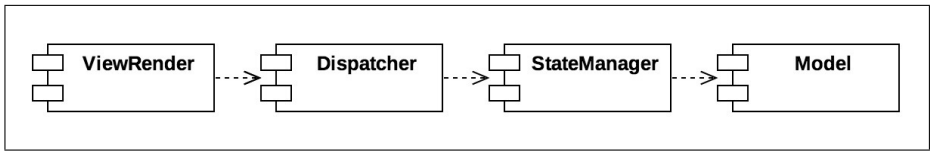

Figure 2.15: Car Plug-in Thread

3 | Individual Steps and Test Description

This section focuses on the interactions between pairs of components that will be progressively integrated. For each pair, a set of tests about the function calls from one component to the other one is provided. This kind of test should cover all the possible calls in order to spot any type of undesirable behaviours just in time. For this reason, each function invocation is here evaluated many times under different circumstances depending on the actual values of the input parameters. Finally, for each such call the desired output is stated. This integration test phase will be organized according to the logical areas division shown in the *Elements to be Integrated* section. For obvious space issues, in the current section only the most significant tests will be proposed, but keep in mind that such verification should be applied to every possible relation between the components.

3.0.1 Management Area

Dispatcher → ReservationManager

ManageNewReservation(username,carID)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	A new reservation for the given username is associated to the specified car

Dispatcher → RideManager

StartRide(username,carID)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The RideManager registers that a new ride associated to the given username and car has started

Dispatcher → RideManager

RideParams(saveMoneyOpt,finalDest)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The RideManager registers whether the user has enabled the Money Saving Option and his final destination

Dispatcher → RideManager

RideStop(peopleOnBoard,position,batteryLevel)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The RideManager registers the number of peopleOnBoard, the final position and the remaining batteryLevel

Dispatcher → RideManager

RidePayment(pluggedIn)	
Valid parameter	The RideManager registers whether the user has plugged the car into the power grid

Dispatcher → StateManager

ModifyCarState(carID,newState)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The StateManager updates the car with the given carID to the newState

3.0.2 Input Area

ViewRender → Dispatcher

DispatchRequest(ReserveRequest)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The request is dispatched to the proper component

ViewRender → Dispatcher

PickUpACar(username, carID)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The ViewRender calls the suitable interface of the Dispatcher passing to it the input data

Device → ViewRender

ReserveACar(username, carID)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumetException
Valid parameters	The user inputs his username and the car he wants to reserve

ViewRender → Dispatcher

ChangeCarState(carID, NewState)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The request of changing the state of the car with the specified carID is sent to the Dispatcher

3.0.3 Ride Area

RideManager → PaymentManager

CheckBalance(username)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The user balance is returned

RideManager → MapController

SearchSuggestedArea(FinalDestination)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The MapController computes the suggested areas where to park the car

RideManager → RideCostCalculator

CalculateCost(peopleOnBoard,position,batterylevel)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The RideCostCalculator computes the total cost of the ride starting from the following input parameters: the number of people on board, the final position of the car, the final battery charge level

RideManager → MapController

ChechPositon(MyPosition)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The MapController checks the position of the car

3.0.4 CarCommunication Area

ServerCommunicationManager → CarCommunicationManager

ReceiveReservation(ExpiringTime)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The CarCommunicationManager receives the reservation and the expiring time for it

CarCommunicationManager → ServerCommunicationManager

ReceiveRideStart(SaveMoneyOpt,FinalDest)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The CarCommunicationManager notifies the ServerCommunicationManager that the ride is starting with indications about the preferences of the user

ServerCommunicationManager → CarCommunicationManager

CommunicateParkArea(AreaPosition)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The ServerCommunicationManager communicates to the CarCommunicationManager the predefined position of the area where the user can park the car

CarCommunicationManager → ServerCommunicationManager

RideStop(carID,peopleOnBoard,position,batteryLevel)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The CarCommunicationManager notifies that the user has ended the ride on the specified car. Information about the peopleOnBoard, the final position and the batteryLevel are also provided

ServerCommunicationManager → CarCommunicationManager

SendCost(Cost)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The ServerCommunicationManager sends the cost of the ride to the CarCommunicationManager

CarCommunicationManager → ServerCommunicationManager

SendPlugInTimeout(PluggedIn)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The CarCommunicationManager indicates if the user has plugged the car into the power grid

3.0.5 Car Area

CentralUnit → CarCommunicationManager

SendBackRideStart(SaveMoneyOpt,FinalDestination)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The CentralUnit sends to the CarCommunicationManager the preferences of the user in terms of the money saving option and the final destination of the ride.

CarCommunicationManager → CentralUnit

StoreParkPosition(AreaPosition)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The CarCommunicationManager sends to the CentralUnit the position of the area where the user can park

CentralUnit → ScreenManager

DisplayParkPosition(Position)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The ScreenManager displays on the screen the position on the map where the user can park to obtain special discount

CentralUnit → CarCommunicationManager

HandleStop(CarID,PeopleOnBoard,Position,BatteryLevel)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The CarCommunicationManager receives from the CentralUnit all the data that have to be passed the to system in order to properly manage the end of the ride

CentralUnit → CarCommunicationManager

PlugInTimeout(PluggedIn)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The CentralUnit notifies the CarCommunicationManager whether the user has plugged the car into the power grid in time

3.0.6 Render Area

MapController → ViewRender

ShowAvailableCars(position,range)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	A webpage with the available cars within the range of distance from the position specified is displayed by the ViewRender

LogInController → ViewRender

ShowMainPage(username)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The Main Page of the specified user is displayed by the ViewRender

RideManager → ViewRender

AbortPickUp(errorMsg)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	A webpage with the critical error is displayed by the ViewRender

3.0.7 Data Area

RideManager → Model

ChangeCarState(carID,newState)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The state of the car with carID is set to newState on the database

RegistrationManager → Model

InsertNewUser(credentials,username,licenseNumber,email,paymentinfo	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	A new record for a new user is created in the Model

LogInManager → Model

FindUser(username,password)	
Input	*Result*
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The LogInManager checks that user is already registered

4 | Tools and Test Equipment Required

4.1 Tools

Testing tools are important because they can automatize and speed up the testing of the system; the used tools that will be presented later in this section, are then useful to properly test firstly the single components and secondly how they integrate and communicate with each other in order to form the complete system and perform all the operations. Considering that the chosen technology for the implementation of webapplication is JavaEE, the following testing tools are taken into account:

- Mockito and JUnit are used in the first step of this process because, even though they are mainly designed for unit testing, it is possible to exploit all the dependencies and interactions for each couple of components, as done both in the bottom-up phase integration both in the thread analysis.
- Arquillian framework because it let us write test cases against Java Containers and also to check dependency injection of components and transaction control (for example for the Data macroarea that involves interactions with the DBMS).

4.2 Test Equipment

In order to properly execute the whole integration testing phase on the system, a set of equipment is required to carry out both the testing on the purely software part (that will be deployed on the server) and the part which will be embedded into the car (that includes the on-board touch screen, the central unit and the specific component devoted to the communication with the server).
In particular, it is fundamental to have:

- at least a device for each type (personal computer, tablet and smart-phone) that will be used by the user to access the system functionalities and by the employees of the car-sharing company.
- the application server, properly configured and already working with the webapplication deployed on it.
- the DBMS already initialized (with all the tables required) and running on a specific server machine.
- both the external components (that is the MapService and the PaymentSystem) must be available through their APIs.

- a fully working prototype of the subsystem that will be installed in each car, complete with the communication part.
- a fully working prototype of a charging station that will be available in each charging area.

For a better and real testing context it would better to have a complete real car and charging station too. However, while for testing the system for the first time it is fine to have only a simplified model of both the car and the charging station, when the testing is in an advanced phase real cars and charging stations should be used to have a better feedback from the system.
In conclusion, to have a simple environment to properly test the overall system, all these elements are the minimum that is required; after having checked that all the functionalities are working properly for this simple case, it is possible to add more components (for example more devices, cars and/or charging stations) to simulate a more realistic case that let also check the performance and the reliability of the developed system.

5 | Program Drivers and Data Test Required

5.1 Program Drivers

Throughout the bottom-up testing phase several drivers that simulate the function invocations among couples of components are needed. Here is the list of the drivers that will be taken into account:

- **ReservationManager Driver**: it is used to simulate methods that are used in the reservation phase.
- **RegistrationManager Driver**: it is designed in order to expose methods with which a user can register.
- **StateManager Driver**: it tests functions that are used to change the state of a car.
- **LogInManager Driver**: it calls interfaces to test the login functionality.
- **RideManager Driver**: it tests the methods that are used to manage the ride.

These drivers are used to simulate the communication between each component of the *Management Area*

- **RideCostCalculator Driver**: it tests methods for the computation of the cost of a ride.
- **ServerCommunicationManager Driver**: This driver on the server side simulates the calls between the server and the car.
- **CarCommunicationManager Driver**: This driver checks the methods that are used between the car and the server.
- **Model Driver**: It is used to invoke methods that are linked with the **DBMS**

As mentioned in the *Entry Criteria* section, there is no need to build a **Dispatcher Driver**, a **PaymentManager Driver** or a **MapController Driver** because they are already fully developed.

5.2 Data Test Required

The integration testing is carried out by using several different instances for the input data of the components to be tested. This is necessary in order to evaluate the behaviour of the system both when the input data are consistent and when they are not. In the former case, a semantically correct output is expected and its result can be checked while in the latter case the focus is on the policy adopted by the system to manage the errors. This approach, which contributes to the robustness of the system, is achieved by testing the methods invocations between the components with a set of possible inputs. Consequently, the set of data input that are used in the testing of a method invocation includes for each parameter:

- A null instance: this is the most common error and the system policy involves that when such an error is detected a NullArgumentException is raised.
- An empty on unknown parameter (including type mismatch) : for this type of error the system policy involves that an InvalidArgumentException is raised.
- A valid parameter: in this case a correct output is expected.

This approach towards the input data management has been used in the definition of the test cases (see the *Individual Steps and Test Description* section).

6 | Effort Spent

In order to complete this document, each author worked for 20 hours.