Testing Document

POLITECNICO
MILANO 1863

Software Engineering 2
PowerEnJoy

Authors:

Bianchi Alessandro (Mat. 875035)
Canciani Francesco (Mat. 807056)
Cannone Lorenzo (Mat. 875802)

Document version: 1.0

Academic year:
2016-2017
Release date:
15-01-2017

Contents

1.Introduction

1.1 Revision History

Version	Date	Authors	Summary
1.0	15-01-2017	Bianchi A., Canciani F., Cannone L.	First release

1.2 Purpose and scope

The Integration Test Plan Document (ITPD) aims at outlining how to accomplish the integration test, by means of delineating, in a clear and comprehensive way, the main aspects concerning the organization of the integration testing activity for all the components that make up the system.
The starting point for the definition of the document is the PowerEnJoy's software architecture, that was previously illustrated in the Design Document. The document's purpose is to then explain to the development team how to approach the integration test, before this process really takes place. Along the entire document, PowerEnJoy's developers will find a specific list of what to test, in which sequence, which tools they will need for testing and which stubs/drivers/oracles need to be developed, so that, if the testing activity is performed correctly, we will guarantee that all different subsystems composing PowerEnJoy system, interoperate consistently with the requirements that they are supposed to fulfill.

1.3 List of Definitions and Abbreviations

- **ITPD**: Integration Test Plan Document
- **DD**: Design Document
- **RASD**: Requirements Analysis and Specification Document
- **MSO**: Money Saving Option
- **Java EE**: Java Enterprise Edition
- **DBMS**: Database Management System
- **DB**: Database
- **Subcomponent**: each of the low level components realizing the functionalities of a subsystem
- **Subsystem**: a high-level functional unit of the system
- **GPS**: Global Positioning System

1.4 List of Reference Documents

- Project description document: Assignments **AA 2016-2017.pdf**
- PowerEnJoy Requirements Analysis and Specification Document: **RASD.pdf**
- PowerEnJoy Design Document: **DD.pdf**
- Integration Test Plan Document examples:
  - Integration testing example **document.pdf**
  - Sample Integration Test Plan **Document.pdf**

2.Integration strategy

2.1 Entry Criteria

Before the integration testing of specific elements may begin, there are certain criteria that must be met in order to fulfill the appointed task:

1. The Requirements Analysis and Specification Document must be fully completed. There would be no need to test anything if we did not know specifically what are the objectives of PowerEnJoy's system. In other words, all the requirements of PowerEnjoy's application must be written and agreed upon by the stakeholders.

2. The Design Document must be fully completed. The reason for this is that, since the ITPD takes the architectural description of the software system as a starting point, we need to have a specific representation of our system-to-be's architecture clearly outlined and the Design Document is where we can find it.

3. All the single components perform well individually, according to their defined purpose. Since the purpose of integration testing is to combine software units together and then test them as group, we need to make sure that all the single components have been unit tested. Typically, by unit testing we mean the testing of single pieces of code, but, in this context, we make the assumption that everything that is inside a certain component is not visible at the integration level and therefore is tested at the unit level. This means that we assume that all of the components' functions have been properly unit tested and we can rely on them without having to worry about their single behavior when we integrate components with each other.

2.2 Elements to be Integrated

The elements that need to be integrated and tested upon are all the components that form the logic tier of the "Component view" diagram that was proposed in our previously released PowerEnJoy's Design Document (paragraph 2.3).

In addition to the components of the logic tier, the integration testing process will also included a "Data Access Utilities" component, which was not explicitly mentioned in the Design Document but it is definitely inferable from the Data Access layer that was represented in the "Overview" paragraph of the DD's architectural design (paragraph 2.1, dark violet rectangle). Due to the presence of constant queries to the database by several architecture's components, it is essential to have the data access component as part of the integration testing process.

Another list of elements that will need integration consists of the external components of the entire architecture:

- The "OnBoardAndroidDevice", which allows to control each car's actuators through the system's car controller;
- The "Operators" component, present in the pre-existing system, that manages the operators who provide assistance to PowerEnJoy's vehicles when notified through the administration controller;
- A number of commercial, already existing components used to achieve distinct functionalities:
  - DBMS
  - SMS Gateway
  - Email Gateway
  - Payment Processor
  - Google Maps

2.3 Integration Testing Strategy

The strategy that we decided to follow to perform the integration testing task is one of the typical strategies for incremental integration and testing, which is the bottom up approach. This approach is conducted by integrating together components that either do not rely on other components to work, or depend on components that have already been developed. The rationale for picking this approach is to find in the approach's advantages: by performing testing on fully developed components it is possible to work with data from the beginning and thus be able to observe test results in an easier way, allowing ourselves to track down major flows when they occur at the "bottom" of the program. Another major advantage of this approach, that comes from its nature of being an incremental integration testing technique, is that components are tested as soon as they are released. This results in an intensive enhancement of efficiency during the development process. Lastly, bottom up approach implies that there is no need for stubs. However, a few stubs will be introduced to simulate the existence of clients and cars. Chapter 5.1 will be more clear about this last topic.

It is important to notice that DBMS, Payment Processor, Google Maps, SMS Gateway and Email Gateway are all commercial components that have already been developed and can thus be immediately used in a bottom up approach without any explicit dependency.

It is not the same for the other remaining external components, which are the OnBoardAndroidDevice and the Operators components. Both of these components do not belong to the internal architecture of the system but testing their integration with the internal components of PowerEnJoy is crucial and that is why they will appear shortly in the following integration sequence.

2.4 Sequence of Component/Function Integration

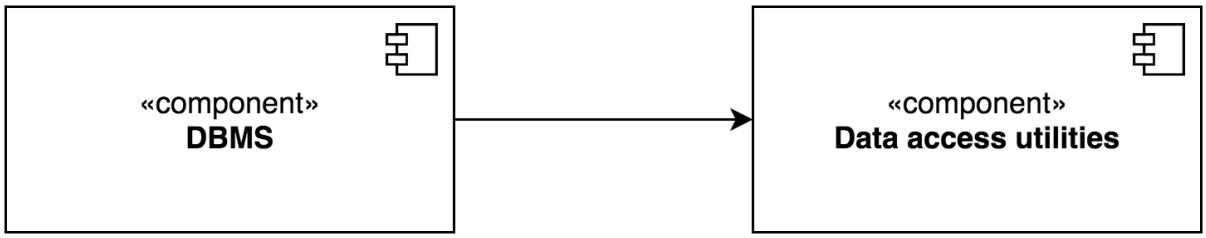
In this section we are going to describe the order of integration (and integration testing) of the various components and subsystems of PowerEnJoy. For what concerns the notation, an arrow going from component C1 to component C2 means that C1 is necessary for C2 to function and so it must have already been implemented.

2.4.1 Software Integration Sequence

Following the already mentioned bottom-up approach, we now describe how the various subcomponents are integrated together.

Data access utilities

The first two elements that need to be integrated are the Data Access Utilities and the DBMS components. We start from here because several components rely on Data Access Utilities to perform queries on the underlying data structure.

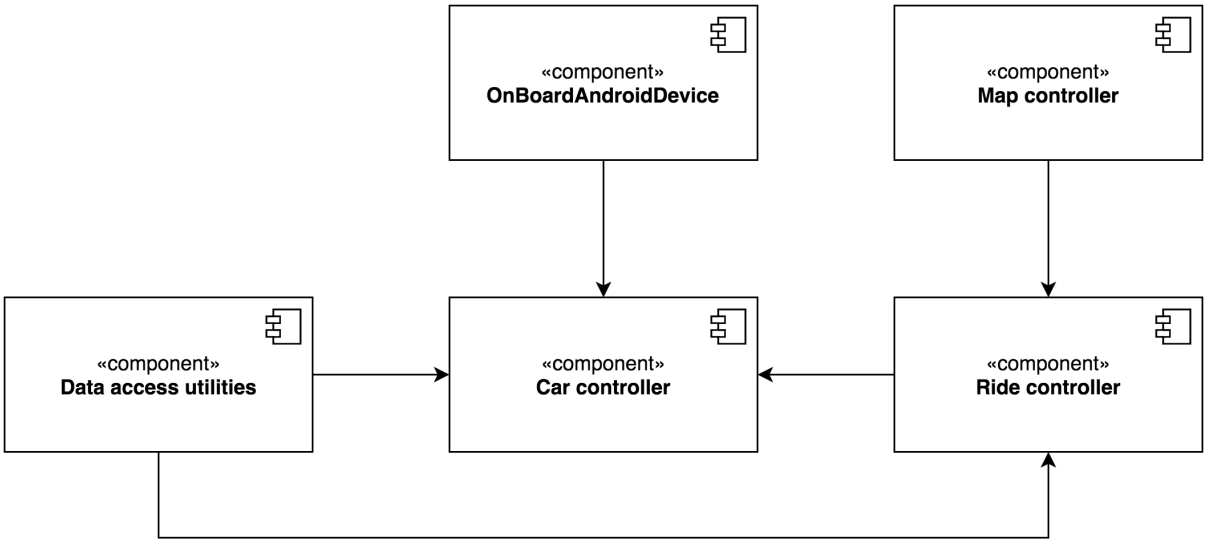


Reservation management system

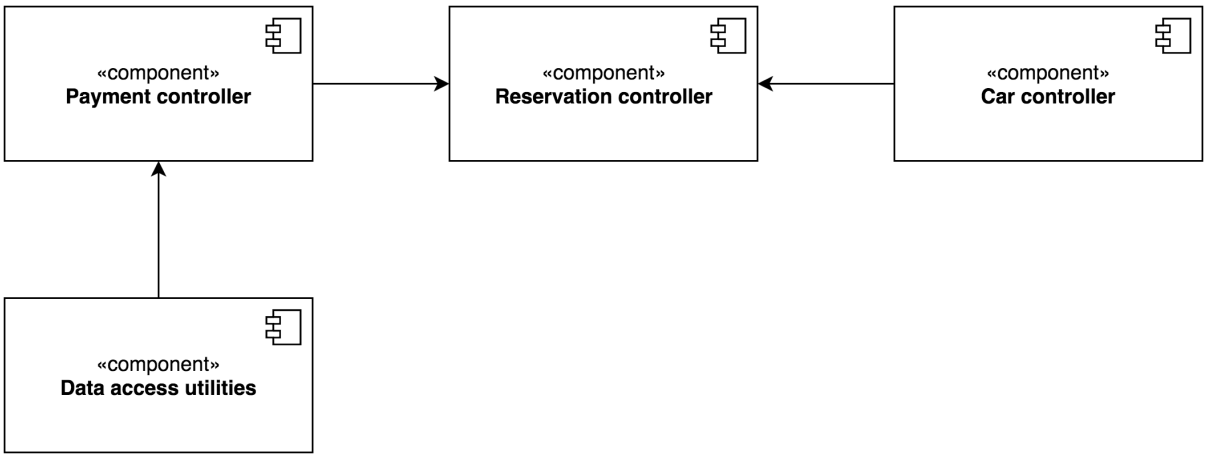
The second step in the integration process is to appropriately connect the subcomponents implementing the Reservation Management System. This choice comes from the idea of testing first what is supposed to be the core of the system. The most critical modules that compose the PowerEnJoy's system are all the ones that relate to the ability for a user to benefit of a car ride. This process starts with a reservation and ends when the ride is over and the user pays automatically for their ride. In the following diagrams, we are going to show exactly which components must be integrated together using the bottom up approach to allow the completeness of the previously mentioned operation.

First, we proceed by integrating together the Car controller subcomponent with the Data access utilities and Ride controller components.

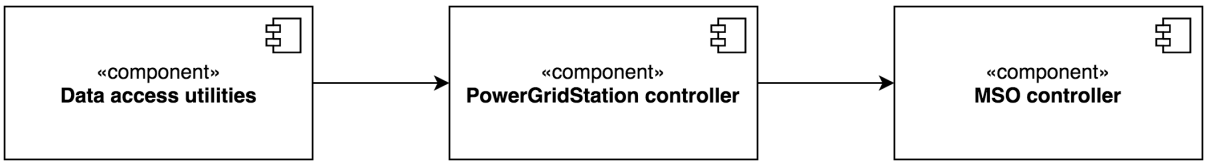
Next, we go on integrating by adding the two subcomponents OnBoardAndroidDevice and the Map controller.



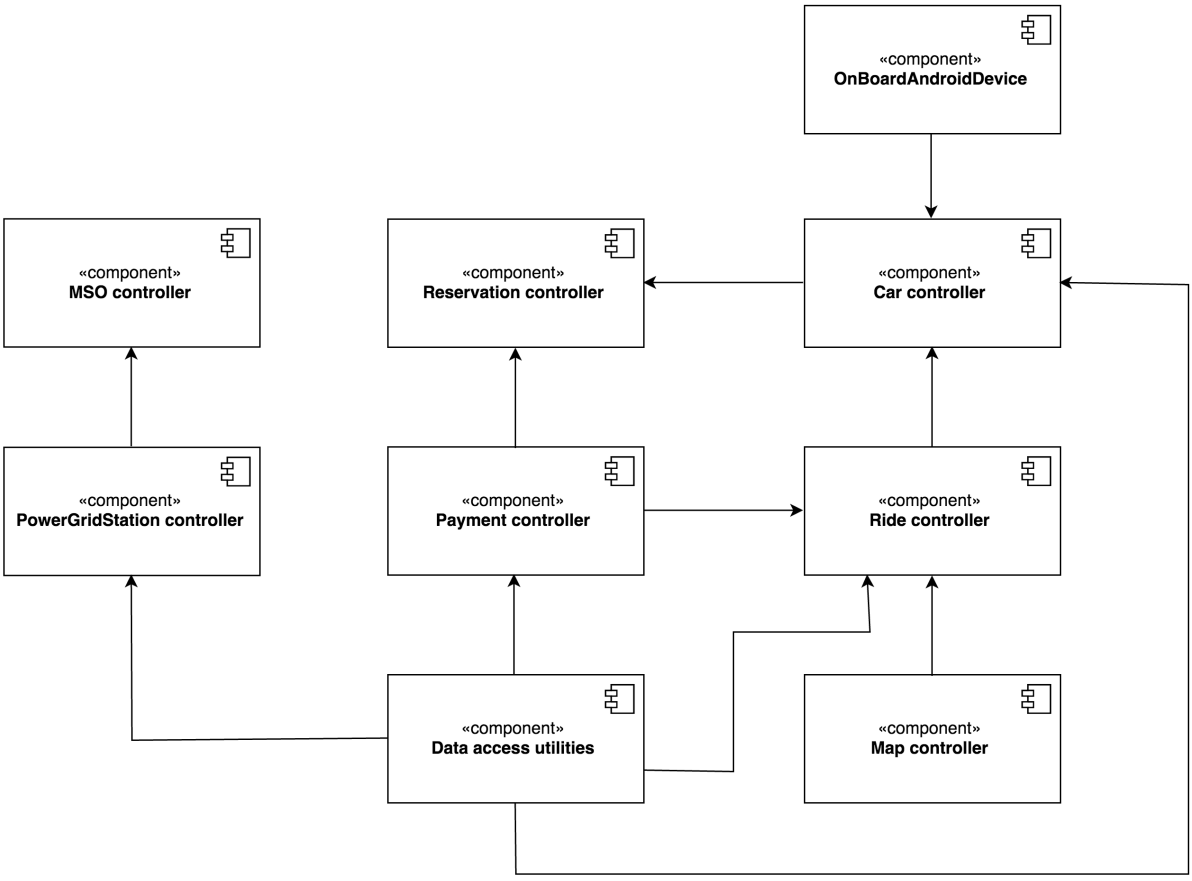
The same activity of testing integration is similarly performed on another integration testing block that still involves Data access utilities and Car controller component, but this time connected to Payment controller and Reservation controller.



Lastly, another block that involves Data access utilities is tested. This time, with the two components that compose the additional money saving option functionality, which are PowerGridStation controller and MSO controller.



Eventually, the components that build the Reservation Management System are ready to be integrated together.

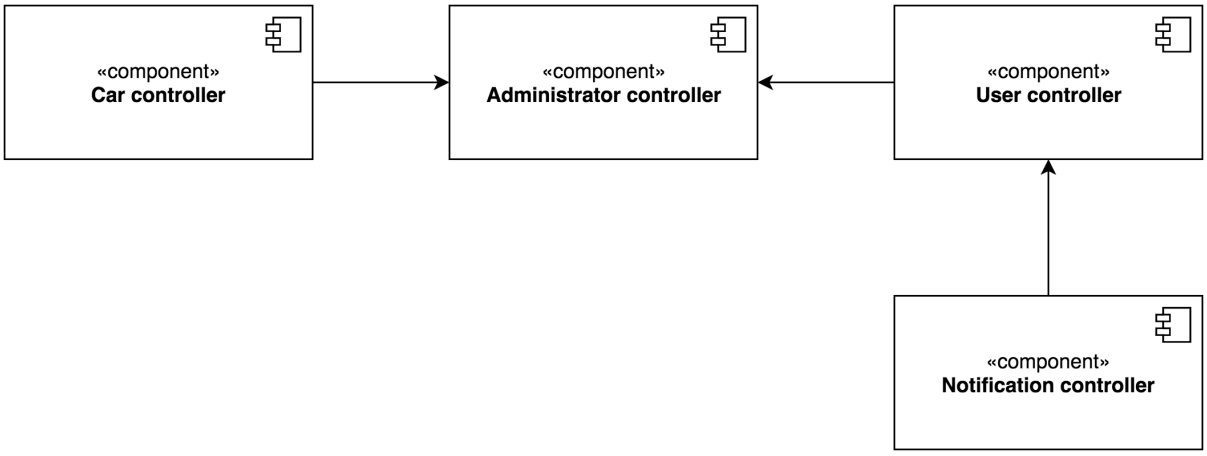


Administrator system

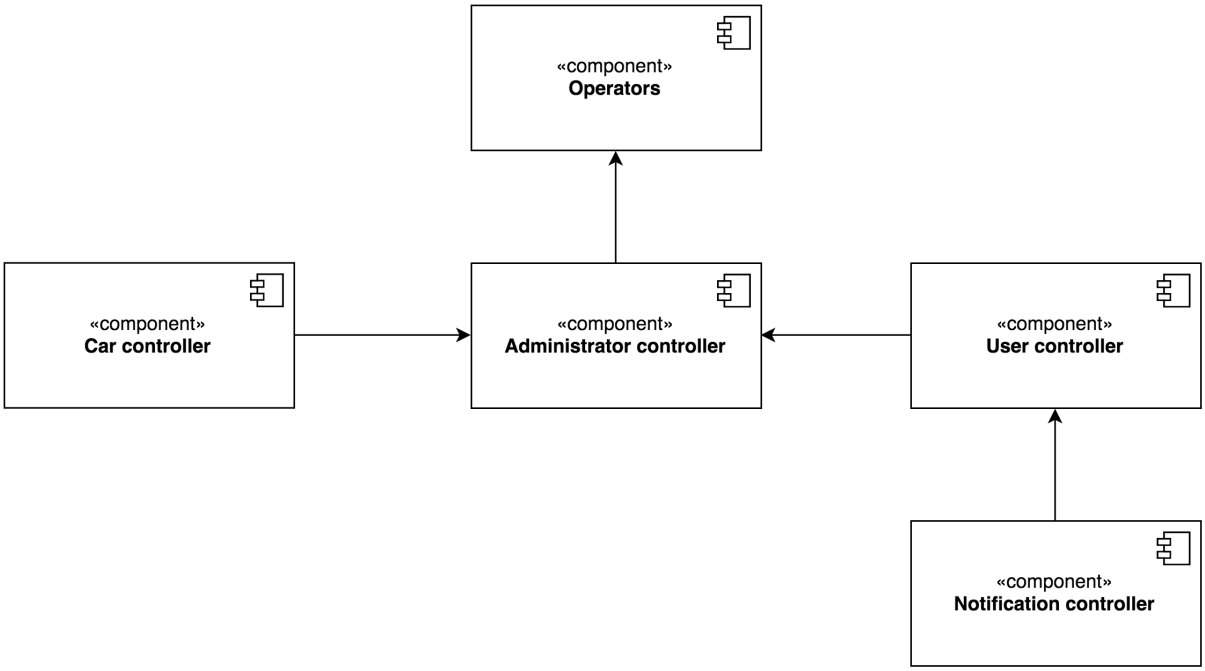
The third step in the integration testing process is to appropriately connect the subcomponents implementing the Administrator System subsystem. Part of this subsystem relies on functionalities offered by the old system, like the Operators component. We will first show, in the following diagrams, how subcomponents of this subsystem interact with each other using a bottom up approach, that will eventually lead to the final interaction with the previously mentioned external component.

First, we proceed by integrating together the Car controller subcomponent and the User controller subcomponent with the Administrator controller.

Then, we add to this initial block, the integration test that connects the User Controller with the Notification controller.

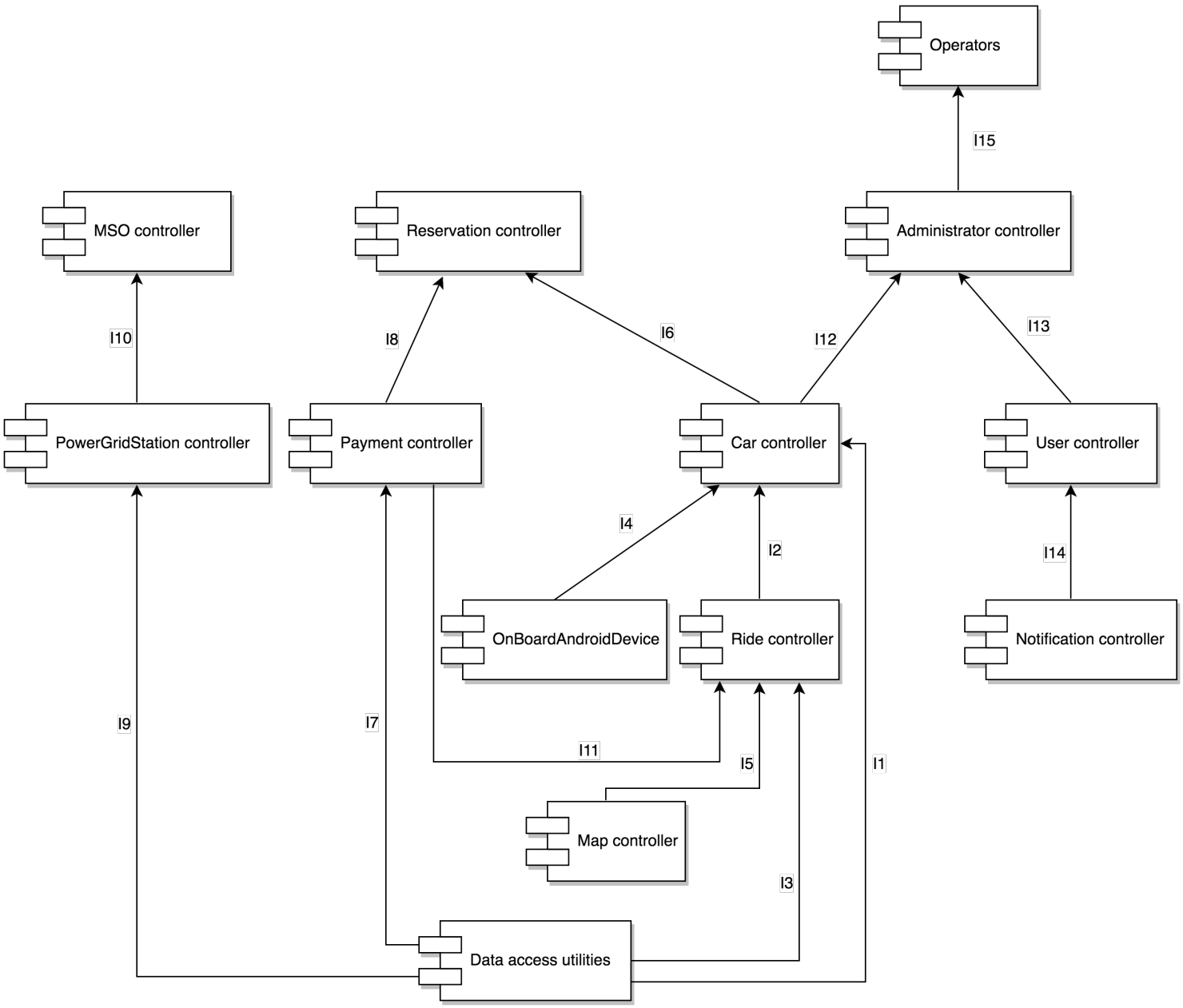


Finally, the four subcomponents of Administrator System are ready to be integrated with the component named Operators system, which, as previously mentioned, belongs to a system which is external to PowerEnJoy's system.



2.4.2 Subsystem Integration Sequence

In the following diagram we provide a general overview of how the various subsystems are integrated together to create the full PowerEnJoy infrastructure.



3.Individual Steps and Test Description

3.1 Reservation management system

I1- Car controller, Data access utilities

insertCar(car)	
Input	**Effect**
Null parameter.	NullArgumentException is thrown.
A Car already existent in the database.	InvalidArgumentException is thrown.
A valid parameter.	A new record is inserted in the database.
deleteCar(car)	
Input	**Effect**
Null parameter.	NullArgumentException is thrown.
A Car non existent in the database.	InvalidArgumentException is thrown.
A valid parameter.	The corresponding record is deleted from the database.
getCars()	
Input	**Effect**
	It returns a list containing all the Car objects in the database

I2 - Car controller, Ride controller

startRide(user, car)	
Input	**Effect**
Null parameter(s).	NullArgumentException is thrown.
Wrong car state (not in EngineOnCar).	InvalidCarStateException is thrown.
Blocked user.	InvalidUserStateException is thrown.
Valid parameters.	Ride starts correctly and startCounting() gets called by the Ride Controller.

I3 - Ride controller, Data access utilities

insertRide(ride)	
Input	**Effect**
Null parameter.	NullArgumentException is thrown.
A Ride already existent in the database.	InvalidArgumentException is thrown.
A valid parameter.	A new record is inserted in the database.
deleteRide(ride)	
Input	**Effect**
Null parameter.	NullArgumentException is thrown.
A Ride non existent in the database.	InvalidArgumentException is thrown.
A valid parameter.	The corresponding record is deleted from the database.
getRides()	
Input	**Effect**
	It returns a list containing all the Ride objects in the database

I4 - Car controller, OnBoardAndroidDevice

getPosition()	
Input	**Effect**
	The Android device returns the correct location by GPS. If GPS isn't able to provide informations, a GPSErrorException gets thrown.
drivingSeatIsUsed()	
Input	**Effect**
	It returns true if someone is seating on the driving seat according to the seat sensor placed underneath it. Otherwise it returns false. If the sensor isn't working, a SeatSensorException is thrown.
getPassengersNumber()	
Input	**Effect**
	It returns the number of the used seats, except for the driving one. If some sensor isn't working, a SeatSensorException is thrown.
engineIsOn()	
Input	**Effect**
	It returns true if the engine is on, false otherwise. If the Engine Sensor cannot send any data, an EngineSensorException is thrown.

doorsAreClosed()	
	It returns true if all the doors are closed, false otherwise. If some Door Sensor doesn't work, a DoorSensorException is thrown.
isCharged()	
Input	**Effect**
	It returns true if the battery is full. In that case the car state switches from Charging to Available. It returns false otherwise, and no states switching are made. If there's some issues with the Battery Sensor, a BatterySensorException is thrown.
lockDoors()	
Input	**Effect**
	It communicates with the Door Sensors and makes it lock all the doors. Returns true if the doors weren't already locked, false otherwise. If the sensors can't work, a DoorSensorException is thrown.

I5 - Ride controller, Map controller

distanceCar(car, PGS)	
Input	**Effect**
Null parameter(s).	NullArgumentException is thrown.
Invalid Position(s).	InvalidPositionException is thrown.
Valid parameters and return value >= 3KM	No further calls, no modifiers applied.
Valid parameters and return value < 3KM	3KM modifier gets applied by the Payment Controller.

I6 - Reservation controller, Car controller

getState()	
Input	**Effect**
	It returns an instance of the car object.
setState(car)	
Input	**Effect**
Null parameter.	NullArgumentException is thrown.
Car not in EngineIsOffCar or AvailableCar.	InvalidCarStateException is thrown.
Current state not in EngineIsOffCar or AvailableCar.	InvalidReservationException is thrown.
AvailableCar, same as the current car state.	AlreadyDroppedReservationException is thrown.
EngineIsOffCar, same as the current car state.	AlreadyReservedException is thrown.
AvailableCar or EngineIsOffCar, different from the (valid) current state.	It changes the car state into the parameter's one.
lock()	
Input	**Effect**
	it calls lockDoors() on the On Board Device. If the doors were unlocked before the call, it returns true, false otherwise. If the inner call throws an exception, it propagates it.

I7 - Payment controller, Data access utilities

insertTransaction(transaction)	
Input	**Effect**
Null parameter.	NullArgumentException is thrown.
A Transaction already existent in the database.	InvalidArgumentException is thrown.
A valid parameter.	A new record is inserted in the database.
deleteTransaction(transaction)	
Input	**Effect**
Null parameter.	NullArgumentException is thrown.
A Transaction non existent in the database.	InvalidArgumentException is thrown.
A valid parameter.	The corresponding record is deleted from the database.
getTransactions()	
Input	**Effect**
	It returns a list containing all the Transactions in the database

I8 - Reservation controller, Payment controller

commitPaymentToProcessor(amount)	
Input	**Effect**
Null float.	NullArgumentException is thrown.
Amount < 0	InvalidAmountException is thrown.
Amount >= 0 and user can afford the tax.	Payment Processor charges the user, true value returned.
Amount >= 0 and user cannot afford the tax.	Payment Processor marks the payment as unsolved and the user as i nsolvent, false value returned and the system administration gets notified.

I9 - Power grid station controller, Data access utilities

insertPGS(pgs)	
Input	**Effect**
Null parameter.	NullArgumentException is thrown.
A PGS already existent in the database.	InvalidArgumentException is thrown.
A valid parameter.	A new record is inserted in the database.
deletePGS(pgs)	
Input	**Effect**
Null parameter.	NullArgumentException is thrown.
A PGS non existent in the database.	InvalidArgumentException is thrown.
A valid parameter.	The corresponding record is deleted from the database.
getPGSs()	
Input	**Effect**
	It returns a list containing all the Power Grid Stations in the database

I10 - MSO controller, Power grid station controller

getLocation()	
Input	**Effect**
	It returns the correct position of the selected Power Grid Station.
getAvailableParkingSpot()	
Input	**Effect**
	It returns a ParkingSpot list in which there is at least one power plug that is not busy. If there's no available Parking Spot, the returned list is empty.

I11- Ride controller, Payment controller

commitPaymentToProcessor(amount)	
Input	**Effect**
Null float.	NullArgumentException is thrown.
Amount < 0	InvalidAmountException is thrown.
Amount >= 0 and user can afford the tax.	Payment Processor charges the user, true value returned.
Amount >= 0 and user cannot afford the tax.	Payment Processor marks the payment as unsolved and the user as i nsolvent, false value returned and the system administration gets notified.

3.2 Administrator System

I12 - Administrator controller, Car controller

getState()	
Input	**Effect**
	This method is called by getCarsThatNeedAssistance(), it is invoked for each car on the map and it simply returns the current state of them. If some car is marked as NeedAssistanceCar, the operators get notified about their the location.

I13 - Administrator controller, User controller

sendEmail(recipient, object, message)	
Input	**Effect**
Null parameter(s).	NullArgumentException is thrown.
Invalid recipient.	InvalidEmailRecipientException is thrown.
Empty object.	InvalidEmailObjectException is thrown.
Empty message.	InvalidEmailMessageException is thrown.
Valid parameters.	It can be called by blockUser() or unblockUser() or notifyOperators(), so the recipient will be the user's e-mail that he provided during the sign in or the operator's one. The object can be: • "Your account has been suspended" and the message will contain the explanation. That's the scenario for the blockUser() call. • "Your account is not suspended anymore" and the message will contain the explanation. That's the scenario for the unblockUser() call. • A custom object, depending on the needs for the e-mail. So it's for the message. That's the scenario for the notifyOperators() call.

sendSMS(phoneNumber, message)	
Input	**Effect**
Null parameter(s).	NullArgumentException is thrown.
Invalid phoneNumber.	InvalidPhoneNumberException is thrown.
Empty message.	InvalidEmailMessageException is thrown.
Valid parameters.	It has the same purpose of sendEmail(String, String, String), so the effects are the same, except for the object that in this case is missing.

I14 - User controller, Notification controller

sendEmail(recipient, object, message)	
Input	**Effect**
Null parameter(s).	NullArgumentException is thrown.
Invalid recipient.	InvalidEmailRecipientException is thrown.
Empty object.	InvalidEmailObjectException is thrown.
Empty message.	InvalidEmailMessageException is thrown.
Valid parameters.	An email is sent to the recipient with the inserted object and message. This function will be called by createPassword() in User Controller any time a new user registers into the system. The message will contain the generated password for that user.

I15 - Operators, Administrator controller

insertCarThatNeedsAssistance(car)	
Input	**Effect**
Null parameter.	NullArgumentException is thrown.
A NeedAssistanceCar already existing in the database.	InvalidArgumentException is thrown.
A valid parameter.	A new record is inserted in the database.
deleteCarThatNeedsAssistance(car)	
Input	**Effect**
Null parameter.	NullArgumentException is thrown.
A NeedAssistanceCar non existent in the database.	InvalidArgumentException is thrown.
A valid parameter.	The corresponding record is deleted from the database.
getCarsThatNeedAssistance()	
Input	**Effect**
	It returns a list containing all the Cars in need of assistance in the database

4.Tools and Test Equipment Required

4.1 Tools

In order to accomplish the integration testing process, PowerEnJoy's testers will take advantage of some tools that automate and facilitate the testing process.

4.1.1 Mockito

The first tool is an open source testing framework for Java, released under the MIT License, named Mockito. Mockito allows the creation of mock objects in automated unit tests. Mocking allows the developers to abstract dependencies and have predictable results, by also testing that the interaction between the testee and the mock is correct. Considering the fair number of external components that are present inside our PowerEnJoy architecture, we believe that mocking is one of the major actions to take in order to test these external services interaction against our system's components and Mockito allows our developers to do so in a simplified environment.

4.1.2 Arquillian

The second tool, which is a fundamental piece in the integration testing process, is Arquillian. Arquillian is an integration testing framework for Java EE. The reason we will use this tool is that it allows us to execute test cases against the container. Since testing a component in a business app is challenging and the interaction with the system is as important as the performed work, a tool like Arquillian is essential. In light of the fact that there is a constant interaction between the system and the database, it is crucial that the transactions are performed correctly and Arquillian allows us test that this process happens just like expected.

4.1.3 Apache JMeter

The final tool included in this list, relates to a different type of testing idea which goes under the title of performance testing. In order to simplify performance testing we decided to use another open source software named Apache JMeter. JMeter is a Java application designed to load test functional behavior and measure performance. It can thus be used to simulate a heavy load on a server and then analyze the overall performance under different load types. The spread in use of PowerEnJoy's application is definitely something that our stakeholder hope to achieve and being able to scale, which means handling a growing amount of work, is vital for our business. This is the main reason why we decided to introduce the need of an performance testing tool like Apache JMeter.

4.1.4 Manual testing

Lastly, it is important to notice that some of the features and operations included in PowerEnJoy's system will not be tested with automated testing tools because these operations may include a lot of manual activities. For instance, including manual testing scenarios is very important when it comes to mobile application testing strategy. When testing a mobile application, the testing team ought to test the various events which may occur when the application is being executed (incoming calls, SMSs, low battery, alerts such as emails and other apps notifications). In PowerEnJoy's mobile apps, users also take advantage of the possibility of been located through the GPS of their mobile phone. This can also be tested manually. Therefore, manual testing has still to be considered one of the "tools" required to perform the testing process.

4.2 Test equipment

The integration testing activities have to be performed in all the environments in which the application and the system will run. For this reason, beneath is a list of all the equipment needed to accomplish this task.

For what concerns the client side, testing has to be performed on the mobile application and the web application.
Regarding the mobile applications, the following devices are required:

- Android smartphone, at least running Android 4.4 KitKat
- iOS smartphone, at least running iOS 7

All the smartphones have of course to be equipped with at least 3G connection and GPS in order for the system to locate them along the map. It would also be wise to test mobile devices with different sizes and display resolution in order to assess the correct visualization of the application content, independently of the phone's format.

With respect to the desktop web application, there is no specific requirements on display resolution, operating system and processing power, besides the need of an updated web browser. This means that tests should be performed for each of the major browser available which we consider them to be: Chrome, Mozilla Firefox, Safari and Microsoft Edge.

Total different environments are needed for what concerns the backend testing, which is the server side of PowerEnJoy system. The business logic components should be deployed on a cloud infrastructure that closely mimics the one that will be used in the operating environment. There is no need at this point to actually test the application on a physical server so a subscription to a cloud computing platform and infrastructure with the same characteristics of our intended final physical server is suitable for our testing needs. As stated in PowerEnJoy's deployment view (paragraph 2.4, Design Document), our main server will run on WildFly, which is written in Java and implements the Java Platform, Enterprise Edition. About which cloud platform to choose, we leave the decision to the testers, in order to allow them to select the best offering subscription, at the time of the testing phase, that suits their needs, but always keeping in mind that it will have to reflect the same specifications of the intended final physical server.

Finally, to the extent of all the equipment that is needed to test the conceived performance of the system, it is obvious that also at least a pair of actual cars to test the correct appliance of remote controls, from the system to the car, are included in the necessary testing equipment.

5.Program Stubs and Test Data Required

5.1 Program Stubs and Drivers

As it was previously mentioned in section 2.3, we decided to adopt the bottom up approach for the integration testing process.
Because of this, several program drivers are needed in order to perform the necessary method invocations on the components that have to be tested.

Following, the list of drivers that are needed to perform the task that was just mentioned, together with their specific purpose:

- **Data Access Driver**: this testing module will invoke the methods exposed by the Data Access Utilities component in order to test its interaction with the DBMS.
- **Ride Controller Driver**: this testing module will invoke the methods exposed by the Ride Controller subcomponent in order to test its interaction with the Data Access Utilities, Payment Controller and Map Controller components.
- **Car Controller Driver**: this testing module will invoke the methods exposed by the Car Controller subcomponent in order to test its interaction with the Data Access Utilities, the Ride Controller components and the OnBoardAndroidDevice external component.
- **PowerGridStation Controller Driver**: this testing module will invoke the methods exposed by the PowerGridStation Controller subcomponent in order to test its interaction with the Data Access Utilities component.
- **MSO Controller Driver**: this testing module will invoke the methods exposed by the MSO Controller subcomponent in order to test its interaction with the PowerGridStation Controller component.
- **Reservation Controller**: this testing module will invoke the methods exposed by the Reservation Controller subcomponent in order to test its interaction with the Payment Controller and the Car Controller components.
- **User Controller Driver**: this testing module will invoke the methods exposed by the User Controller subcomponent in order to test its interaction with the Notification Controller component.

- Administrator Controller Driver: this testing module will invoke the methods exposed by the Administrator Controller subcomponent in order to test its interaction with the Car Controller and the User Controller components.
- Payment Controller Driver: this testing module will invoke the methods exposed by the Payment Controller subcomponent in order to test its interaction with the Data Access Utilities component.
- Operators Driver: this testing module will invoke the methods exposed by the Operators subcomponent in order to test its interaction with the Administrator Controller component. This driver has to be developed even though it relates to an already existing external component just to avoid damaging the actual component, present in the old system, during the testing phase of the new system.

In most cases, the bottom up approach does not require the usage of program stubs. However, a full test of the system is not possible without introducing a few of them. In fact, there is a mutual dependency between the clients, which send requests (e.g. from their mobile application) and the system, which replies to them. Moreover, the same concept applies for the cars and the system, that are in a constant mutual dependency. Because of this, we need to introduce stubs to simulate the presence of clients and cars, until they eventually physically become part of the architecture. A practical way to implement this concept could easily be to simply make these program stubs to write on a log that they have correctly received messages and commands from the server.

5.2 Test Data

In order to perform the integration tests that we have previously defined, some special test data are needed in favor of an exhaustive testing activity:

- A list of both valid and invalid candidate rides to test the Ride Controller component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - Cars to ride that are not in the database
  - Users that want to drive the car that are not in the database

- A list of possible commands to test the OnBoardAndroidDevice component. The set should contain instances exhibiting the following problems:
  - Locking doors command when doors are already locked
  - Locking doors command when the driving seat is still in use or there is at least one passenger inside the car
  - Locking doors command when at least one of the doors is still open

- A list of both valid and invalid candidate locations to test the Map Controller component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - Locations outside of the city's boundary
  - Locations that belong to two overlapping safe areas

- A list of both valid and invalid candidate cars to test the Car Controller component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - License plates with an illegal format
  - Car object in each one of the possible states in which it can be

- A list of both valid and invalid candidate payments to test the Payment Controller component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - Payments to commit with a negative amount
  - Payments that do not succeed due to faults on the client side

- A list of both valid and invalid candidate users to test the User Controller component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - Driving license not compliant with the legal format Invalid email address
  - Invalid mobile phone number
  - User's location outside of city's boundary
  - User in each one of the two states in which it can be

- A list of both valid and invalid candidate notifications to test the Notification Controller component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - Email address with an illegal format ○ Phone number with an illegal format

- A list of both valid and invalid candidate cars that need assistance and users to test the Administrator Controller component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - Cars' license plates not compliant with the legal format
  - Users with an invalid email address
  - Cars' and users' locations outside of the city's boundary

Other information regarding the required test data can be found in the input columns of all the test cases that were presented in chapter 3 of this document.

6.Effort spent

To redact this document, each of the group members has spent approximately 15 hours.