POLITECNICO
MILANO 1863

Computer Science and Engineering
Project of Software Engineering 2

PowerEnJoy

Integration Test Plan Document

Authors:

Alberto Zeni 813977

Manuel Parenti 876085

Reference Professor: Mottola Luca

Academic Year 2016–2017

Table of Contents

1 Introduction

1.1 Purpose and Scope

This document represents the integration testing document for PowerEnJoy.

The purpose of this document is to outline the integration of the integration testing activity for all the components of the system. Providing a good testing strategy is necessary in order to have a working system with no unexpected behaviors. Integration testing is necessary because each subsystem must operate correctly with all the subsystems that it interacts with. In this document we'll focus on this specific elements:

- A list of all the subsystems in PowerEnJoy and all their subcomponents involved in integrated activities that we'll test.

- The status of development that the project must have reached before the integration testing of determined subsystems may begin.

- The order in which all the components will be integrated.

- The description of each integration testing approach step and the reasons behind it, including input pre-determined inputs and expected outputs.

- The tools we'll plan to use during the testing activities combined with a description of all the environments in which the testing activities will be performed.

1.2 Glossary

SUBSYSTEM: functional unit observed at high level of the system.

COMPONENT: each of the logical grouping of functions that provides functionalities of the subsystem.

USER: A person who is registered on PowerEnJoy, he can register, find information about his account, search for a car and use the car sharing service using a web application via a browser or using a mobile application. Users are provided with an account ID and a password which they need to use to access the service. Sometimes in the document they are referred to as Customers.

SAFE AREA: Public parking areas defined by the PowerEnJoy company, these won't cause any police officer to write a ticket for the car or even remove the car.

SPECIAL PARKING AREA: These areas are defined by the PowerEnJoy company and they are provided with electric plugs for charging the battery of the cars.

RESERVATION: This term refers to the possibility of reserving a car for a specific user. He can do this with the web application or the mobile app, after he finds a car he wants to pick up. The cars will be reserved for that user for an hour, and after the time expires if the user doesn't pick up the car he is charged of a small fee. Users can reserve a car for up to an hour before they need to drive.

DISPATCHER: A person or a system that will manage the parking and recharging of the cars. The dispatcher will send operators on field to do the operations required in order to overcome the eventual bad behavior of some users.

GPS: The Global Positioning System is a global navigation satellite system that provides geolocation and time information to a GPS receiver in all weather conditions, anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites.

SENSOR: A sensor is a device that converts real world data (Analog) into data that a computer can understand.

1.3 Reference Documents

Project description document: PowerEnJoy Requirement Analysis and Specification Document (RASD)

Design description Document: Design document (DD)

Assignments AA 2016-2017.pdf

2 Integration strategy

2.1 Entry Criteria

The integration testing process must be done after some requisites of the project have been already achieved, otherwise it will be difficult to perform and it will produce meaningless results. The Requirement Analysis and Specification Document (RASD) and the Design Document (DD) must be already discussed and completed at the point the testing will be performed. This two documents are required since they describe each part of the system we want to provide and the functionality of each component in it, without them we'll not know what to test and how to test it. Other than this the integration testing process can be performed only after some components have already reached a precise stage in development of their functionalities, otherwise it will be useless.

These are the requirements of the completion of each component:

- 100% of the **Database Component**
- 90% of the **Car Management subsystem**
- 90% of the **Safe Area**, **Reservation** and **Ride components**
- 80% of the **Payment component**
- 70% of the **User and Operator Management subsystems**
- 60% of the **client applications** (user, operator and dispatcher)

Every component that has to be tested in integration with others must be near to completion, with all its main functionalities already finished (and the ones that can be tested in a unit test should have been tested already).

We need at least a percentage of 60% for the **client apps** in the sense that we only need the main functionalities to be implemented and we don't really need a perfect graphical interface and the maximum optimization of the applications at this stage of testing.

Later on, though, we will need definitive interfaces to be tested by a human interface testing team that will report on the usability and touch-and-feel of the interfaces, as well as the performance of the applications in a sort of real environment.

2.2 Elements to be integrated

In our Design Document we divided the PowerEnJoy system in different components and some of them have a complex combination of subsystems. In this paragraph we define which are the elements that have to be integrated with others to make the system work, so that they can be tested together, when they meet the entry criteria described above.

In particular the Business Logic or Application Server component is the most complex because it has to offer a wide range of functionalities, while communicating with different devices and external components. So we will need some integration testing on the internal subsystems/components and then we will have to test the application's functionalities at an higher level, testing the interaction between clients and other high level components with the application server.

In order to offer their functionalities, these subsystems of the Application Server component need a collaboration between more than one component/subsystem:

- **User Management System**: User Info, User Controller, Payment Controller
- **Car Management System**: Car Info, Car Controller
- **Reservation Management System**: User Controller, Car Controller, Reservation Controller, Safe Area Controller, Ride Controller
- **Operator Management System**: Operator Info, Operator Controller
- **Dispatcher Management System**: Operator Management System, Dispatcher Controller, Car Management System

At an higher level of view we introduce the most important integrations between external components and the Application Server subsystems:

- **User App** and **User Management System**
- **User App** and **Reservation Management System**
- **Operator App** and **Operator Management System**
- **Dispatcher Service** and **Dispatcher Management System**
- **Car Component** and **Car Management System**
- All Database dependent components/subsystems with the **Database Server Component**

With **User App** we mean that all the functionalities offered by the User Application have to be tested, on every device they can be used and in the different graphical interfaces offered.

2.3 Integration Testing Strategy

Given the analysis of the items that need to be integrated it is a natural choice to use a bottom-up approach during the integration testing phase. We have different subsystems that depend on low level components and these components offer their functionalities to different subsystems. With a bottom-up approach we can develop the lower level components first and then reuse them as many times as we want to create higher level subsystems and to test the interactions with them. Since this kind of dependency is really strong it is not recommended to follow a top-down approach, because there would be the necessity of creating different stubs for a single component. These stubs could behave differently and may not be compatible, with a bottom-up approach we can develop the PowerEnJoy system in a safer way. Surely we will need some drivers but they would be less than the possible stubs and the team that develops one of the Management Systems only has to deal with one driver, so the possible inconsistencies are really reduced.

We also follow an approach that exposes the risks in the most critical subsystems as early as possible, testing them before going through the other subsystems' tests. This way we can recover faster from errors and have more time to notice them.

In our case the most critical subsystems are the **Car**, **User** and **Reservation Management Systems**, because they are at the heart of the whole service.

The **Database Component** needs to be ready before the others, because a lot of subsystems depend on it, so the integration between them should be tested early in the testing phase. Then the integration testing of the subsystems that are contained in the Application Server Component can begin, leading to the integration with the high level components.

2.4 Sequence of Component/Function Integration

In this paragraph we make explicit the order of the integration tests of components and subsystems composing the PowerEnJoy application, starting from the analysis of the elements and the testing approaches described in the previous paragraphs.

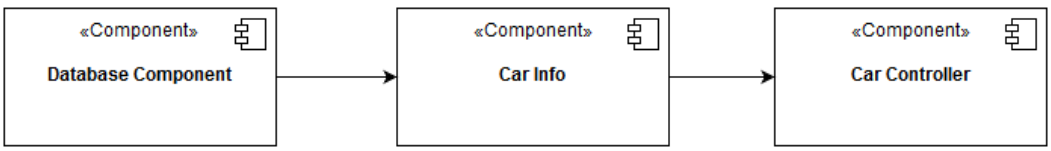
2.4.1 Software Integration Sequence

In this paragraph, for every high level subsystem that we defined previously, we are going to describe the order in which the software components have to be integrated for testing. These subsystems are all part of the main component (Application Server) and include the communication with the Database Server, the interaction between these subsystems and external ones is described in the following paragraph.

Car Management System

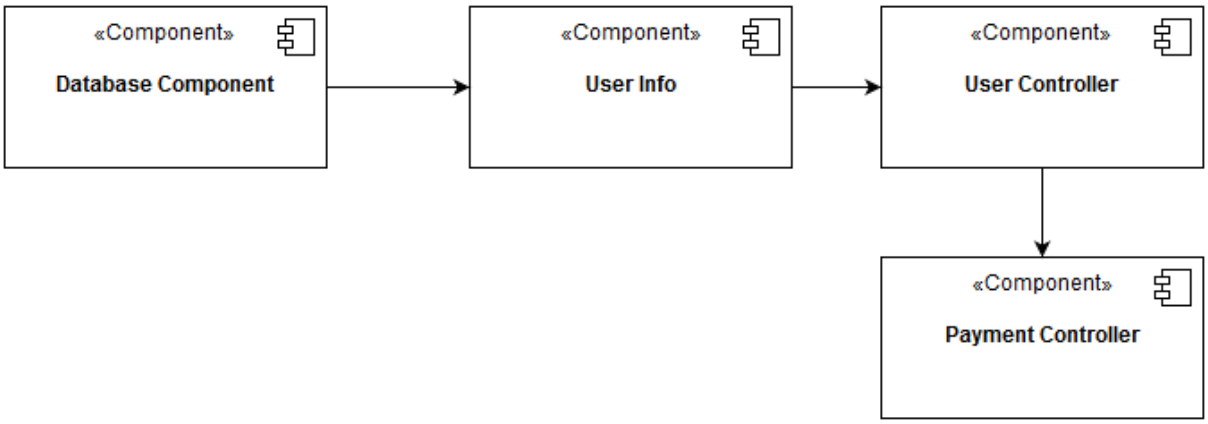
We start from the most critical subsystems, building up the integration testing of the Car Management System, which involves 3 software components.

After the functionalities offered by the Database Component have been tested, the Car Info component can be added to the testing environment. After the 2 components have been tested together, the Car Controller component can be added up to end the integration testing of this subsystem.



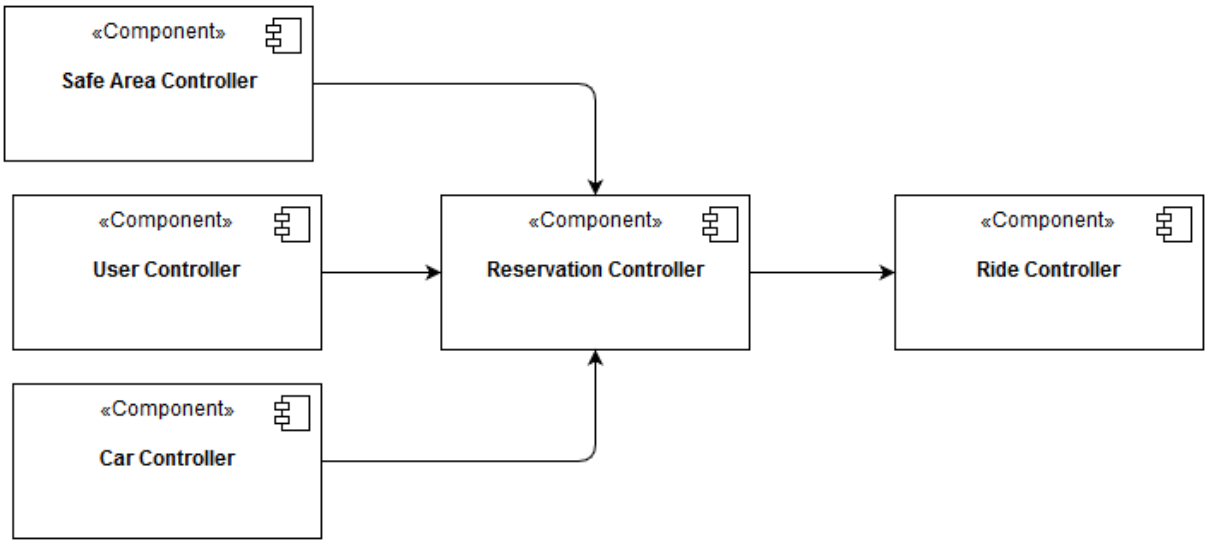
User Management System

Like the previous subsystem we have to test the interaction between the database and the user's information component, and the interaction between this kind of information and the component that manages the changes that can be applied to it (User Controller). After that the Payment Controller should be added to the testing environment because it offers the functionalities to use the payment information contained in User Info and to change the user's balance.



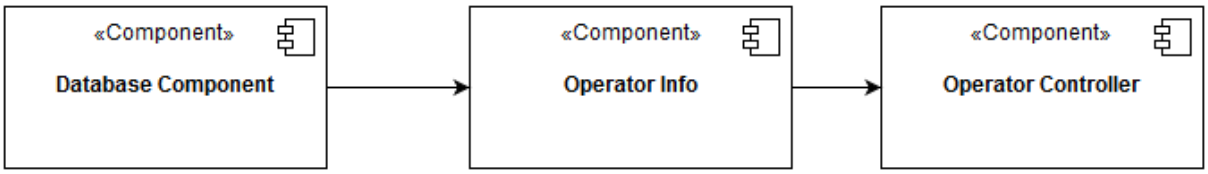
Reservation Management System

In this subsystem we have different components that need to cooperate to create a reservation and make a ride. So the basic components (Safe Area, User and Car Controllers) need to be present before the Reservation Controller is added to the integration testing. The same goes for the Ride Controller, that needs the reservations as well for the functionalities it has to offer.



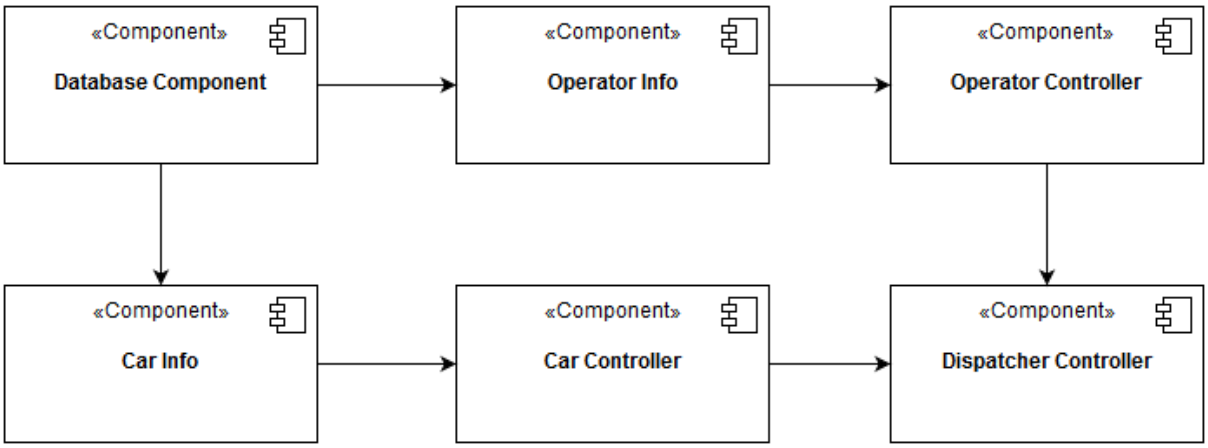
Operator Management System

This subsystem is the first to be non-critical, and it needs to be tested in a very similar way to the User Management System.



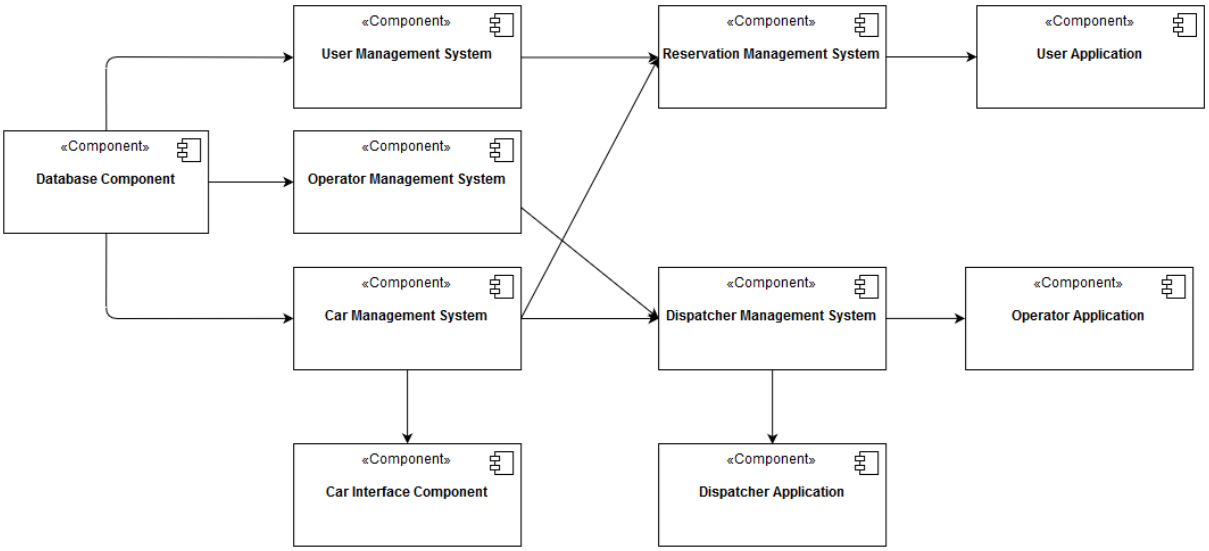
Dispatcher Management System

The dispatcher subsystem works on the shoulders of the components that manage the information of operators and cars, so to test it we need those components that can be used by the dispatcher for its operations. Therefore the Dispatcher Controller should be added to the testing phase as the last one.



2.4.2 Subsystem Integration Sequence

This diagram describes the order in which the high level subsystems should be integrated, reaching a point in which a test of the entire system can be run.



3 Individual Steps and Test Description

Each subsection indicates a pair of components that needs to be tested, the first is the component that has the methods that are invoked, and the second is the component which the first one refers to.

3.1 User Management System

3.1.1 User Controller, DB Server

requestUserInformations(u: User)	
Input	*Effect*
Null Parameter	NullException is raised
Not existent user is inserted	NoUserException is raised
Formally correct info is inserted	Information about the user are returned

insertUserInformations(u: User)	
Input	*Effect*
Null Parameter	NullException is raised
User information are already present in the DB	AlreadyRegisteredException is raised
Formally incorrect user info is inserted	InvalidInsertionException is raised
Formally correct user info is inserted	Tuple of User added in the DB

updateUserInformations(u: User)	
Input	*Effect*
Null Parameter	NullException is raised
Formally incorrect updates wants to be made	InvalidUpdateException is raised
An non-existent user wants to be updated	NonExistentUserException is raised
Formally correct updates and user data inserted	User data are updated in the DB

3.1.2 User Controller, Payment Controller

rechargeAccount(u: User, p: PaymentMethod)	
Input	*Effect*
Null Parameter	NullException is raised
Wrong payment method info inserted	InvalidPaymentMethodException is raised
Correct payment method info inserted	Grant to recharge account is gained

3.2 Reservation Management System

3.2.1 Safe Area Controller, Reservation Controller

verifiesUserRequests(r: Reservation)	
Input	*Effect*
Null Parameter	NullException is raised
Reservation with an already existent ID is inserted	AlreadyExistentReservationException is raised
Formally incorrect reservation data is inserted	InvalidInsertionException is raised
Formally correct reservation data is inserted	The reservation process proceeds

3.2.2 Safe Area Controller, Car Controller

updateStatus(c: Car, r: Reservation)	
Input	*Effect*
Null Parameter	NullException is raised
Invalid car data is inserted	NonValidCarException is raised
Inserted car data does not correspond to the car data in the reservation	IncorrectInsertionException is raised
Formally correct data wants to be inserted	The car status is updated to “reserved”

3.2.3 Car Controller, Reservation Controller

createReservation(c: Car, r: Reservation)	
Input	*Effect*
Null Parameter	NullException is raised
Car with a “busy” state is inserted	NonValidCarException is raised
Invalid car data is inserted	NonValidCarException is raised
A car wants to be added to a reservation that already has a car assigned	NonValidReservationException is raised
Formally correct data is inserted	Car is inserted in the reservation data

3.2.4 Reservation Controller, Car Controller

getPlate(c: Car)	
Input	*Effect*
Null Parameter	NullException is raised
No Active Reservation has the inserted Car	InvalidInsertionException is raised
An active reservation has the inserted Car	The car plate is returned

3.2.5 Car Controller, Ride Controller

startRide(u: User, c: Car)	
Input	*Effect*
Null Parameter	NullException is raised
User has not unlocked the car	InvalidAssignmentException is raised
The user and car data does not match any active reservation	InvalidRideException is raised
Formally incorrect data is inserted	InvalidInsertionException is raised
Formally correct data is inserted	The ride starts

endRide(u: User, c: Car)	
Input	*Effect*
Null Parameter	NullException is raised
The user and car data does not match any active reservation	InvalidRideException is raised
The car has a different state than “in use”	InvalidCarException is raised
Formally correct data is inserted	The ride ends properly

3.2.6 Car Controller, Safe Area Controller

updateCarsStatus(c: Car)	
Input	*Effect*
Null Parameter	NullException is raised
Not existing car data is inserted	NonValidCarException is raised
Formally correct data is inserted	Cars in the safe area are updated

3.3 Car Management System

3.3.1 Car Controller, DB Server

getCar(s: String)	
Input	*Effect*
Null Parameter	NullException is raised
The plate inserted does not correspond to any car in the system	InvalidCarPlateException is raised
The inserted data is formally correct	Car data is returned

3.4 Operator Management System

3.4.1 Operator Controller, DB Server

getOperatorInfos(s: String)	
Input	*Effect*
Null Parameter	NullException is raised
The inserted SSN does not correspond to any operator in the system	InvalidSSNException is raised
The inserted data is formally correct	Operator data is returned

3.5 Dispatcher Management System

3.5.1 Dispatcher Controller, Operator Controller

checkAvailability(o: Operator)	
Input	*Effect*
Null Parameter	NullException is raised
Invalid operator data is inserted	InvalidOperatorException is raised
Formally correct data is inserted	Availability if the selected operator is returned as a boolean

submitTask(o: Operator, c: Car)	
Input	*Effect*
Null Parameter	NullException is raised
The selected operator is busy with another operation	InvalidOperatorException is raised
The selected car is in an “available” state	InvalidCarException is raised
Formally correct information is inserted	The task is added to the operator schedule

addTask(o: Operator, s: String)	
Input	*Effect*
Null Parameter	NullException is raised

3.5.2 Dispatcher Controller, Car Controller

checkState(c: Car)	
Input	*Effect*
Null Parameter	NullException is raised
The inserted car has formally incorrect data	InvalidCarDataException is raised
The inserted data is formally correct	The state of the passed car is returned

4 Tools and Test Equipment Required

4.1 Tools

JUnit:

The integration testing may be done after the development of (part of) the software components involved in some functionalities of the application. But that is not enough, in the entry criteria we stated out that the functionalities and parts of the software components that can be tested in isolation should be tested in their components.
To perform those unit tests we are going to use the JUnit Framework, since it is the most natural tool to use looking at the choices that we already took for the implementation of the system. JUnit can also be used, alongside other tools, to verify that the functionalities that involve the integration of some software components and subsystems are done in the right way and that the subsystems behave correctly.

Arquillian:

The integration testing framework provided by Arquillian is the tool we need to perform integration testing, because it provides the tools to test containers in an easy and flexible way. Arquillian can be used to execute tests of subsystems that need to be integrated with others and this is done through the execution of test against containers, that may contain complex networks of components.

Jmeter:

To execute performance tests on the components we chose to use Jmeter. It is an application designed to load test functional behavior and measure performance, it can be used to simulate a heavy load on a server or object, to test its strength or to analyze the overall performance under different load types

Manual Testing:

To verify the usability and the overall performance of the application from the point of view of a user we will need help from humans, since the final users will be people we need some feedback from people that try to use the applications. To check these kind of performance requirements we will also take advantage of alpha and beta releases, with some final users submitting their opinions on the product.

4.2 Test Equipment

The devices used to test the client applications need to be representative of the devices that are more likely to be used by the final users. Thus we will need:

- Android smartphones with different screen sizes (from 4.5'' to 6''), different CPUs and GPUs, different OS versions (from 4.4 to 7).
- Android tablets with different screen sizes (from 7'' to 10''), different CPUs, GPUs and OS versions.
- Different versions of iPhones and iPads.
- At least one Apple laptop for each of the latest versions of macOS.
- Some laptops and PCs with different screen sizes, specifications and OS.
- Windows phones with different screen sizes (from 4'' to 6'') and OS versions varying between Windows Mobile 8 and 10.

To test the business logic and database components we will need a cloud environment that is the closest possible to the environment in which our product will run.

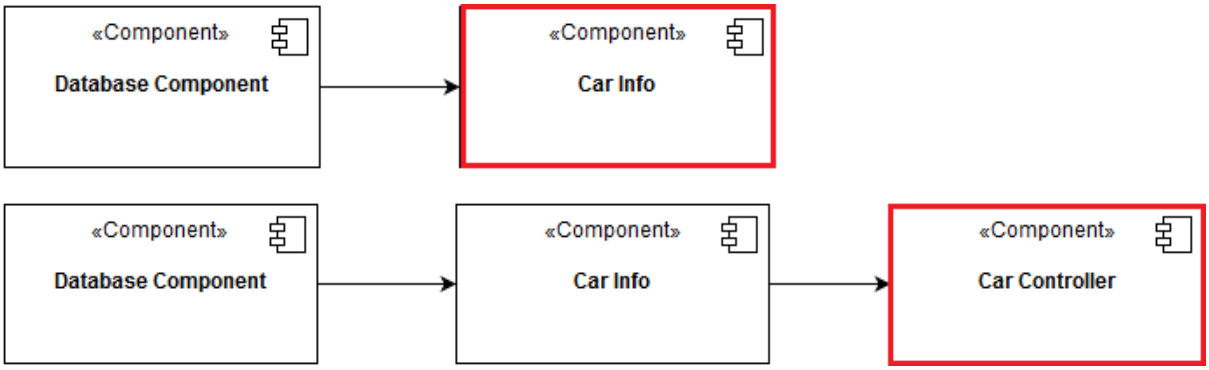
We could benefit from a smaller version of the running environment of the PowerEnJoy system hosted by the same company that will host the system.

5 Program Stubs and Test Data Required

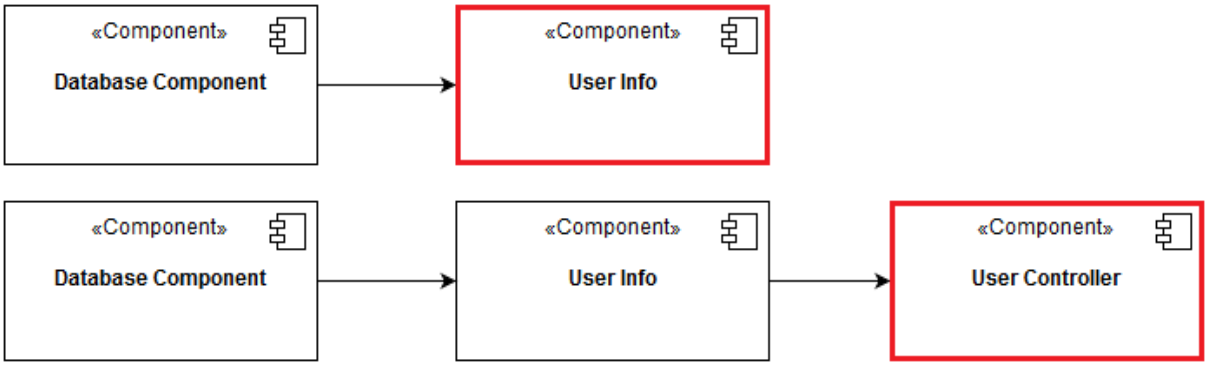
5.1 Stubs and Drivers

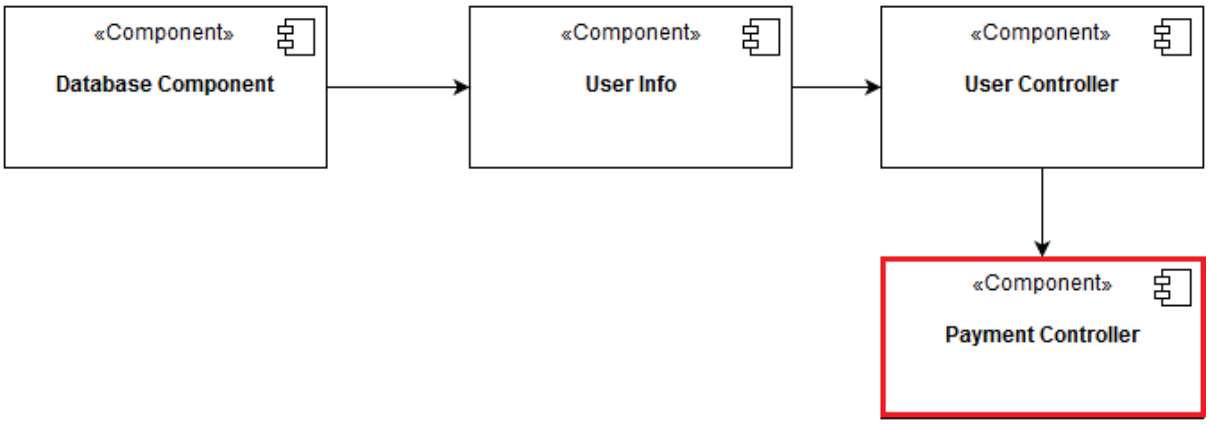
Since we followed a bottom-up approach for the integration testing strategy we are going to need a list of drivers to simulate the behavior of “non top-level” components that will call the functions of the bottom-level components, to check if they work correctly.
Below we have a graphical representation of the list of drivers that we are going to need for each subsystem.
The red components are the drivers needed to simulate the functionalities offered by the component they represent, in order to interact with the components that stand below them and test them in the context of the entire subsystem.

Car Management System:

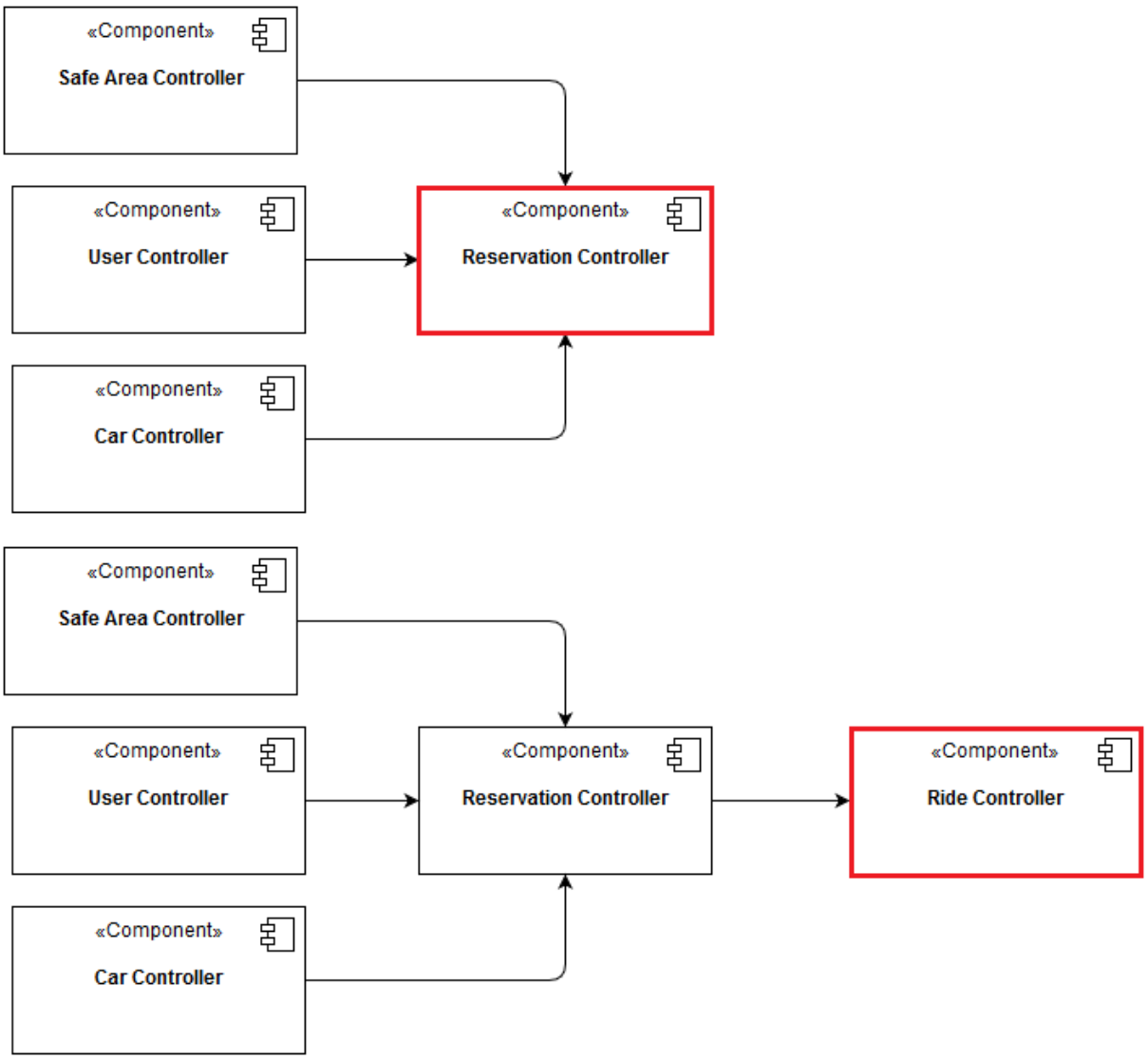


User Management System:

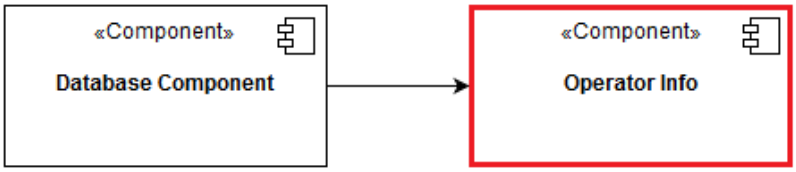


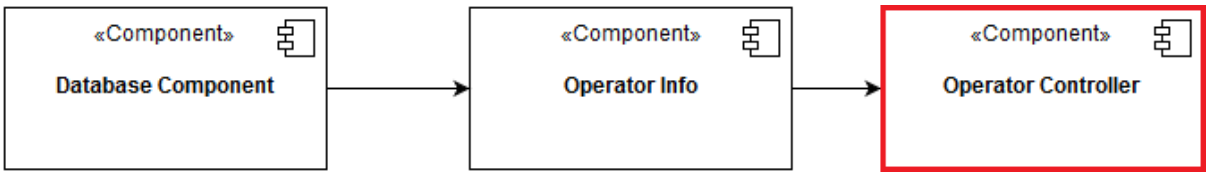


Reservation Management System:

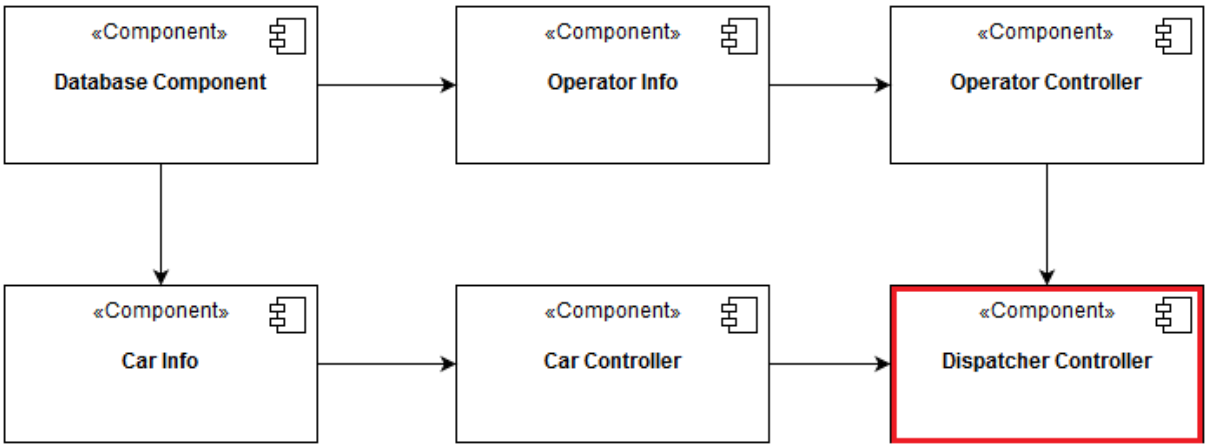


Operator Management System:





Dispatcher Management System:



5.2 Test Data

Since we want to perform all the integrated tests indicated above, we are going to need both valid and invalid data values to test each component. This is what we need:

- A set of users that contains both valid and not valid candidates for our system to test the **User Management System**. This set must contain instances that underline the following problems:
    - Null objects
    - Null Fields
    - Invalid SSN
    - Invalid email address
    - Invalid Driving License
    - Invalid Phone Number

- A set of reservations that contains both valid and not valid candidates for our system to test the **Reservation Management System.** This set must contain instances that underline the following problems:
    - Null objects
    - Null Fields
    - Invalid Safe Area
    - Invalid Car
    - User without a positive account balance

- A set of cars that contains both valid and not valid candidates for our system to test the **Car Management System.** This set must contain instances that underline the following problems:
    - Null objects
    - Null Fields
    - Invalid Plate Number

- A set of operators that contains both valid and not valid candidates for our system to test the **Operator Management System.** This set must contain instances that underline the following problems:
    - Null objects
    - Null Fields
    - Invalid SSN

- A set of dispatchers that contains both valid and not valid candidates for our system to test the **Dispatcher Management System.** This set must contain instances that underline the following problems:
    - Null objects
    - Null Fields
    - Invalid Operator Data
    - Invalid Car Data

6 Additional Information

6.1 Hours of work

Time spent to write this document:

Manuel Parenti : 20 hours

Alberto Zeni : 20 hours

6.2 Used Tools

The tools that supported us are:

MS Office Word 2016: to write and redact this document;

Draw.io : to create the diagrams.