



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 15<sup>th</sup>, 2017*  
*Version 1.0*

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Revision History . . . . .	1
1.2	Purpose and Scope . . . . .	1
1.3	List of Definitions and Abbreviations . . . . .	1
1.4	List of Reference Documents . . . . .	2
<b>2</b>	<b>Integration Strategy</b>	<b>3</b>
2.1	Entry Criteria . . . . .	3
2.2	Elements to be Integrated . . . . .	3
2.3	Integration Test Strategy . . . . .	7
2.4	Sequence of Component Integration . . . . .	7
2.4.1	Bottom-Up Integration Strategy . . . . .	8
2.4.2	Thread Integration Strategy . . . . .	8
<b>3</b>	<b>Individual Steps and Test Description</b>	<b>11</b>
3.0.1	Management Area . . . . .	11
3.0.2	Input Area . . . . .	12
3.0.3	Ride Area . . . . .	13
3.0.4	CarCommunication Area . . . . .	14
3.0.5	Car Area . . . . .	16
3.0.6	Render Area . . . . .	17
3.0.7	Data Area . . . . .	17
<b>4</b>	<b>Tools and Test Equipment Required</b>	<b>19</b>
<b>5</b>	<b>Program Stubs and Test Data Required</b>	<b>20</b>
<b>6</b>	<b>Effort Spent</b>	<b>21</b>

# List of Figures

2.1	Input Area . . . . .	4
2.2	Management Area . . . . .	4
2.3	Render Area . . . . .	5
2.4	Ride Area . . . . .	5
2.5	Data Area . . . . .	6
2.6	CarCommunication Area . . . . .	6
2.7	Car Area . . . . .	7
2.8	Bottom-up test example . . . . .	8
2.9	Bottom-up notation . . . . .	8
2.10	Login Thread . . . . .	8
2.11	Ride Start Thread . . . . .	9
2.12	Ride Stop Thread . . . . .	9
2.13	Reservation Thread . . . . .	9
2.14	Registration Thread . . . . .	10
2.15	Car Plug-in Thread . . . . .	10

# 1 | Introduction

## 1.1 Revision History

## 1.2 Purpose and Scope

## 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 v1.0

## 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
- No specific constraints on the **ViewRender** for the first structural testing phase which involves operations that don't integrate the UI
- Components like the **Payment Manager** and the **MapController** that are to be linked with third-part components (**Payment System** and **MapService**) must have 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 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.

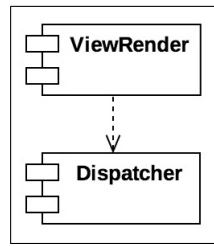


Figure 2.1: Input Area

- **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.

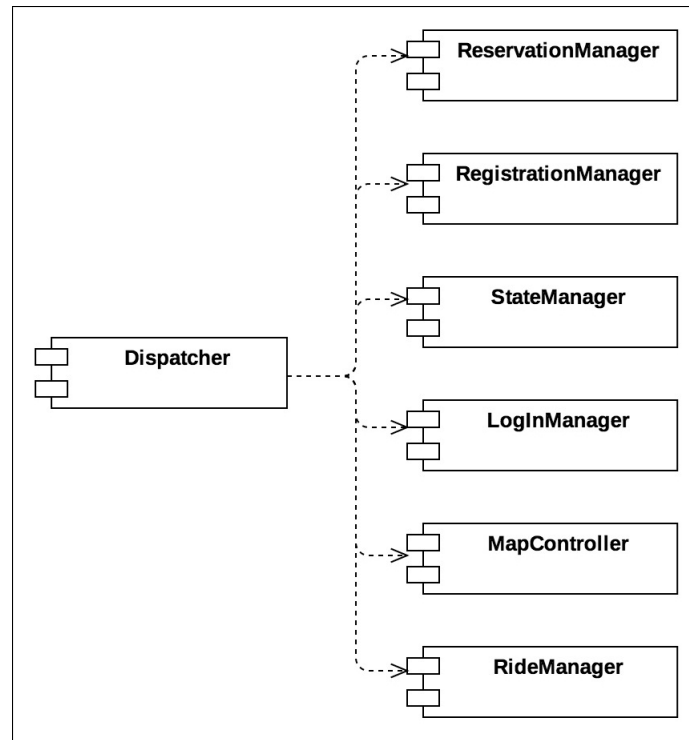


Figure 2.2: Management Area

- **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.

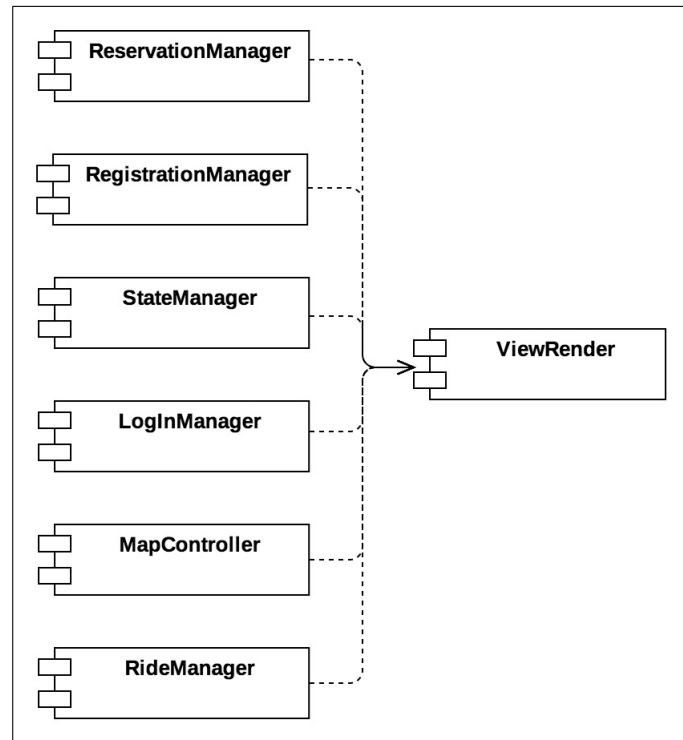


Figure 2.3: Render Area

- **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.

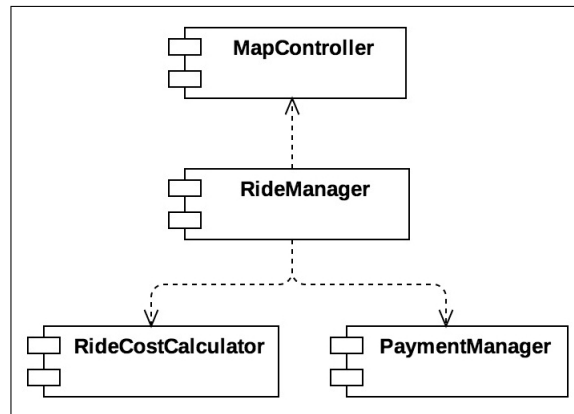


Figure 2.4: Ride Area

- **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.



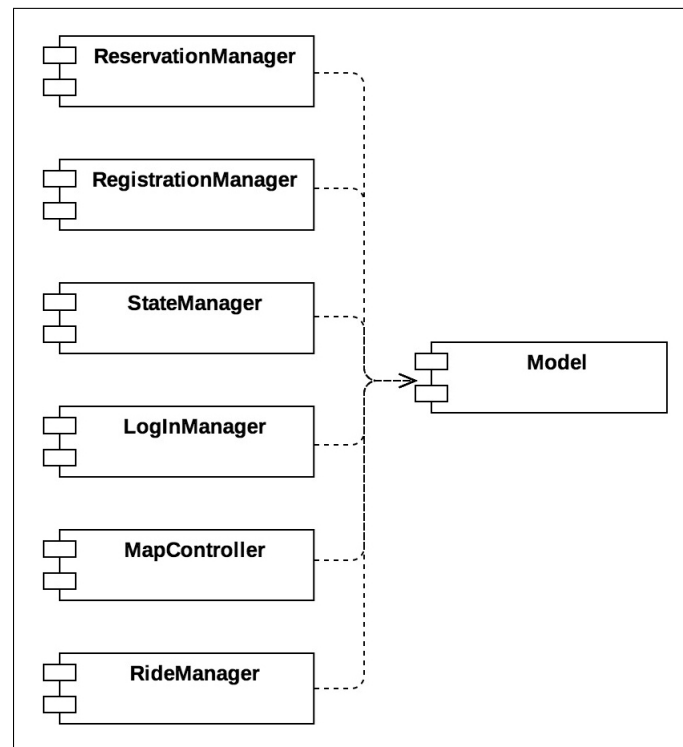


Figure 2.5: Data Area

- **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.

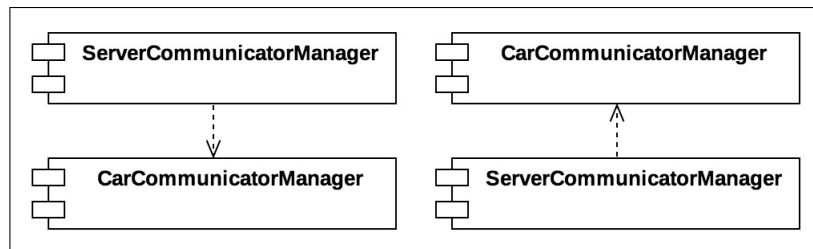


Figure 2.6: CarCommunication Area

- **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.

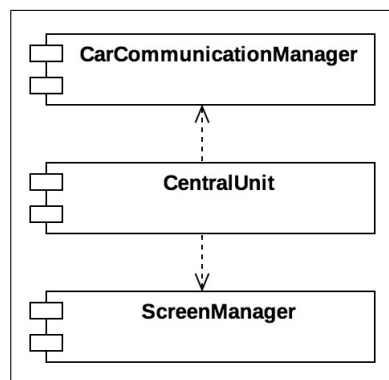


Figure 2.7: Car Area

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. Finally, a remark on the external components (*MapService*, *Payment System* and *DBMS*): they are already available for integration testing and they only require a sufficient level of completion of the internal components to be actually tested.

## 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 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.

A final remark on the external components: the **MapService**, the **Payment System** and the **DBMS** components are already fully developed and in a bottom-up perspective they can be tested immediately using the corresponding system components as proper drivers.

## 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 one must have performed the bottom-up testing 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:

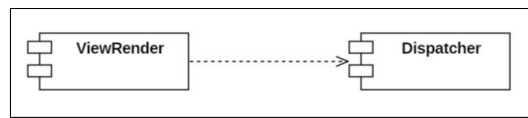


Figure 2.8: Bottom-up test example

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

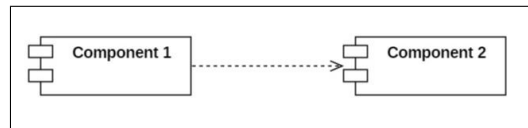


Figure 2.9: Bottom-up notation

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:

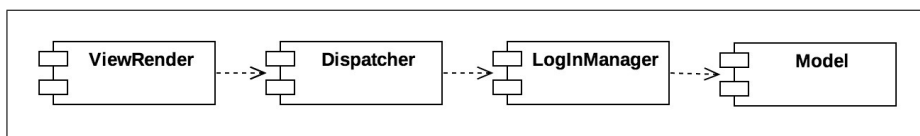


Figure 2.10: Login Thread

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 reasonings.

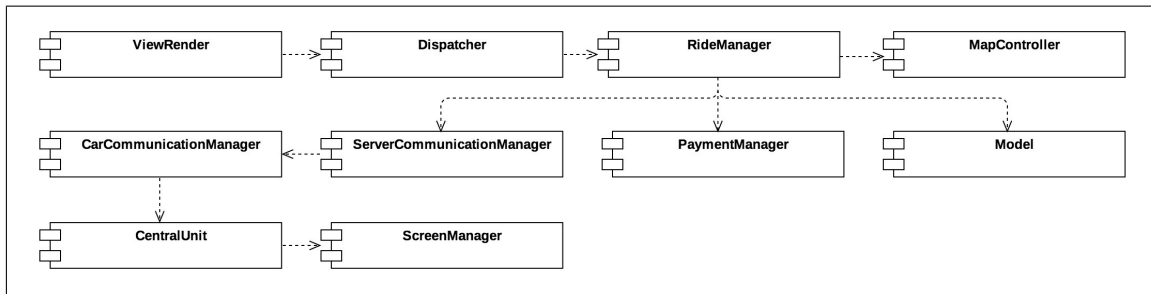


Figure 2.11: Ride Start Thread

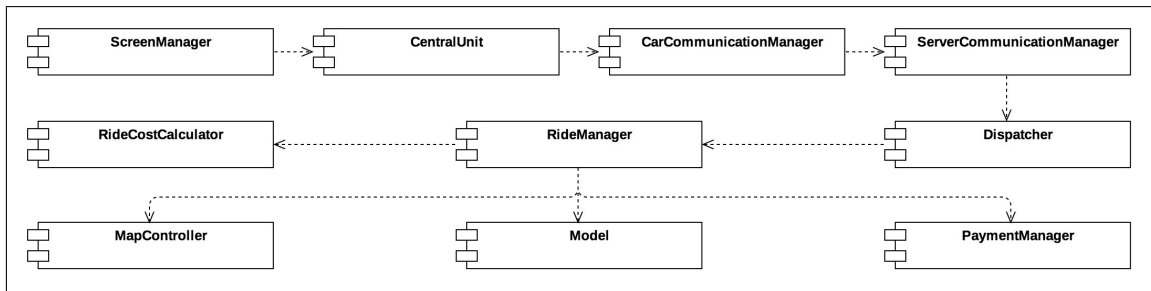


Figure 2.12: Ride Stop Thread

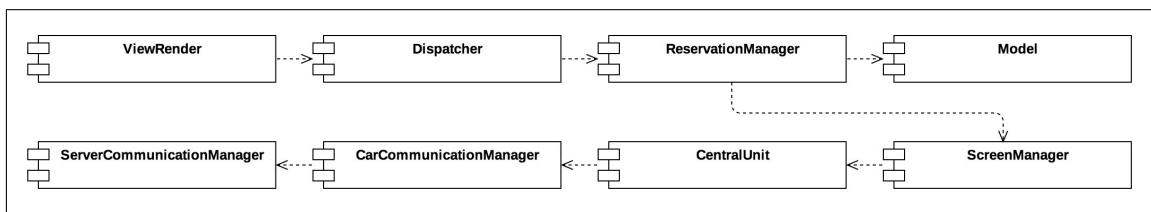


Figure 2.13: Reservation Thread

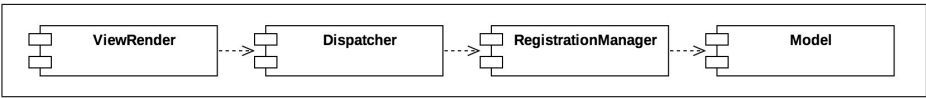


Figure 2.14: Registration Thread

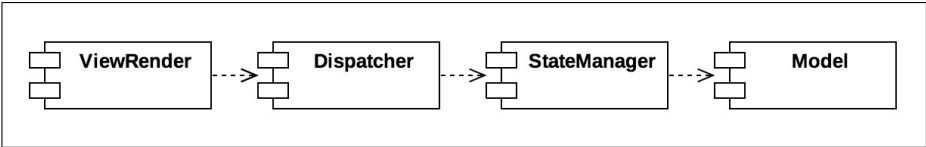


Figure 2.15: Car Plug-in Thread

## 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)	
<i>Input</i>	<i>Result</i>
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)	
<i>Input</i>	<i>Result</i>
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**

<b>RideParams(saveMoneyOpt,finalDest)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>RideStop(peopleOnBoard,position,batteryLevel)</b>	
<i>Input</i>	<i>Result</i>
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**

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

**Dispatcher → StateManager**

<b>ModifyCarState(carID,newState)</b>	
<i>Input</i>	<i>Result</i>
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**

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

**ViewRender → Dispatcher**

<b>PickUpACar(username, carID)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>ReserveACar(username, carID)</b>	
<i>Input</i>	<i>Result</i>
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameters	The user inputs his username and the car he wants to reserve

**ViewRender → Dispatcher**

<b>ChangeCarState(carID, NewState)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>CheckBalance(username)</b>	
<i>Input</i>	<i>Result</i>
A null parameter	NullArgumentException
An empty or unknown parameter	InvalidArgumentException
Valid parameter	The user balance is returned

**RideManager → MapController**

<b>SearchSuggestedArea(FinalDestination)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>CalculateCost(peopleOnBoard,position,batterylevel)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>ChechPositon(MyPosition)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>ReceiveReservation(ExpiringTime)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>ReceiveRideStart(SaveMoneyOpt,FinalDest)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>CommunicateParkArea(AreaPosition)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>RideStop(carID,peopleOnBoard,position,batteryLevel)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>SendCost(Cost)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>SendPlugInTimeout(PluggedIn)</b>	
<i>Input</i>	<i>Result</i>
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)	
<i>Input</i>	<i>Result</i>
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)	
<i>Input</i>	<i>Result</i>
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)	
<i>Input</i>	<i>Result</i>
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)	
<i>Input</i>	<i>Result</i>
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**

<b>PlugInTimeout(PluggedIn)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>ShowAvailableCars(position,range)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>ShowMainPage(username)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>AbortPickUp(errorMsg)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>ChangeCarState(carID,newState)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>InsertNewUser(credentials,username,licenseNumber,email,paymentinfo)</b>	
<i>Input</i>	<i>Result</i>
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**

<b>FindUser(username,password)</b>	
<i>Input</i>	<i>Result</i>
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

## 5 | Program Stubs and Test Data Required

## 6 | Effort Spent