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

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 we have to estimate how much the components are completed in order to use their functionalities.

- **Dispatcher** must have been fully completed in order to manage requests
- Controllers like **ReservationManager**,**RegistrationManager**, **StateManager**, **LogInManager** and **RideManager** can have functionalities that are to be completed, but must have been developed in order to provide basis methods
- **ViewRender** can be forgotten for the first structural test that doesn't integrate the UI
- Components like the **Payment Manager** and the **MapController** that interface with third-part components (**Payment System** and **MapServce**) must have fully developed in order to use external APIs

Secondly, Requirements Analysis and Specification Document and Design Document must have been written.
Thirdly, components must be individually tested (unit testing is not part of ITPD) in order to ensure that bugs from future tests between parts will be caused by the iteration among these and not by an 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.

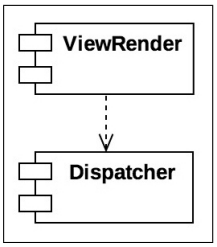

Figure 2.1: Input Area

- **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.

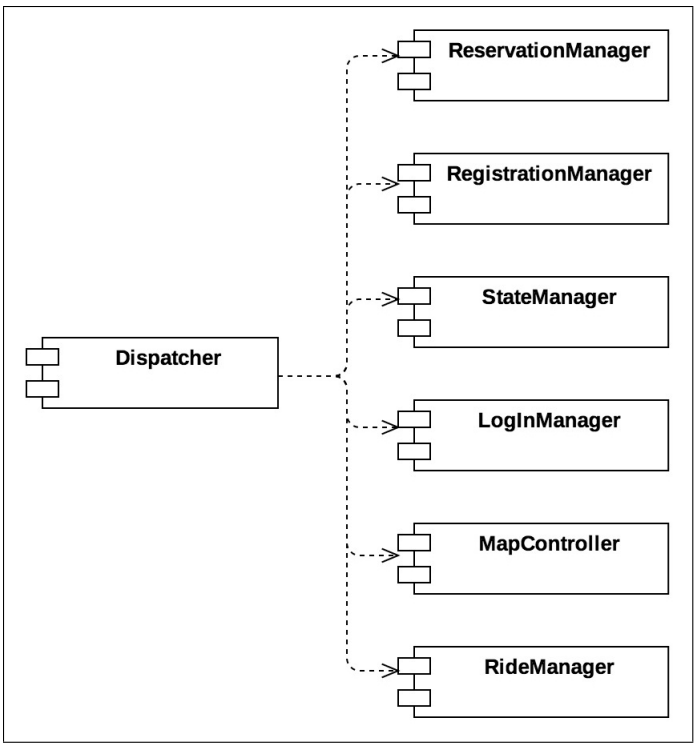

Figure 2.2: Management Area

- **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.

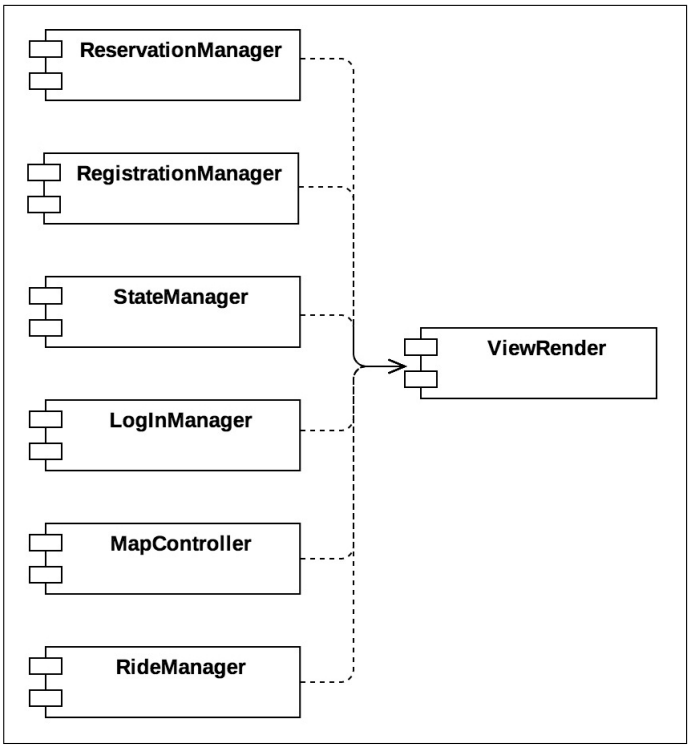

Figure 2.3: Render Area

- **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.

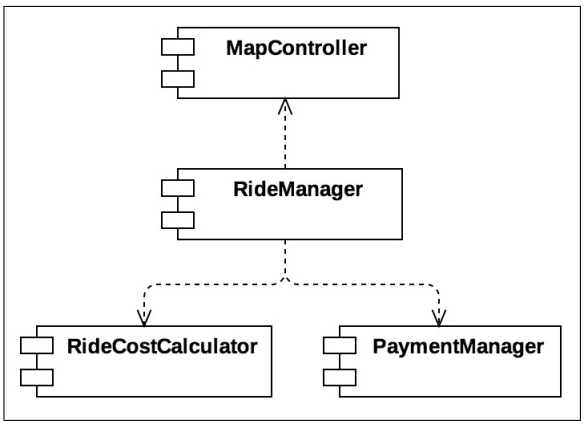

Figure 2.4: Ride Area

- **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.

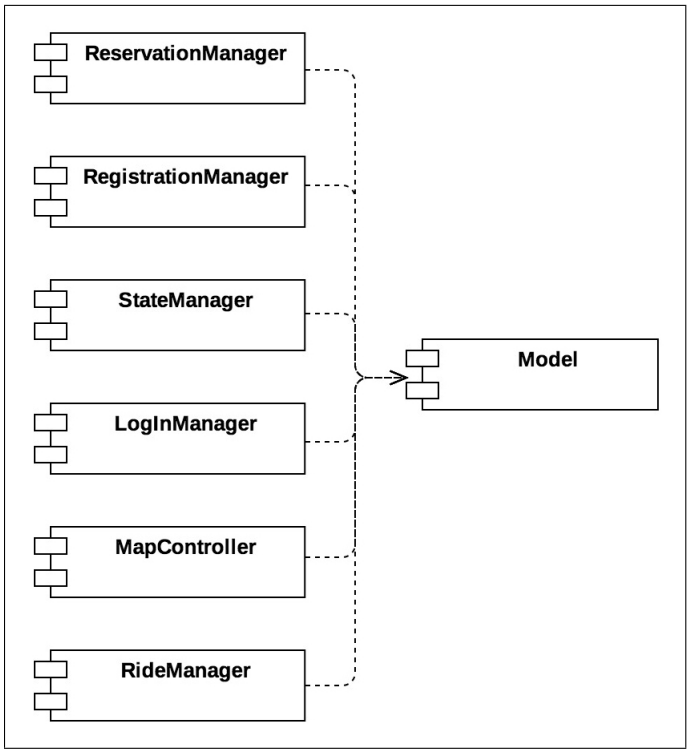

Figure 2.5: Data Area

- **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.

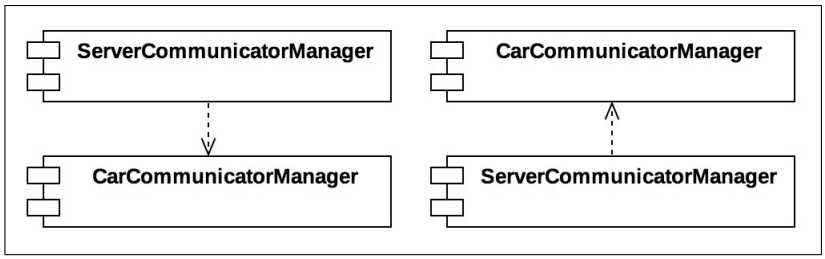

Figure 2.6: CarCommunication Area

- **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.

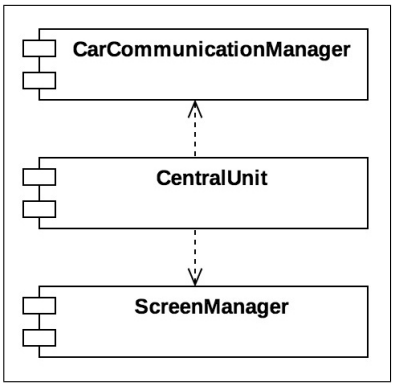

Figure 2.7: Car Area

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 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 the 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

2.4.1 Software Integration Sequence

2.4.2 Subsystem Integration Sequence

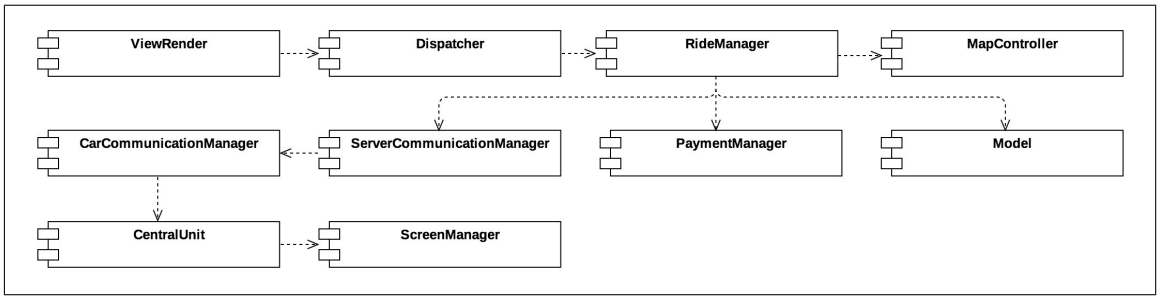

Figure 2.8: Ride Start Thread

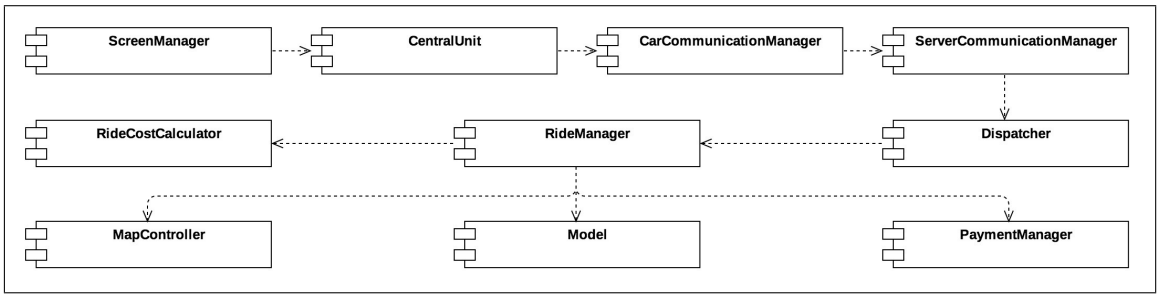

Figure 2.9: Ride Stop Thread

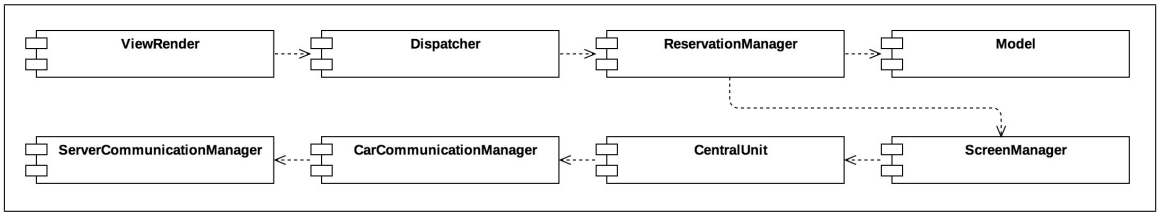

Figure 2.10: Reservation Thread

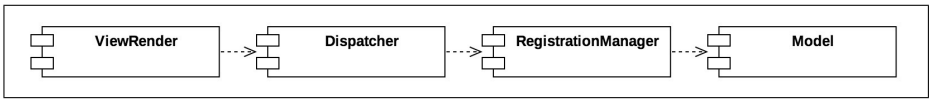

Figure 2.11: Registration Thread

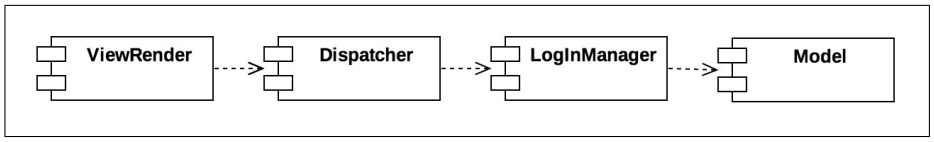

Figure 2.12: Login Thread

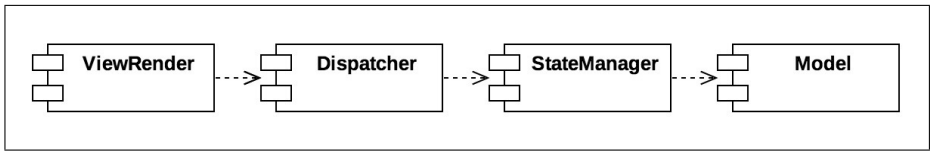

Figure 2.13: Car Plug-in Thread

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

5 | Program Stubs and Test Data Required

6 | Effort Spent