Figure 1: Politecnico di Milano

Integration Testing Document
Version 1.0

Emanuele Ricciardelli (mat. 875221)
Giorgio Tavecchia (mat. 874716)
Francesco Vetró (mat. 877593)

January 15, 2017

Contents

1 Introduction

1.1 Purpose

This document represents the Integration Testing Plan Document for PowerEnJoy service. The purpose of the testing is to verify and ensure that the system operation is correct and complies with all the requirements expressed in the previous documents (RASD and DD). All components of the system will have to comply with the constraints and functionalities for which they were designed, without showing unexpected behavior. The purpose of this document resides therefore in specifying the methods and the time needed to complete a full testing, both from the point of view of the individual components, both in their interactions. In order to provide a detailed description of the future testing, in the following paragraphs it will be defined:

- A list of the subsystems and their subcomponents of the PowerEnJoy system-to-be that will be involved in the testing activity;
- The criteria that must be met by the project status before integration testing of the outlined elements may begin;
- A description of the integration testing approach and the rationale behind the specific choose;
- The sequence of component and function integration;
- A description of individual steps and test description: for each step of integration, we will define a test design with related expected results for each test;
- A list of tools, test equipment and test data required.

1.2 Definitions and Abbreviations

- RASD: Requirement Analysis and Specification Document;
- DD: Design Document;

For all other definitions, please refer to previous documents listed in the next paragraph

1.3 Reference Documents

- Specification document: Assignments AA 2016-2017.pdf
- RASD Version 1.1
- DD Version 1.1
- Integration testing example document.pdf

2 Integration strategy

2.1 Entry criteria

With reference to the past documents and in particular to the design document, it is necessary to specify how the individual components that make up the system require to achieve a given percentage of completion with respect to the features for which they were designed. Especially, with respect to the High Level Component view of the Design Document:

- 100% for the DBMS component: in order to allow a correct testing not only of the database structure itself, but also of the various components of the system that base their operation on access to the database;
- 60% of the Client components: many of the functions covered by the client involve the GUI and have no priority over interaction with the system functionality;
- 90% of the Application Server;
- 90% of the Car Sensor Subsystem: in order to ensure a correct testing of the application server, given the close interaction.

These criteria must be met before the start of the Integration testing. Different percentages want to reflect the order of integration, placed its focus on certain components, and otherwise in the background some secondary aspects such as the graphical user interface on the client.

2.2 Elements to be integrated

In order to specify the elements to be integrated, we refer not only to High Level Component view contained in the DD, but also to the Component view that allows us to identify components at a lower level and the relationships between them . As specified in the Design Document, the components of our system have been designed in such a way as to ensure a high level of modularity, by separating the different functionality into distinct components, thus also facilitating testing. For this reason, right from the Component view it is possible to note how different components are strongly dependent on one another in order to achieve more complex functionality of the system-to-be. The recognition of these complex functionality allows us to split our system into a number of subsystems, addressed respectively to the access management and registration system (*Account Subsystem*), management of payments (*Payment subsystem*), reservation management and everything that concerns the sphere of rental (*Reservation subsystem*) and finally management of all the sensors and the exchange of information between cars and system and everything related to the maintenance (*Car / Sensors subsystem*). In particular, the functions relating to the *Reservation subsystem* will involve the integration of:

- Reservation Controller;
- Ride Controller (integrated with Google Maps API);
- Notification Helper (with related gateway);

- Communication Manager;
- Request Manager;
- Data Access Manager.

With respect to the *Account subsystem*, it is concentrated only in the *Login / Registration Controller* that later, through the *Communication Manager*, will interface with the other components of the PowerEnJoy system. In detail:

- Login/Registration Controller;
- Communication Manager;
- Request Manager;
- Notification Helper (with related gateway);
- Data Access Manager.

The same reasoning can be made for the *Payment subsystem* and the respective *Payment Controller*. In detail:

- Request Manager;
- Payment Controller;
- Communication Manager;
- Data Access Manager;
- Payment Gateway;

Finally, the *Car / Sensors subsystem* in order to perform all its functionalities will need to integrate:

- Cars Controller;
- Communication Manager;
- Collector Data (with related gateway);
- Maintenance Gateway;
- Request Manager;
- Data Access Manager;

It is important to underline that the components *Notification Helper Request Manager* and *Communication Manager* are significant for all subsystems and therefore their integration. This means that if on the one hand their presence ensures greater flexibility in the management of functions, on the other hand emphasizes the importance of their proper functioning in order to ensure the fulfillment of all the functionality of the overall system. Given the above definitions of subsystems, our integration testing will proceed initially considering a single subsystem at a time, following finally with the integration of the same until the establishment of the system in its entirety.

2.3 Integration testing strategy

The approach suggested in the process of integration testing is bottom-up. This choice is based on considerations relating to the features covered by our system: in particular, as has been documented in the DD, it has been chosen to decompose the system into a series of components, each with features related to a particular field. The functionalities that the system will perform are derived from joint work of these components. From this observation it can be seen that the choice of a bottom-up reflects the needs of the testing related to our system. In this way we can favour the ability to identify problems at an early stage of development of the components and, in case, to react quickly. Later, after having tested the elementary interactions between components and subsystems, we can proceed with the integration to a higher level. It will therefore be necessary to develop a set of drivers to drive the testing, but given the simultaneous development of the overall system, this will prove to be quite natural. Once the system will be completely formed, we shall provide the implementation of a series of System Testing in order to verify not only the achievement of the functional requirements, as a whole, but also of the non-functional aspects such as the loading capacity as a function of the expected values.

2.4 Sequence of component / function integration

In this section we will provide a description of the order of integration testing of components and subsystems of PowerEnJoy system-to-be.

2.4.1 Software integration sequence

Given the previous subdivision into subsystems and starting from the *Reservation subsystem*, we now define the sequence in which the components will be integrated for each subsystem and then the sequence related to the integration of the subsystems themselves.

Reservation subsystem
Since many features covered by this system need an interaction with the database (performing queries or updating tables), the very first components to be integrated are the **Data Access Manager** and the **DBMS**.

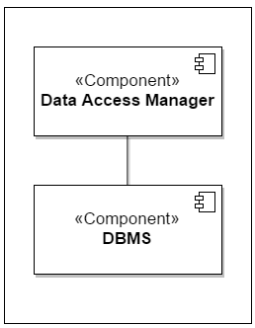

Figure 2: Integration sequence - 1 - Data Access Manager, DBMS

As will later be made clear, this first integration will also be required for other subsystems, but will not be repeated if it is unnecessary; otherwise it will be repeated the possible variants of integration between these first components and the surrounding ones. With respect to the bottom-up approach, the next integration step will cover the correct communication between the database and the **Communication Manager**.

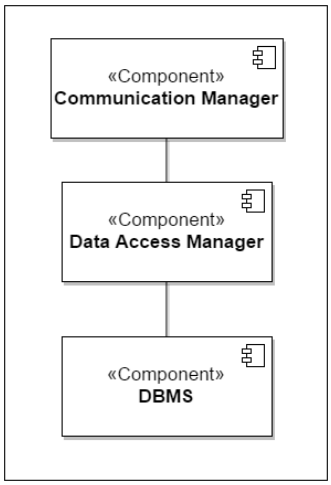

Figure 3: Integration sequence - 2 - Communication Manager & above

We can now proceed to a higher level of integration considering the **Ride Controller**: this component will be integrated with the API provided by the services of google in order to verify the correct use within their own methods, with the "Notification Helper" in order to verify the correct sending of notifications generated to users, and finally with **Communication Manager** which, as described in the DD, is designed to allow (in a flexible and transparent way) the exchange of information between internal components of the system, to verify proper access to the database. At the same level of integration we will introduce the **Reservation Controller** integrated with **Communication Manager** and **Notification Helper** for the same reasons mentioned above.

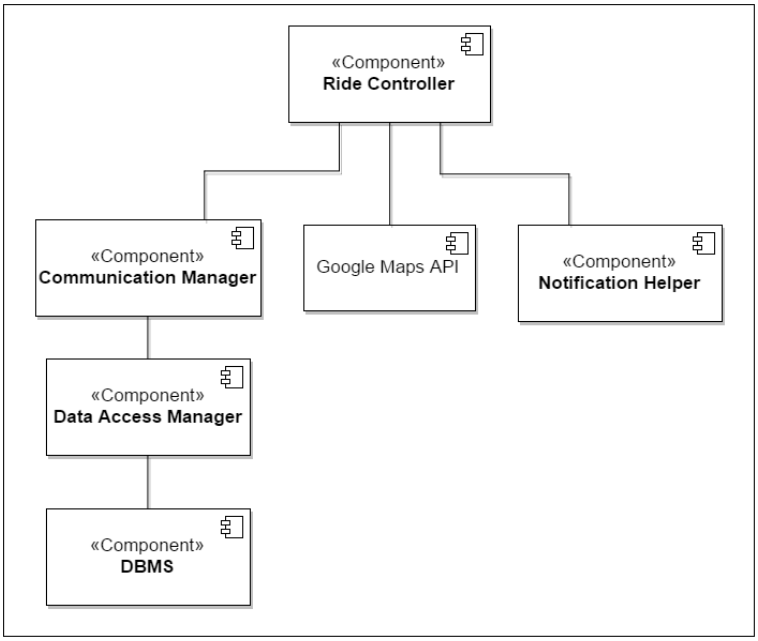

Figure 4: Integration sequence - 3 - Ride Controller, Notification Helper, Google APIs & above

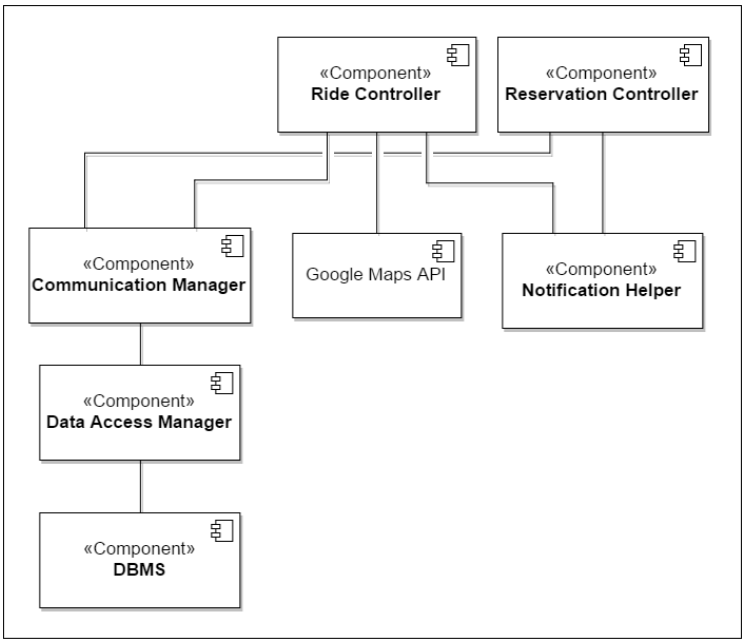

Figure 5: Integration sequence - 4 - Reservation Controller & above

Since different functions performed by the controller such as **Reservation Controller** and **Ride Controller** are performed by request from external clients, and given that the same demands are made through the component **Request Manager**, the next components to be integrated will be the following.

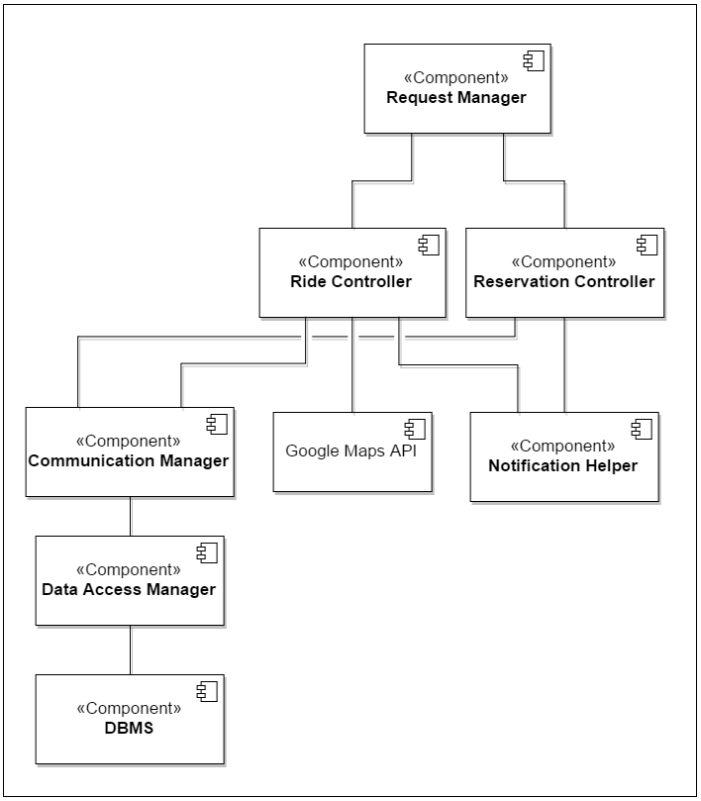

Figure 6: Integration sequence - 5 - Request Manager & above

This will make it possible to test the correct routing of requests towards the selected components and the exchange of information between them. Finally we integrate the Client, to test that all of the functionalities of this subsystem work properly.

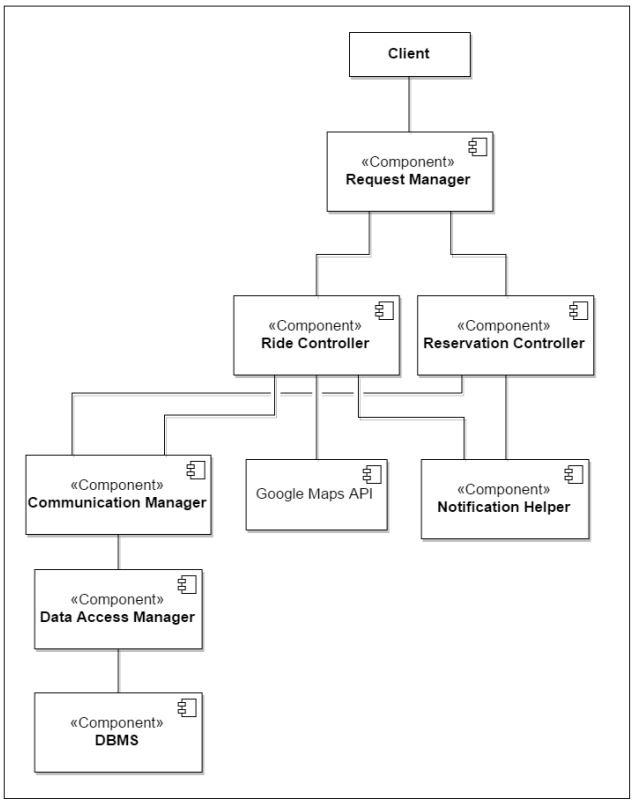

Figure 7: Integration sequence - 6 - Client & above

Account subsystem
The starting point relative to the sequence of integration of this subsystem should be regarding the access to the database, but, as previously expressed, since it was done for the previous subsystem, it is not required to do it again. It is now shown for the sake of completeness to the reader.

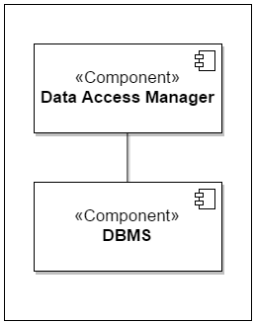

Figure 8: Integration sequence - 1 - Data Access Manager, DBMS

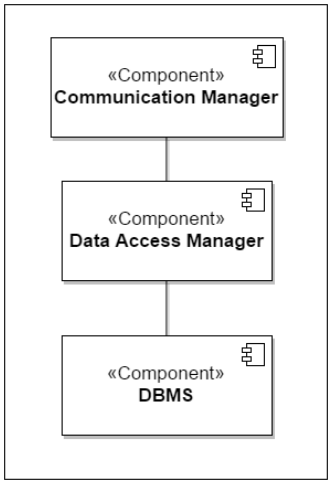

Figure 9: Integration sequence - 2 - Communication Manager & above

We follow now with the integration between the **Login / Registration Controller** and **Notification Helper** to verify, as done in the previous subsystem, the correct forwarding of notifications generated and, at the same time, integrating the **Login / Registration Controller** with **Communication Manager** and **Data Access Manager** in order to test the proper functioning of methods that require database access.

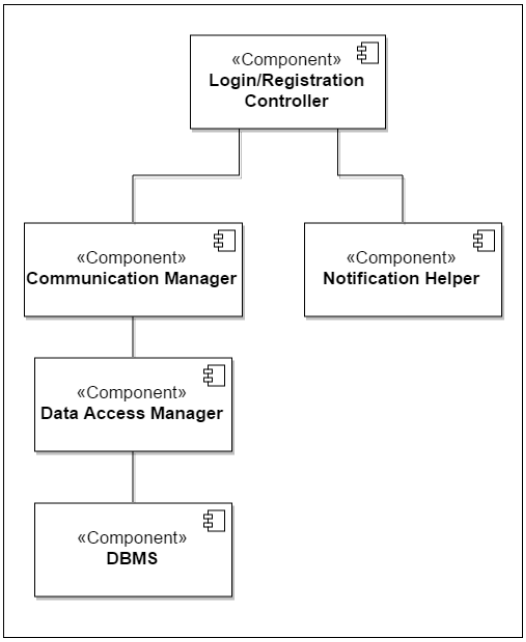

Figure 10: Integration sequence - 7 - Login/Registration Controller, Notification Helper & above

Going up to a higher level of integration, we introduce the **Request Manager** in order to test the correct routing of requests coming from the outside to the component suited to the management of the system accounts. And again we terminate this sequence of integration introducing the Clients to test that all of the functionalities of this subsystem work properly. Thus completing the integration of this subsystem.

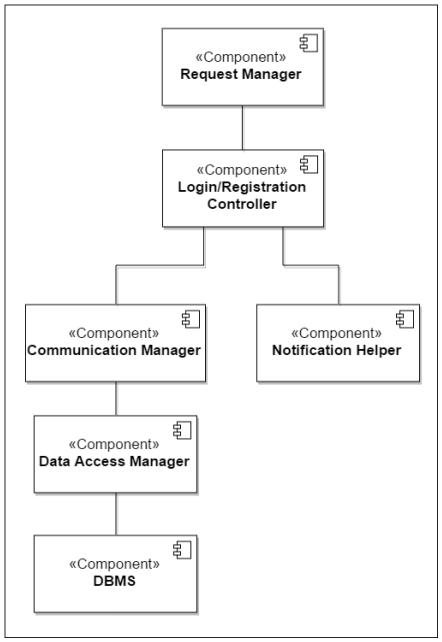

Figure 11: Integration sequence - 8 - Request Manager & above

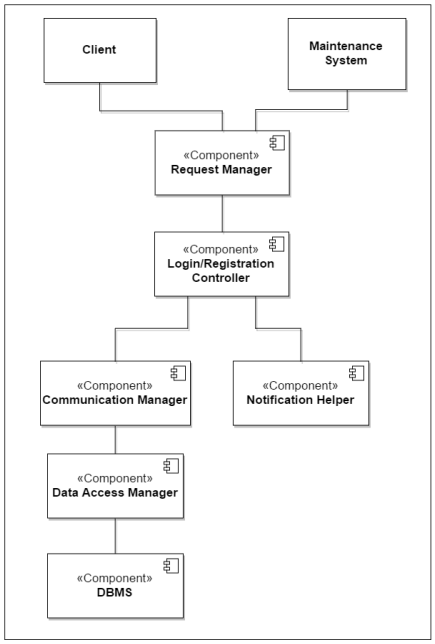

Figure 12: Integration sequence - 9 - Client & above

Payment subsystem
We now proceed in the integration process Considering the *Payment Subsystem*. As before, we start integrating the components related to the access to the database. The first components that need to be integrated are the **Payment**

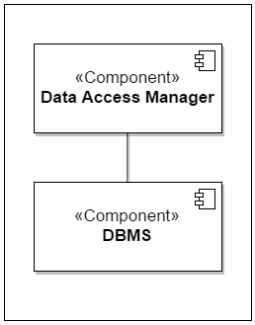

Figure 13: Integration sequence - 1 - Data Access Manager, DBMS

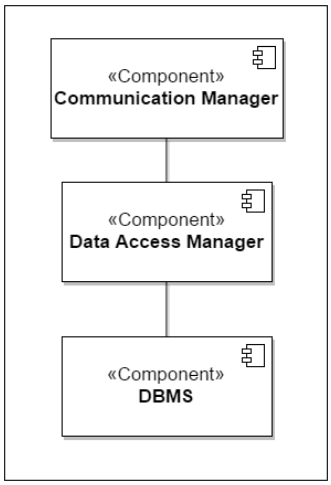

Figure 14: Integration sequence - 2 - Communication Manager & above

Controller and the **Payment Gateway**, this to verify that the results relating to payment functions are correctly sent to the external payment companies, and the integration between **Payment Controller** and **Communication Manager** to test the communication with the database.

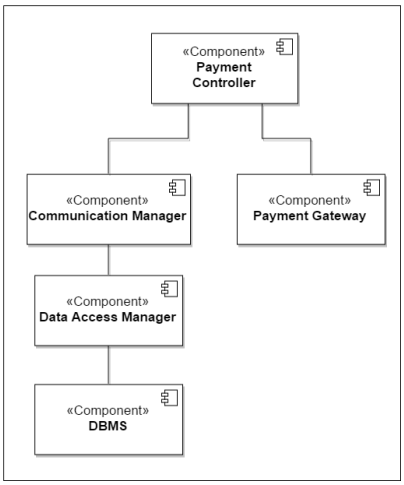

Figure 15: Integration sequence - 10 - Payment Controller, Payment Gateway & above

Followed by the integration with the **Request Manager** to verify the correct forwarding of information requests to the subsystem. As before we terminate this sequence of integration testing the connection with the Client.

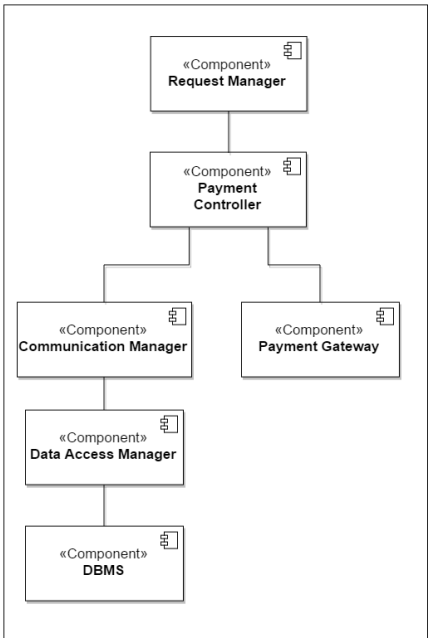

Figure 16: Integration sequence - 11 - Request Manager & above

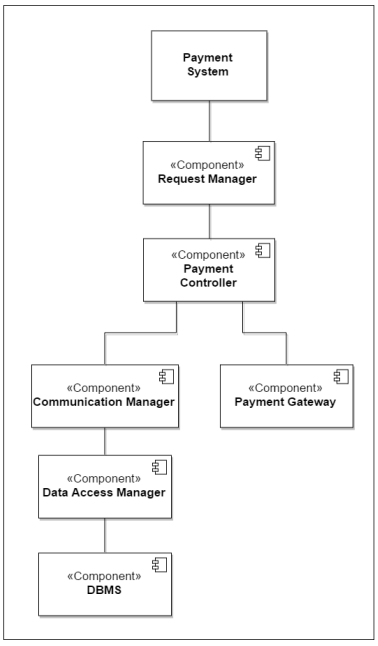

Figure 17: Integration sequence - 12 - Client Payment System & above

Car / Sensors subsystem
The last subsystem to be included in the sequence of integration is the *Car / Sensors subsystem* that will handle all exchanges of information and operations to be performed on vehicles and everything related to the maintenance. The starting point is still represented by the access to database.

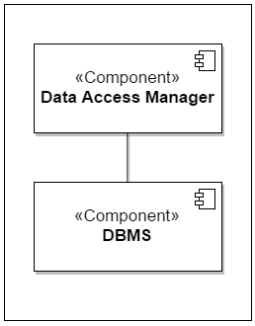

Figure 18: Integration sequence - 1 - Data Access Manager, DBMS

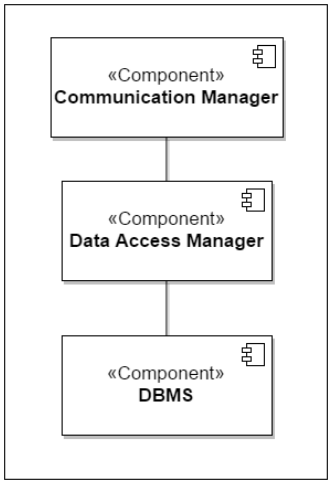

Figure 19: Integration sequence - 2 - Communication Manager & above

As first components to be integrated, we choose **Cars Controller** and **Data Collector**: this choice derives from the importance that covers the proper exchange of information between system and machine, upon which are based a considerable number of other functions related to the **Cars Controller** itself. In addition we integrate our principal component with the **Maintenance Gateway** to test the proper sending of information from the system to the maintenance company and with the **Communication Manager** to verify the correct access with the database.

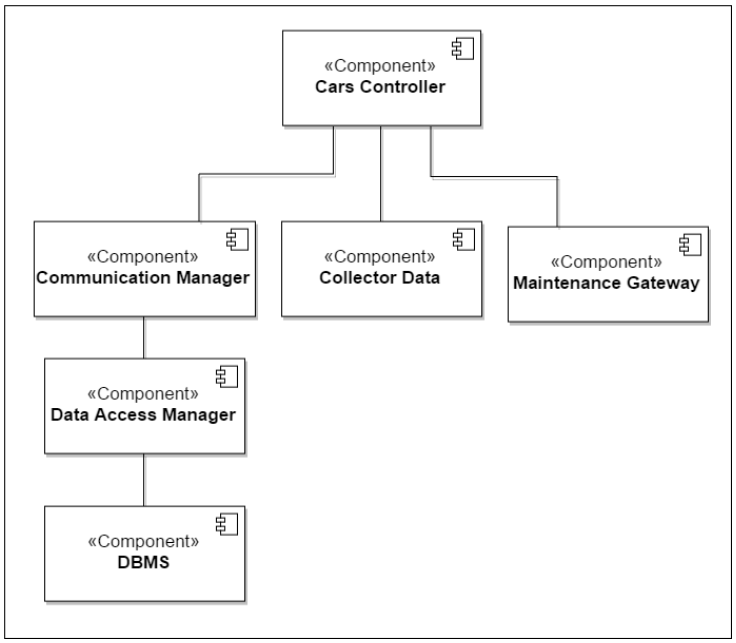

Figure 20: Integration sequence - 13 - Cars Controller, Collector Data, Maintenance Gateway & above

Finally the sequence of integration will end integrating the **Request Manager** and so to verify the correct implementation of functionalities such as reporting a fault to a machine or access by the maintenance to the list of reported cars. As last integration, like in other subsystems, we verify and test the integration with the clients.

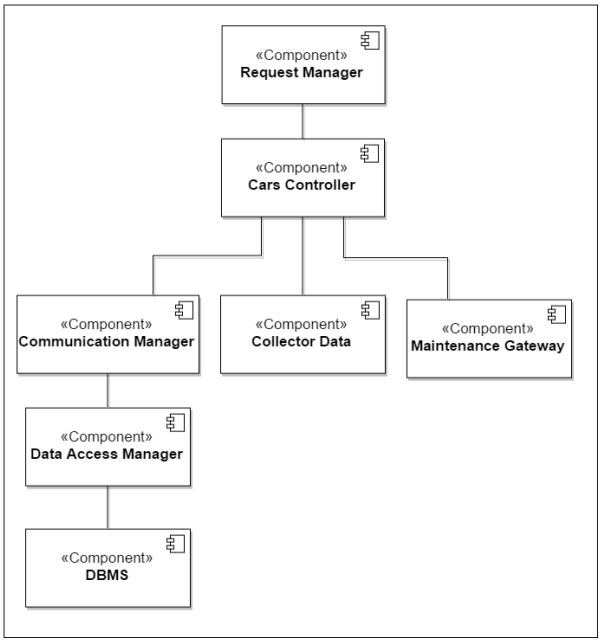

Figure 21: Integration sequence - 14 - Request Manager & above

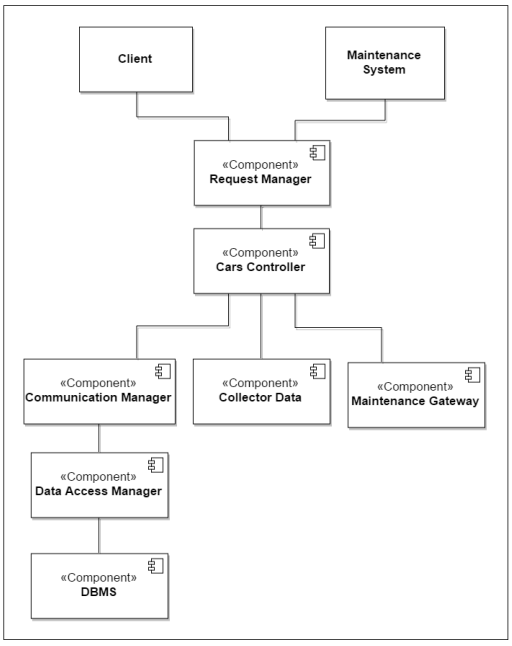

Figure 22: Integration sequence - 15 - Clients & above

Note to the reader: In previous integration sequences it was performed a choice concerning the gateway present in the component view of the DD: we decide to show explicitly only the payment and maintenance gateway. This is because the integration of the omitted gateway is to consider joint with the components to which they are directly linked: For example, the **Sensor Gateway** is considered as a joint to the component **Data Collector** and the gateway relative to the notifications joint with the **Notification Helper**.

2.4.2 Subsystem integration sequence

In order to recreate from the subsystems, considered in the previous section, the overall system, the process of integration will initially proceed by integrating the **Reservation Subsystem** with **Car / Sensor Subsystem** in order to test all of those functions of the first subsystem that require access to the functionality provided by the second.

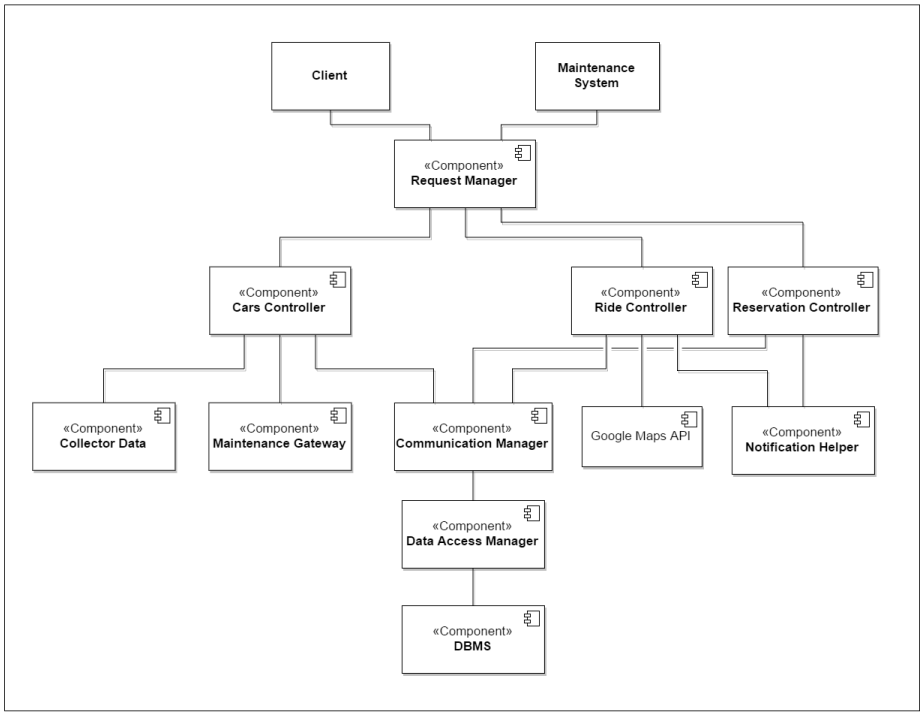

Figure 23: Integration sequence - 16 - Clients & above

It is now integrated the **Account Subsystem**in order to test the integration of external user and system, through all the features that will provide PowerEnjoy to it, excluding those relating to payments that are later tested with the integration of the related subsystem.

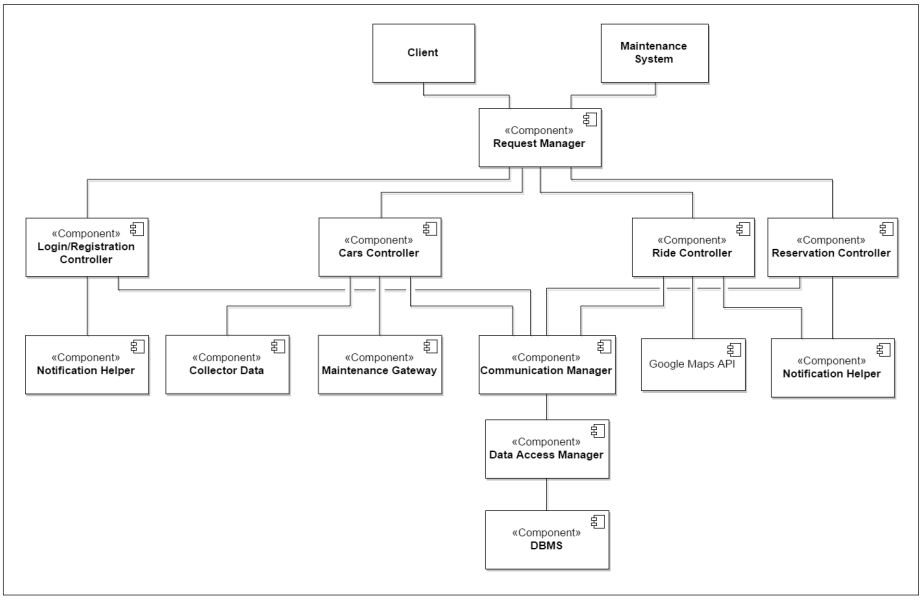

Figure 24: Integration sequence - 17 - Clients & above

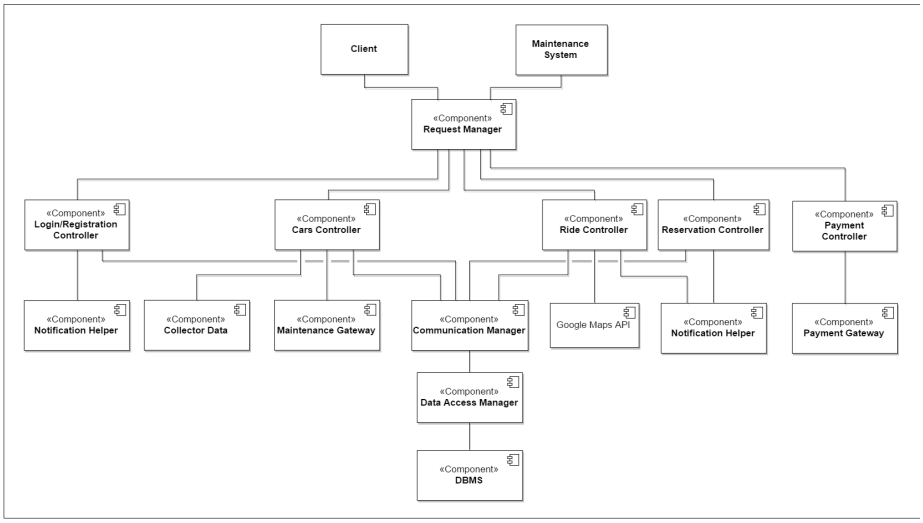

Figure 25: Integration sequence - 18 - Clients & above

3 Individual steps and test description

In this section we would describe, according to the Component Interfaces (see DD at Section 2.6), how the interaction between the various components of your system is tested, using the properly interfaces provided by us in order to get this purpose. The description is made by going through the different subsystems and for each of that, focusing on their working and describing in an accurate way what are the methods used to reach the scope. For each method described, we give a brief description on what are the possible inputs values and the respective effect expectation.

3.1 External calling

With external calling we mean one of various type of Client that we described in the Component View (Client, Maintenance System or Payment System). Each of that, in the request phase, performs and sends a new request to our system. It is processed by the Request Manager. The request type specified below is generic (abstract superclass) and different types of requests exist (Client has ClientRequest, Maintenance has MaintenanceRequest and so on).

3.1.1 Client (generic), Request Manager

This test is done to verify that the request manager can receive correctly an admissible request and that it is able to handle this in order to be dispatched to the correct component.

Table 1: sendRequest(request:Request)

Input	Effect on the system
A null parameter	A NullPointerException is thrown
One of the request type cited above	The request is processed by proper policies, depending on the request type and it is ready to be dispatched.

3.2 Reservation subsystem

3.2.1 Communication Manager, Data Access Manager

This test is made to verify that the Data Access Manager can parse the string received by the Communication Manager as a query and if the component can perform it properly returning the correct instance of Model required by the query submitted. This test is made to verify that the Data Access Manager can parse the string received by the Communication Manager as an updating query and if the component can perform it properly returning the outcome of the operation.

Table 2: sendQuery(query: String):Model

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A string not parsed in a Query Language	A InvalidArgumentException is thrown.
Valid query string	The Data Access Manager performs the query on the Database and returns the proper instance of the Model class (or its subclass) based on what the query asks.

Table 3: sendUpdate(query: String):boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A string not parsed in a Query Language	A InvalidArgumentException is thrown.
A string not parsed in a updating instruction	A InvalidArgumentValueException is thrown
Valid query string	The Data Access Manager performs the updating on the Database and returns true if it succeed, otherwise false.

3.2.2 Reservation Controller, Communication Manager

This test is made to verify that the Communication Manager can perform the Method sent by Reservation Controller in a proper way in order to perform the desired effect on the system.

Table 4: executeRemoteMethod(method: Method):boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A method that is not listed in the class	Return false
Valid method	The Communication Manager dispatches the method come from the reservation and performs the proper action, described by the parameter passed.

3.2.3 Ride Controller, Communication Manager

This test is made to verify that the Communication Manager can perform the Method sent by Ride Controller in a proper way in order to perform the desired effect on the system.

Table 5: executeRemoteMethod(method: Method):boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A method that is not listed in the class	Return false
Valid method	The Communication Manager dispatches the method come from the RideController and performs the proper action, described by the parameter passed.

3.2.4 Request Manager, Reservation Controller

This test is made to verify that the reservation controller can validate the input received by the request manager and it is able to create a new valid instance of reservation using the user informations passed by the previous component.

Table 6: newReservation(user: User, carPlate: String):boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing user	A InvalidArgumentException is thrown
A user suspended from the service	A UserSuspendedException is thrown.
A not existing car	A InvalidArgumentException is thrown
Valid informations	Reservation Controller, first retrieves the cars data according to the carPlate passed, then creates a new instance of reservation and fills it with the information received as parameters. A new instance is saved in a permanent way to the database.

This test is made to verify that the Reservation Controller can retrieve the last reservation performed by a specific user and show it to the client.

Table 7: showLastReservation(user: User): void

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing user	A InvalidArgumentException is thrown
A user that hasnt performed a reservation yet	A InvalidArgumentValueException is thrown.
Valid informations	Reservation Controller creates a properly method instance to do the query on the database for retrieving the users last reservation. At the end, it shows the informations.

This test is made to verify that the reservation controller can extend correctly its duration.

Table 8: extendReservation(reservationId: Int, user: User, carPlate: String): boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid informations	Reservation Controller, first retrieves the reservation instance related to the inputs parameter, then modifies the information concerning the ending time extending its duration and it updates the record on the database, through a proper Method instance.

This test is made to verify that the reservation controller can delete the reservation passed. This test is made to verify that the reservation controller can extend correctly its duration.

Table 9: deleteReservation(reservationId: Int, user: User, carPlate: String): boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid informations	Reservation Controller deletes the record that corresponds to the inputs parameter on the database , through a proper Method instance.

This test is made to verify that the reservation controller can perform the code generator algorithm to provide the QR Code for unlocking the car reserved.

Table 10: showUnlockCode(reservationId: Int, user: User, carPlate: String): Code

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid informations	Reservation Controller, first retrieves the reservation instance related to the inputs parameter, then performs the unlocking code by the algorithm devoted to it and explained in Design Document (ER Diagram section) and returns it.

This test is made to verify that the reservation controller can perform the distance algorithm in order to providing a suitable list of reservable cars.

Table 11: retrieveCarsPosition(position: Position, range: Int) : List¡Car¿

Input	Effect on the system
A null parameter	A NullPointerException is thrown
An invalid position	A InvalidArgumentException is thrown
A nonpositive range	A InvalidArgumentValueException is thrown
Valid informations	Reservation Controller retrieves the list of available cars from the database and performs a specific algorithm to filter them according to their last position. At the end, it returns the filtered list

3.2.5 Notification Helper, Push Gateway

This test is made to verify that the Push Gateway notifies correctly the clients device using push notification.

Table 12: sendMessage(message: Message) : boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
An empty message	A InvalidArgumentValueException is thrown
Valid parameter	Push Gateway notifies the Client device through push notification reporting the string message contained in the Message passed.

3.2.6 Notification Helper, Email Gateway

This test is made to verify that the Email Gateway notifies correctly the clients device using email notification.

Table 13: sendMessage(message: Message) : boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
An empty message	A InvalidArgumentValueException is thrown
Valid parameter	Email Gateway notifies the Client application through email notification reporting the string message contained in the Message passed.

3.2.7 Reservation Controller, Notification Helper

This test is made to verify that the Notification Helper is able to handles correctly the information passed by Reservation Controller and can create and send a proper message to the right gateway.

Table 14: sendNotification(notification: Notification): void

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid parameter	Notification Helper handles the notification containing the information to be sent to the client, creates a proper message and send it through the right gateway.

3.2.8 Ride Controller, Notification Helper

This test is made to verify that the Notification Helper is able to handles correctly the information passed by Ride Controller and can create and send a proper message to the right gateway.

Table 15: sendNotification(notification: Notification): void

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid parameter	Notification Helper handles the notification containing the information to be sent to the client, creates a proper message and send it through the right gateway.

3.3 Account Subsystem

3.3.1 Communication Controller, Data Access Manager

See section 3.2.1.

3.3.2 Login/Registration Controller, Communication Manager

This test is made to verify that the Communication Manager can perform the Method sent by Login/Registration Controller in a proper way in order to perform the desired effect on the system.

Table 16: executeRemoteMethod(method: Method)

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A method that is not listed in the class	Return false
Valid method	The Communication Manager dispatches the method come from the Login/Registration Controller and performs the proper action, described by the parameter passed.

3.3.3 Notification Helper, Push Gateway

See section 3.2.5

3.3.4 Notification Helper, Email Gateway

See section 3.2.6

3.3.5 Login/Registration Controller, Notification Helper

This test is made to verify that the Notification Helper is able to handles correctly the information passed by Login/Registration Controller and can create and send a proper message to the right gateway.

Table 17: sendNotification(notification: Notification): void

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid parameter	Notification Helper handles the notification containing the information to be sent to the client, creates a proper message and send it through the right gateway.

3.3.6 Request Manager, Login/Registration Controller

This test is made to verify that the Login/Registration Controller is able to create a new stable session with the users data retrieved by the database in order to allow him for performing actions.

Table 18: login(password:String): User

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing password in the database	Return false
Valid password	The Login/Registration Controller retrieves all data referred by the password inserted in order to execute efficiently the next operations, so it creates a new session with the data for keeping track on the system about the user connection. The User instance is returned to the caller in order to give the possibility to save it into a suitable collection and to associate it to the respective session ID.

This test is made to verify that the Login/Registration Controller is able to delete the Session of the logging out user.

Table 19: logout(sessionId: int): boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid parameter	The Login/Registration Controller deletes the session related to the user that has requested to be logged out. It returns false if problems occur.

This test is made to verify that the Login/Registration Controller is able to create a new User.

Table 20: registration(name: String, surname: String, drivingLicense: String, email:String,paymentMethod:String): String

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A driving license that already exists	Return false
Valid credentials	The Login/Registration Controller creates a new instance of User and inserts it into the database, then it sends the password to the user through an email (using the Notification Helper).

3.4 Payment subsystem

3.4.1 Communication Controller, Data Access Manager

See section 3.2.1

3.4.2 Payment Controller, Communication Manager

This test is made to verify that the Communication Manager can perform the Method sent by Payment Controller in a proper way in order to perform the desired effect on the system.

Table 21: executeRemoteMethod(method: Method)

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A method that is not listed in the class	Return false
Valid method	The Communication Manager dispatches the method come from the Payment Controller and performs the proper action, described by the parameter passed.

3.4.3 Request Manager, Payment Controller

This test is made to verify that the Payment Controller is able to retrieves correctly the user and his list of pending payments.

Table 22: showPendingPayments(userPaymentID: int) : List¡Payment¿

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A wrong userPaymentID	A InvalidArgumentValueException is thrown
Valid ID	The Payment Controller performs the specific algorithm for converting the ID passed by the Payment societys system in order to retrieve the related User and then it checks if he has pending payments on the database. If he hasnt, it returns null, otherwise it returns the pending payments as a list.

This test is made to verify that the Payment Controller is able to update a payment record on the database.

Table 23: updatePayment(payment: Payment): boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing payment in the database	A InvalidArgumentValueException is thrown
Valid payment	The Payment Controller updates the related payment record in the database.

3.4.4 Payment Controller, Payment Gateway

This test is made to verify that the Payment Controller is able to send correctly the message to the payment company.

Table 24: sendMessage(message: Message) : boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid payment	The Payment Gateway sends the message, properly prepared by the Payment Controller, to the payment company. It returns false if an issue occurs, otherwise true.

3.5 Car subsystem

3.5.1 Communication Controller, Data Access Manager

See section 3.2.1

3.5.2 Cars Controller, Communication Manager

This test is made to verify that the Communication Manager can perform the Method sent by Cars Controller in a proper way in order to perform the desired effect on the system.

Table 25: executeRemoteMethod(method: Method)

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A method that is not listed in the class	Return false
Valid method	The Communication Manager dispatches the method come from the Cars Controller and performs the proper action, described by the parameter passed.

3.5.3 Collector Data, Car Gateway

This test is made to verify that the Car Gateway can catch the message arrived by external systems and can decide to which component delivers it.

Table 26: receiveAndDispatchMessage():Message

Input	Effect on the system
Nothing	The Car Gateway taps the message from the external system and it dispatches it to the proper component (in this case, the Collector Data) by a return.

This test is made to verify that the Car Gateway can send correctly the message passed by the Collector Data.

Table 27: sendMessage(message:Message):boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid message	The Car Gateway sends the message to the right external addressee. It returns false if an issue occurs, otherwise true.

3.5.4 Cars Controller, Collector Data

This test is made to verify that the Collector Data is able to elaborate the message arrived by external systems in order to extracts useful data.

Table 28: retrieveData(message: Message): Collection $\langle Data \rangle$

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid message	The Collector Data elaborates the message and extracts the data contained into it. It puts what extracted into a Collection and it sends it to the Cars Controller.

This test is made to verify that the Collector Data is able to create a proper message for the cars external systems using a list of data passed.

Table 29: createMessage(car: Car, data: List $\langle Data \rangle$) : void

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing car	An InvalidArgumentException is thrown
Valid parameters	The Collector Data creates a suitable message using the list of data passed by Cars Controller, then the message is sent to the cars external system indicated by the Car instance.

This test is made to verify that the Cars Controller is able to check if the code caught by the cars sensors is equal to that has been associated in the registration phase for that specific car and id it can unlock correctly the car.

Table 30: checkUnlockCode(code: Code): boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid parameters	The Cars Controller checks if the QR Code read by the cars system corresponds to the code associated to that specific car in a valid registration phase, by reading on the database. If the correspondence occurs, it return true and the car will be unlocked, otherwise false.

3.5.5 Cars Controller, Maintenance Gateway

This test is made to verify that the Maintenance Gateway can send correctly the message passed by the Cars Controller.

Table 31: sendMessage(message:Message):boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
Valid message	The Maintenance Gateway sends the message to the right external addressee. It returns false if an issue occurs, otherwise true.

3.5.6 Request Manager, Cars Controller

This test is made to verify that the Cars Controller can retrieve correctly the list of maintenance records store on the database.

Table 32: getReportedCars(): List$\langle MaintenanceRecord \rangle$

Input	Effect on the system
Nothing	The Cars Controller retrieves all the maintenance record related to cars which are waiting assistance. It puts them into a list and it returns that.

This test is made to verify that the Cars Controller can updates a maintenance record stored on the database with data passed by the Maintenance societys external system.

Table 33: updateMaintenanceRecord(mr: MaintenanceRecord):boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing record in the database	An InvalidArgumentException is thrown
Valid record	The Cars Controller updates the maintenance record on the database. If an issue occurs, it returns false, otherwise true.

This test is made to verify that the Cars Controller can open and save correctly a new report performed by the maintenance employee.

Table 34: openReportIssue(carPlate: String , description: String): boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A car's plate referred to a not existing car	An InvalidArgumentValueException is thrown
Valid parameter	The Cars Controller, first retrieves the cars information from the database, then creates a new instance of MaintenanceRecord based on the parameters and inserts it into the database. If an issue occurs, it returns false, otherwise true.

3.6 Subsystem interconnection and communication

This last section is devoted to show the testing of the interconnection and the communications among the subsystems listed above in order to pursuit the systems scope and to perform its functionalities.

3.6.1 Ride Controller, Communication Manager, Cars Controller (Cars subsystem, Reservation subsystem)

This test is made to verify that the Reservation Controller can terminate correctly a reservation when a ride related to it is started.

Table 35: completeReservation(reservation: Reservation): boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing reservation	An InvalidArgumentValueException is thrown
Valid parameter	The Reservation Controller updates on the database the record referred to the reservation passed as parameter in order to mark it completion, so updating the endingTime attribute. If an issue occurs return false, otherwise true.

3.6.2 Ride Controller, Communication Manager, Cars Controller (Cars Subsystem, Reservation Subsystem)

This test is made to verify that the Cars Controller can check if the car referred to the Car instance passed by the Ride Controller is plugged or not.

Table 36: isPlugged(car: Car): boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing car	An InvalidArgumentValueException is thrown
Valid parameter	The Cars Controllers subsystem checks if the car reported by the Car instance is plugged through its sensors. It returns true if its plugged, otherwise false.

This test is made to verify that the Cars Controller can check the battery charge level of a given car.

Table 37: getCharge(car: Car): Int

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing car	An InvalidArgumentValueException is thrown
Valid parameter	The Cars Controllers subsystem queries the cars sensors indicated by the Car instance to get the level of battery charge. It returns the value.

This test is made to verify that the Cars Controller can correctly retrieve the position of a given car.

Table 38: getPosition(car: Car): Area

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing car	An InvalidArgumentValueException is thrown
Valid parameter	The Cars Controllers subsystem checks if the cars engine is off, then it queries the database retrieving the informations about where the car is left when the ride before is finished, otherwise it retrieves the cars position using the GPS system located on the vehicle. It returns the Areas information based on the fact explained above.

This test is made to verify that the Ride Controller can create a new Ride when a Cars Controller observes that the engine of a reserved car is turned on.

Table 39: newRide(reservation: Reservation): boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing reservation	An InvalidArgumentValueException is thrown
Valid parameter	The Ride Controller creates a new instance of Ride and fills it with the informations concerning the associated reservation, then it inserts it into the database. If an issue occurs, it returns false, otherwise true.

This test is made to verify that the Ride Controller can correctly mark as finished a specific ride.

Table 40: completeRide(ride: Ride): boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A not existing ride	An InvalidArgumentValueException is thrown
Valid parameter	The Ride Controller updates specific attributes (for example, rides ending Time, in which safe area it is parked and so on) of the record situated on the database and related to the parameter passed in order to mark the ending of that existing ride. Since that moment, that specific car is unreserved and it is ready to be reserved again. If an error occurs, it returns false, otherwise true.

3.6.3 Ride Controller, Communication Manager, Payment Controller (Payment Subsystem, Reservation Subsystem)

This test is made to verify that the Payment Controller is able to notify correctly the Payment Society about a new pending payment concerning a finished ride.

Table 41: sendPayment(payment: Payment):boolean

Input	Effect on the system
A null parameter	A NullPointerException is thrown
A payment referred to a not existing ride or not existing user	An InvalidArgumentValueException is thrown
Valid parameter	The Payment Controllers subsystem sends the payment request to the Payment societys external system in order to notify them about a new payment to enforce. If an error occurs, it returns false, otherwise true.

4 Tools and test equipment required

4.1 Test tools

Testing for the PowerEnJoy system-to-be will be splitted in three main parts:

- Business logic components: in order to tackle testing of this section of the system, the two main tools that will be used are Arquillian integration testing framework and the JUnit framework. The first one is specifically designed for checking the execution of a component with respect to the surrounding environment. In this specific system, its functionalities will be used to verify that communication with database and how the communication flow is performed by the system with respect to what was expected. JUnit framework will be used for both the purpose of unit testing in combination with test coverage assist like Sonarqube, and its potentiality will be adapted to integration testing with respect to the correctness of responses that the various component give back when an error and exceptions occur;
- Mobile applications: the mobile application must be tested with respect of CPU and memory usage and energy consumption. This could be done by specific tools within the SDK of the platform target of testing;
- Validation, certification and standardisation for web pages: W3C is the specific tool for validating the system-to-be web code related to the user experience. Our aim with this kind of testing is to generate a product that will target the most wide pool of potential users and therefore clients.

4.2 Test equipment

In order to perform the entire test process proposed in the previous sections, we have chosen to stipulate a contract with a Cloud provider for hiring computational environment. Then we could use a cloud service like Platform as a Service (PaaS) in order to perform our test plan, including the integration test. This choice has been done because the cloud computing is a very cheap way to get computational power and also because our future intention (explained on the Design Document - Overview), after the first official release, is to move the system onto a cloud infrastructure, therefore the PaaS allows us to feel and to measure what will be the performances of our system running on this type of infrastructure, before the deployment; in addition, performing the test on an architecture of this kind allows us to make scalable tests, in order to undergo the system with incremental workloads. Once obtained the computational infrastructure, we will use those devices which will be implied by users to interact with our system: were referring to mobile phone devices (with various O.S. installed like iOS, Android, Windows Phone and so on) and Personal Computers in order to test the interaction between different types of clients and our system and also for testing how the systems responses are elaborated by those clients and how they are visualized. This kind of tests can be completed by using at full extent the tools that some development kits offer: the idea is to simulate different scenarios via this tools in order to check whether the performances, both visually and system-related, are matched and the correctness of the service erogated.

5 Required program driver and test data

5.1 Program driver and stubs

As stated before in this document, we are going to follow a bottom-up approach for integration testing. This decision leads to the necessity of creating the needed set of drivers in order to perform the identified tests pools, in particular we will construct drivers for each module, excluding those coming from third parties, as the DBMS and the API provided by the Google services. Hereafter is a list of drivers necessary to test the integration of the system-to-be:

- **Data Access Driver**: the development of this test module will be focused on invoking methods exposed by the *Data Access Manager* in order to test its interaction with the DBMS. We emphasize that the DBMS does not need a direct driver to be tested with, being a product supplied by third parties.
- **Communication Manager Driver**: the development of this test module will be focused on invoking methods exposed by the *Communication manager* in order to test the internal communication capabilities. Especially by testing the ability to invoke methods on the *Data Access Manager* and then allow access to the database to system components.
- **Ride Controller Driver**: the development of this test module will be focused on invoking methods exposed by the *Ride Controller* and therefore the invocation of all the methods related to the ride operations, and will focus on verifying the interaction with components such as *Communication Manager*, *Notification Helper* and *Google Maps API*.
- **Reservation Controller Driver**: as for the *Ride Controller Driver*, this module will be focused on invoking methods exposed by the *Reservation Controller* and therefore the invocation of all the methods related to the reservation operations. In addition, it will be used to test the correct interaction with components such as *Notification Helper* and *Communication Manager*.
- **Notification Helper Driver**: the development of this test module will be focused on invoking methods exposed by the *Notification Helper* in order to test the proper communication between this component and the gateway forwarding notifications to users, *Push Gateway* and *Email Gateway*.
- **Request Manager Driver**: the development of this test module will be focused on invoking methods exposed by the *Request Manager* and then verify the correct forwarding of requests coming from the outside to the right components. This module, in detail, check the communication with different components: *Login / Registration Controller*, *Ride Controller*, *Reservation Controller*, *Payment Controller* and finally *Cars Controller*.
- **Login/Registration Controller Driver**: the development of this test module will be focused on invoking methods exposed by the *Login/Registration Controller* and then verify the correct operation about login and

registration.This module will furthermore verify the interaction with *Notification Helper* for submitting notifications and *Communication Manager* for interacting with other components and database.

- **Payment Controller Driver**: the development of this test module will be focused on invoking methods exposed by the *Payment Controller* in order to check the proper functioning the operations relating to the management of payments. This module will verify the correct connection with the *Communication Manager* and the *Payment Gateway* to test the correct routing of payment notifications to external company.

- **Payment Gateway Driver**: the development of this test module will be focused on invoking methods exposed by the *Payment Gateway* to test che correct communication with the external company.

- **Cars Controller Driver**: the development of this test module will be focused on invoking methods exposed by the *Cars Controller* in order to check the correct functioning of the operations relating to the management of sensors and actuators on the machines, as well as functions relating to maintenance. This module will furthermore test the interaction with differents components such as *Communication Manager*, *Maintenance Gateway* to test the correct forwarding of notifications to the maintenance company, and *Collector Data* to ensure the correct communication with cars.

- **Maintenance Gateway Driver**: the development of this test module will be focused on invoking methods exposed by the *Maintenance Gateway* to verify the correct communication with the maintenance company.

- **Collector Data Driver**: the development of this test module will be focused on invoking methods exposed by the *Collector Data* to test the proper communication with sensors and actuators on cars.

5.2 Test data

As expressed in section 3 (Individual Steps and Test Description) in order to perform the set of tests listed, you must define a data set to cover all variants. In particular we will need:

- A set of possible *Request* to test the Request Manager component, providing different features:

  - a null object;

  - more than one correct instance of Request in order to cover all the possible request from the clients.

- A set of possible query to test the Data Access Manager component, providing different features:

  - a null parameter;

  - a string not correctly parsed in a Query Language;

- more than one correct query to cover different type of access to the database.

- A set of possible *Method* to test the Communication Manager component, providing different features:

  - a null object;

  - a Method undefined;

  - more than one correct instance of Method to cover different types of communication.

- a set of possible *User* to test the Reservation Controller component, providing different features:

  - a null object;

  - a not registered user;

  - more than one valid user with different features like having or not a reservation, being in a ride, being active or suspended from the service.

- a set of possible password to test the Login/Registration Controller component, providing different features:

  - a null parameter;

  - a not defined password in the database;

  - a valid password.

- a set of possible driving license to test the Login/Registration Controller component, providing different features:

  - a driving license already registered in the database;

  - a driving license new to the database.

- a set of possible car plate to test the Reservation Controller and Car Controller components, providing different features:

  - a null parameter;

  - a car plate not listed in the database;

  - a correct car plate.

- a set of possible ReservationID to test the Reservation Controller component, providing different features:

  - a null parameter;

  - a not existing id;

  - a valid id.

- a set of possible Reservation to test the Ride Controller component, providing different features:

  - a null parameter;

  - a not existing Reservation in the database;

  - more than one valid Reservation to test the correct management of them.

- a set of possible Ride to test the Ride Controller component, providing different features:

  - a null parameter;

  - a not registered Ride in the database;

  - a set of valid Ride covering all possible cases such as as being completed, in progress or being just started.

- a set of possible position to test the Reservation Controller component, providing different features:

  - a null parameter;

  - an invalid position;

  - a set of valid positions related to safe and unsafe area.

- a set of possible Message to test the Push, Email, Car, Payment and Maintenance gateway components, providing different features:

  - a null parameter;

  - an empty Message;

  - a set of proper defined Messages to be sent via email;

  - a set of proper defined Messages to be sent via push notification;

  - a set of proper defined Messages to the cars of the company;

  - a set of proper defined Messages to be sent to the maintenance system;

  - a set of proper defined Messages to be sent to the payment system.

- a set of possible Notification to test the Notification Helper component, providing different features:

  - a null parameter;

  - a set of valid Notifications.

- a set of possible Payment to test the component, providing different features:

  - a null parameter;

  - a not defined payment;

  - more than one valid payment to cover different possibilities such a payment completed, a pendant one and a payment not well-defined (for example not linked with an existing Ride).

- a set of possible Car to test the Collector Data and Cars Controller components, providing different features:

  - a null parameter;

  - a not registered car;

  - a valid car.

- a set of possible unlock Code to test the Car Controller component, providing different features:

  - a null parameter;

  - a set of valid code in order to verify the proper checking of the code linked to a car reservation.

- a set of possible Maintenance Record to test the Cars Controller component, providing different features:

  - a null parameter;

  - a not registered Maintenance Record;

  - a set of valid records in order to test operations such as the update and the closure by the maintenance.

Used tools

- Github: for version control
- GoogleDoc: to write the document
- Draw.io: to create the diagrams
- LaTeX: to create the pdf

Work hours

- Emanuele Ricciardelli: $\sim$ 30 hrs.
- Giorgio Tavecchia: $\sim$ 30 hrs.
- Francesco Vetró: $\sim$ 30 hrs.