Integration Test Plan Document

Software Engineering 2

January 15,2017

POLITECNICO
MILANO 1863

Authors:

Claudio Salvatore	ARCIDIACONO	Matr 879109
Antonio	DI BELLO	Matr 878786
Denis	DUSHI	Matr 879115

Contents

1 Introduction

1.1 Revision history

1.2 Purpose and scope

The purpose of this document is to give to the test team some precise guidelines to follow in order to accomplish the testing phase. In particular this document will focus on the integration of the various components presented in the **Design Document**. These tests aim to find possible errors in the cooperation between components. In the following sections are provided which criteria must be met before starting the integration test phase, the elements to be integrated, the strategy that will be used during the integration test phase, the sequence of integration. Then for each integration between components or subsystems will be specified the tests to run and the expected behavior of our system, the stubs that must be implemented and the test data to be produced in order to accomplish these tests.

1.3 List of definitions and abbreviations

- **RASD:** Requirements analysis and specifications document
- **DD:** Design Document

1.4 List of reference documents

- RASD Document
- Assignment AA 2016-2017.pdf
- Integration testing example document.pdf
- Integration Test Plan Example.pdf

2 Integration strategy

2.1 Entry criteria

In order to perform the integration testing of **PowerEnJoy** the following criteria must be met:

- The Requirements Analysis and Specification Document (**RASD**) and the Design Document (**DD**) must be fully written
- The components :
  - UserController
  - MapController
  - ReservationController
  - RentController
  - FareController
  - CarController
  - DistributionOptimizer
  - CarRemoteController
  - ClientApp

  must be developed and unit tested
- Unit testing must be performed on the **Model** classes.
- All the high prioritized faults and bugs found during unit testing must have been fixed

2.2 Elements to be integrated

The elements to be integrated are the components presented in the design document.

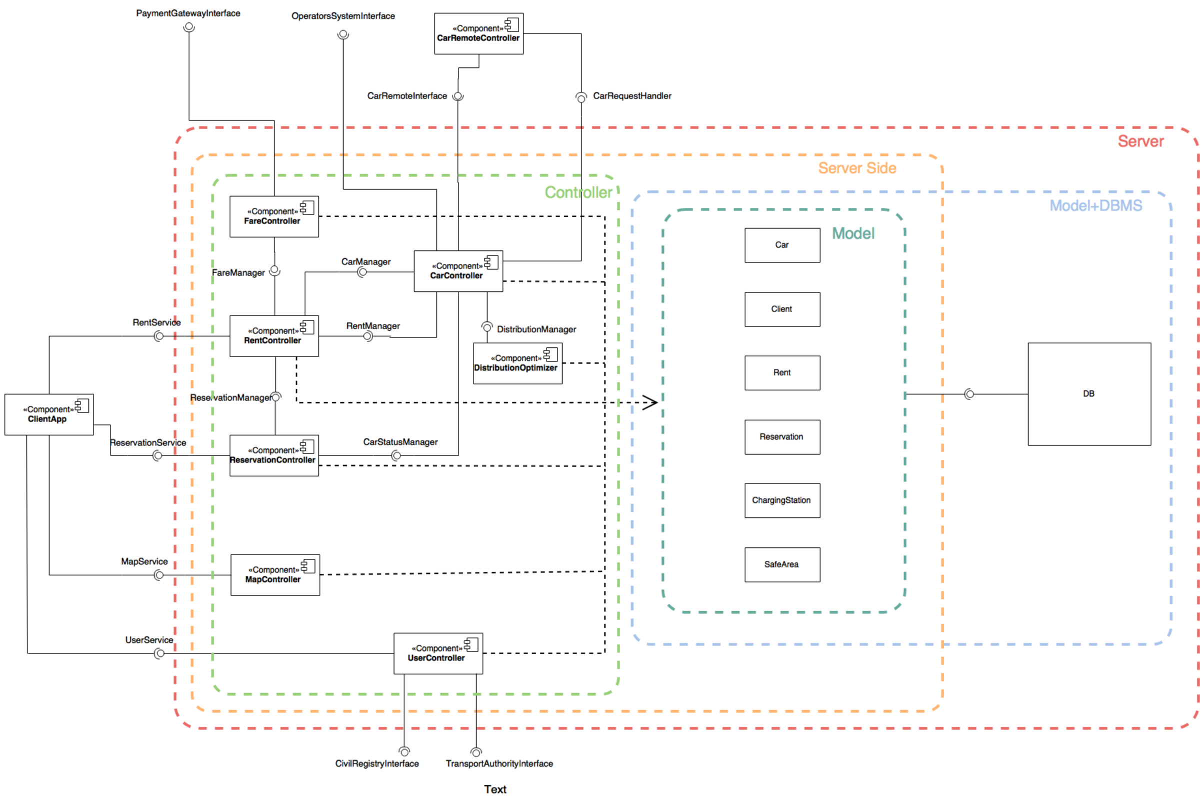


Figure 1: Component diagram.

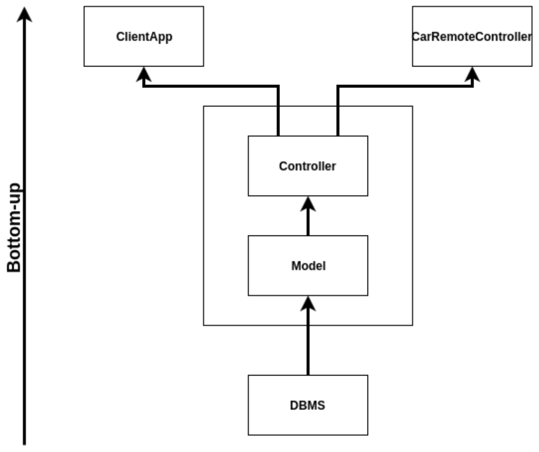
Starting from the components of the diagram above we have identified the following subsystems :

- **Controller:** a set of all the controller components on the server side
- **Model:** composed by all the model components on the server side
- **Model + DBMS** : composed by the Model and the DBMS
- **Server:** composed by the integration of the Controller with the subsystem Model+DBMS

The first integration is between the **Model** components, however the only interaction between them are getters and setters so we can integrate all these components without testing them. From now on we will proceed considering all the model components as an unique component named just **Model**. The next components to be integrated is the **DBMS** and the **Model**, obtaining a subsystem called **Model+DBMS**. Now we have to integrate the **Controller** components to the **Model+DBMS** subsystem. Then we can proceed to the integration between the controller components taking into account the yet succeeded integration between each of the **Controller** modules and the **Model+DBMS** subsystem. The result of this phase will be the **Server** subsystem. The next step is to integrate the remaining standalone components : the **Client** and the **CarRemoteController**. In order to accomplish this task will be adopted a functional approach. We assume that the teams that will develop the components that use external services will test the external interfaces during the unit testing phase using some tools given by the service provider, so the integration of the external interfaces does not need to be tested again and they will be integrated directly to each component that makes use of them.

2.3 Integration testing strategy

Due to the complexity of the project and the fact that it is deployed on several hardware devices we are going to use different integration testing approaches. On a high level we are going to use a **bottom-up** approach to integrate the subsystems of **PowerEnjoy**. This incremental approach allows us to detect faster faults and bugs (compared to approaches like "big bang") and to evaluate easily the testing progress.



As written in the "Entry Criteria" all the components of the **Model** have already been tested and they don't need integration testing because interacts only with getters and setters. Therefore we are going to consider the model as a unique component.
The first step consist on the integration of the **Model** with the **DBMS** to test their interaction. **DBMS** is a commercial component that has already been developed so it can be immediately used in a bottom-up approach without any explicit dependency. All the components of the **Controller** subsystem will be first integrated and tested with the **model** in order to facilitate future integration testing. This can be done separately and independently in order to save time.
To integrate between them the components of the **Controller** subsystem we are going to use a "**Critical Module First**" approach. Starting from the riskiest and most connected parts will help us to detect bugs as soon as possible, but we will also need to develop some stubs and drivers. We will start integrating CarController with :

- ReservationController
- RentController
- DistributionOptimizer

Also these integrations will be done independently using appropriate stubs and drivers. We will procede integrating ReservationController with RentController and, after testing,integrating also FareController.

Once the Server has been fully tested, we are going to test the interaction between the three subsystems:

- ClientApp
- CarRemoteController
- Controller + Model + DBMS (= Server)

In this phase a Big Bang approach would be too dispersive and would not guarantee a proper visibility on the functionalities because the system is big and presents complex interactions among his subsystems. Even if the integration may result more complex, we opt for a **functional grouping** approach that will require less stubs/drivers and will provide better process visibility. Portions of the different subsystems interact together to provide the following user-visible functions:

- Reservation
- Start Rent

- Map visualization
- User Account Management
- End Rent
- Damage Report
- MoneySavingOption Request

Another advantage of this approach is that functions can be tested separately and independently by different testing teams in order to speed up the testing process.

2.4 Sequence of component/function integration

In the following sub-paragraphs will be explain the order in which the tests between various components and subsystems will be performed.

2.4.1 Software integration sequence

ID	Integration Test	Paragraph
I2T1	(DBMS + Model) → MapController	3.2
I2T2	(DBMS + Model) → UserController	3.2
I2T3	(DBMS + Model) → CarController	3.2
I2T4	(DBMS + Model) → DistributionOptimizer	3.2
I2T5	(DBMS + Model) → ReservationController	3.2
I2T6	(DBMS + Model) → RentController	3.2
I2T7	(DBMS + Model) → FareController	3.2
I3T1	CarController→ DistributionOptimizer	3.3
I3T2	CarController→ ReservationController	3.3
I3T3	CarController→ RentController	3.3
I4T1	ReservationController→ RentController	3.4
I5T1	RentController→FareController	3.5

As already mentioned, the integration between components follows a mix of bottom-up and critical-module-first approach. In the diagrams below, we focus on the component of the server side. The direction of the arrows indicates the order of integration. First of all, the model will be integrated with all controller component, and, for speed up this phase, this can be do in in parallel. This will allow every component to access to data needed for the other tests. On the next step we will integrate the components of the Controller subsystem. We will first integrate CarController with the interacting components: DistributionOptimizer ReservationController, RentController. Finally we will perform the integration between ReservationController and RentController and between RentController and FareController . The numbers that labels the edges indicate the sequence of integration. Notice that this order coincide with a topological order of the graph.

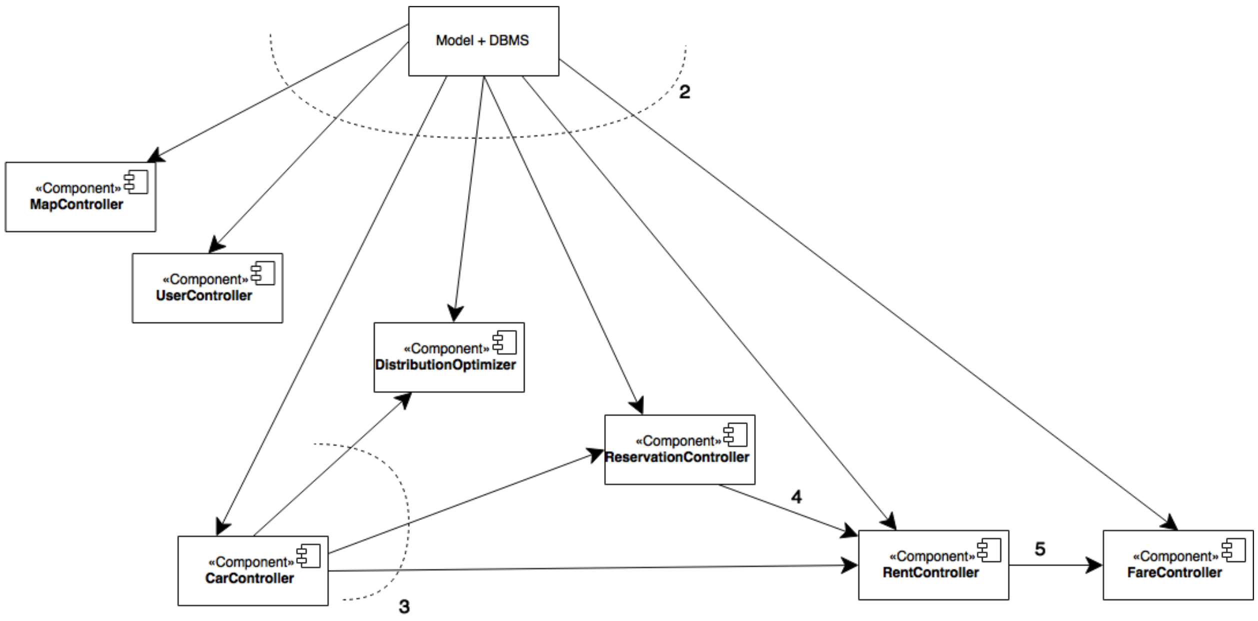

Model + DBMS
2
«Component» MapController
«Component» UserController
«Component» DistributionOptimizer
«Component» ReservationController
«Component» CarController
3
4
«Component» RentController
5
«Component» FareController

Figure 2: Integration strategy server side.

2.4.2 Subsystem integration sequence

ID	Integration Test	Paragraph
I1	DBMS → Model	3.1
I2	(DBMS + Model) → Controller	3.2
I6	Client → Server → CarController	3.6

PowerEnJoy can be divided into different subsystems, as seen in Design document. Since that the integration of the subsystems could hide various problems, we have decided to use a functional grouping approach. This strategy allows us to obtain a general view of the desired effects over the full system.

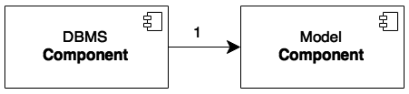


Figure 3: Model + DBMS subsystem.

First of all, to make relevant all the following tests, we must test the integration between **DBMS** and **Model** subsystem. Then we focused on the **Controller** subsystem, as mentioned above, and integrate it with **DBMS** + **Model**. After that, we integrate the other server components. This integration has as result the **Server** subsystem. Finally the last integration will be between the Server subsystem and the remaining components :**CarController** and **ClientApp** .

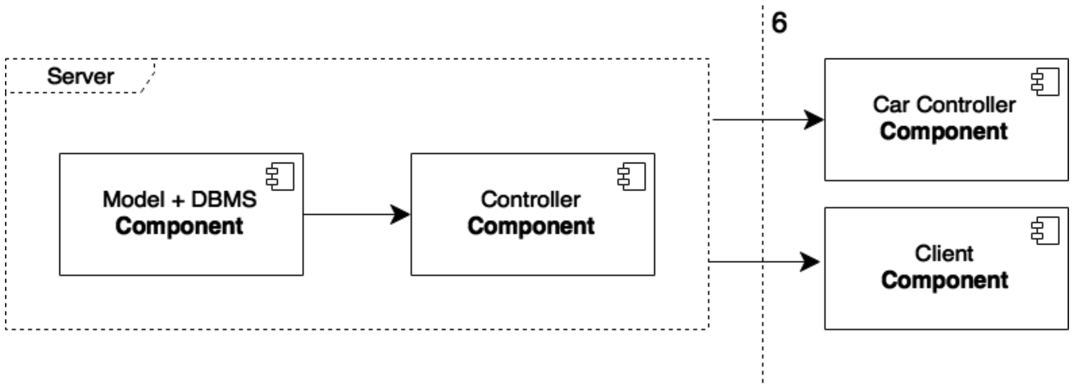


Figure 4: Integration strategy subsystem.

3 Individual steps and test description

3.1 Integration test case I1

Test Case Identifier	I1T1
Test Item(s)	Model and DBMS
Input Specification	Car,User,Rent,Reservation,ChargingStation,SafeArea
Output Specification	The Database queries are correctly called by the methods of the Model and there is referential integrity between the data in the database and the data of the model
Environmental Needs	N/A
Dependencies	Test data to fill the Database

3.2 Integration test case I2

Test Case Identifier	I2T1
Test Item(s)	Model+DBMS and MapController
Input Specification	a gps position and a distance
Output Specification	correct information about the indicated area like cars available, safe areas and charging stations
Environmental Needs	MapServiceDriver
Dependencies	test data about map elements

Test Case Identifier	I2T2
Test Item(s)	Model+DBMS and UserController
Input Specification	personal data and payment information in a correct format
Output Specification	the system calls the right methods in order to insert the new user data into the database and allow the new user to use the service
Environmental Needs	UserServiceDriver
Dependencies	N/A

Test Case Identifier	I2T3
Test Item(s)	Model+DBMS and CarController
Input Specification	Car, User
Output Specification	The system invokes the right methods in order to change the car information in the model
Environmental Needs	CarManagerDriver, CarRequestHandlerDriver, CarStatusManager-Driver
Dependencies	CarRemoteController Stub,RentController Stub

Test Case Identifier	I2T4
Test Item(s)	Model+DBMS and DistributionOptimizer
Input Specification	gps position
Output Specification	The distributionOptimizer is able to get information about cars within a certain area
Environmental Needs	DistributionManagerDriver
Dependencies	Test data about cars

Test Case Identifier	I2T5
Test Item(s)	Model+DBMS and ReservationController
Input Specification	Reservation, User, Car
Output Specification	The ReservationController invokes the right methods of the model in order to add a new reservation of the specified user for the specified car car
Environmental Needs	ReservationServiceDriver, ReservationManagerDriver
Dependencies	test data about users and reservations, CarController stub

Test Case Identifier	I2T6
Test Item(s)	Model+DBMS and RentController
Input Specification	Rent,Reservation, User, Car
Output Specification	The RentController invokes the right methods of the model in order to add a new rent of the specified user for the specified car preceded by the specified Reservation
Environmental Needs	RentServiceDriver , RentManagerDriver
Dependencies	test data about rents and reservations and cars, FareControllerStub, CarControllerStub

Test Case Identifier	I2T7
Test Item(s)	Model+DBMS and FareController
Input Specification	Rent,User,Car
Output Specification	The FareController acquire the needed information about the user
Environmental Needs	FareManagerDriver
Dependencies	test data about users

3.3 Integration test case I3

Test Case Identifier	I3T1
Test Item(s)	CarController and DistributionOptimizer
Input Specification	a gps area
Output Specification	The CarController invokes the right methods of DistributionOptimizer
Environmental Needs	I1T3 and I1T4 succeeded, CarRequestHandlerDriver
Dependencies	Test Data about cars

Test Case Identifier	I3T2
Test Item(s)	CarController and ReservationController
Input Specification	Car
Output Specification	The ReservationController invokes the right methods of CarController in order to change the status of the specified car
Environmental Needs	I1T3 and I1T5 succeeded, ReservationServiceDriver
Dependencies	CarRemoteController Stub

Test Case Identifier	I3T3
Test Item(s)	CarController and RentController
Input Specification	Car
Output Specification	The RentController invokes the right methods of CarController in order to unlock the doors of the specified car
Environmental Needs	I1T3 and I1T6 succeeded, RentServiceDriver,CarRequestHandlerDriver
Dependencies	CarRemoteController Stub, FareControllerStub, ReservationControllerStub

3.4 Integration test case I4

Test Case Identifier	I4T1
Test Item(s)	RentController and ReservationController
Input Specification	Car,User,Reservation
Output Specification	The RentController invokes the right methods of ReservationController in order to modify the status of the reservation made by the specified user for the specified car
Environmental Needs	I1T5, I1T6, I2T2 and I2T3 succeeded, RentServiceDriver
Dependencies	CarRemoteController Stub, FareController Stub, test data about reservation, users and cars

3.5 Integration test case I5

Test Case Identifier	I5T1
Test Item(s)	RentController and FareController
Input Specification	Car,User,Rent
Output Specification	The RentController invokes the right methods of FareController in order to start the payment operations.
Environmental Needs	I1T6, I1T7, I2T3 and I3T1 succeeded,RentManagerDriver
Dependencies	test data about rent and users

3.6 Integration test case I6: Reservation

Test Case Identifier	I6T1
Test Item(s)	ClientApp and Server and CarRemoteController
Input Specification	Client,Car
Output Specification	Server stores the request if all condition about client profile and car status are satisfied, then modify the status of the car invoking the right method of CarRemoteController and starts the reservation countdown.
Environmental Needs	I1...I5 succeeded.
Dependencies	

3.7 Integration test case I7: Start rent

Test Case Identifier	I7T1
Test Item(s)	ClientApp and Server and CarRemoteController
Input Specification	Client,Car
Output Specification	The server checks if exist a valid reservation for (Client,Car) and after that, sends a remote command to the car to unlock the doors.
Environmental Needs	I1...I5 succeeded.
Dependencies	N/A

3.8 Integration test case I8: Map visual

Test Case Identifier	I8T1
Test Item(s)	ClientApp and Server
Input Specification	User, position
Output Specification	Server collects information about the available cars near the position, then send it to ClientApp.
Environmental Needs	I1...I5 succeeded.
Dependencies	N/A

3.9 Integration test case I9: User Account Management

Test Case Identifier	I9T1
Test Item(s)	ClientApp and Server
Input Specification	User
Output Specification	User submit his username and password, Server checks if the credentials are valid and allows the client to access to the system.
Environmental Needs	I1...I5 succeeded.
Dependencies	N/A

Test Case Identifier	I9T2
Test Item(s)	ClientApp and Server
Input Specification	User
Output Specification	User submit his personal data, the ClientApp checks if they are valid, the Server manages the request and register a new user.
Environmental Needs	I1...I5 succeeded.
Dependencies	N/A

3.10 Integration test case I10: End rent

Test Case Identifier	I10T1
Test Item(s)	Server and CarRemoteController
Input Specification	Request of end rent on CarRemoteController passing a GPS position that doesn't correspond to a safe area
Output Specification	The end of the rent is not allowed, is called a method on CarRemoteController that notifies the user on the car display.
Environmental Needs	I1...I5 succeeded.
Dependencies	N/A

Test Case Identifier	I10T2
Test Item(s)	Server and CarRemoteController
Input Specification	Request of end rent on CarRemoteController passing a GPS position corresponding to a safe area
Output Specification	The right methods are called in CarRemoteController and in Server in order to : start the "lock doors" countdown, lock the car's doors, check if the car is plugged,save the rent information,calculate the discounts based, charge the client.
Environmental Needs	I1...I5 succeeded.
Dependencies	N/A

3.11 Integration test case I11: Damage Report

Test Case Identifier	I11T1
Test Item(s)	Server and CarRemoteController
Input Specification	Request of report damage/dirty car on the CarRemoteController
Output Specification	The right methods are called in CarRemoteController and in Server in order to: save the information of the report, acquire information about the current and previous rent and send them to the operator system,enable opportunity to end rent without charge.
Environmental Needs	I1...I5 succeeded.
Dependencies	N/A

3.12 Integration test case I12: MoneySavingOption Request

Test Case Identifier	I12T1
Test Item(s)	CarRemoteController and Server
Input Specification	Request to enable "Money saving option" on the CarRemoteController passing the destination address
Output Specification	The right methods are called in CarRemoteController and in Server in order to calculate and return information about the charging station where to plug the car based on destination address,cars distribution,charging spots availability.
Environmental Needs	I1...I5 succeeded.
Dependencies	N/A

4 Tools and test equipment required

In order to simplify and speed up the testing phase, we are going to use some automated tools. Furthermore, the test must be run in a specific environment with a reasonable data, these must approximate as much as possible the expected reality.

4.1 Tools

For testing the business logic, that could be written in Java EE, we can use two main tools that work together, **Arquillan** and **JUnit**.
Arquillan is a tool that help to verify the correct interaction between a component and its execution environment. Therefore we will use this tools in order to identify eventually problems into a Java container that has deployed on server side. More precisely, we check the right component injection where another component need them.
Even if JUnit framework is used during unit test, it can be also used as support to verify that during an interaction between components the right effect will be produced. For example, we use it to check that the correct object is returned when a method is invoked or to assure that an exception is raised when the invoked method is invalid.
Finally one other tool that can be useful is **Mockito**. This tool help us to create the stub necessary during the integration test. In particular, when a component to be called isn't yet implemented, Mockito easily allows us to define a fake response for the caller.
Furthermore, particularity attention will be posed on the performance that the system should be guarantee during certain phase. For example, it's requested a high reactivity during the operation of start a rent and end a rent. For this purpose we can use **JMeter**, that allows us to load the system with a reasonable quantity of request and verify that the system performance meets a certain criteria of performance.

4.2 Test equipment required

In order to perform a valid integration tests, an environment with a minimum specific requirement is needed. The PowerEnJoy system is made up by a different types of hardware component, so during the test phase it's required an heterogeneous representation of them. Surely, in addition to server and database that constitute the core of the system, is required a car provided by equipment necessary to interact and communicate remotely. The most critical aspect is represented by the choice of the client device. In fact, if for testing the web application can be used a little set of the most used browser, for the mobile app it's necessary to use more configurations. For a minimum coverage of possible real situations is requested at least one mobile device for each OS, and it's recommended to use different display size. Finally should be recreated special situations like for example those in which there is low data connectivity or the GPS signal is absent.

5 Program stubs and test data required

5.1 Stubs

Since we are using a critical-first strategy we need 5 stubs:

5.1.1 CarRemoteControllerStub

Usages:

- I2T3
- I3T2
- I3T3
- I4T1

Description This stub emulates the behavior of the car system, it is used by the CarController to achieve several tasks:

- Updating periodically the car position in the model
- Opening and closing the car doors after a client request
- Changing the car status

5.1.2 CarControllerStub

Usages:

- I2T5
- I2T6

Description This stub emulates the behavior of the CarController component, it is used by the ReservationController to check and eventually change the car status after a new request of rent or reservation.

5.1.3 FareControllerStub

Usages:

- I2T6
- I3T3
- I4T1

Description This stub emulates the behavior of the FareController component, it is used by the RentController to make a payment request and verify that the payment succeed after an end rent request.

5.1.4 RentControllerStub

Usages:

- I2T3

Description This stub emulates the behavior of the RentController component, it is used by the CarController to end a rent after receiving an end rent request.

5.1.5 ReservationControllerStub

Usages:

- I3T3

Description This stub emulates the behavior of the ReservationController component, it is used by the RentController to verify the existence of a valid reservation after receiving a start rent request .

5.2 Drivers

Since we are using a bottom-up approach we will need several Divers in order to accomplish our integration tests, here there is a list of drivers with their functionalities:

- **MapServiceDriver**: used in the test **I2T1**, this driver calls the methods of the **MapController** in order to test its integration with the **Model**, mostly are getters methods, to get the information about a certain area or about cars.
- **UserServiceDriver**: used in the test **I2T2**, this driver calls the methods of the **UserController** in order to test its integration with the **Model**, this driver have to emulate a user that try to register or log in to **PowerEnjoy**, so it has to enter some personal data calling the right methods of **UserController**.
- **CarManagerDriver**: used in the test **I2T3**, this driver calls the methods of the **CarController** in order to test its integration with the **Model** . The driver has to emulate the environment of a new rent request, made by the **RentController**
- **CarRequestHandlerDriver**: used in the test **I2T3**, this driver calls the methods of the **CarController** in order to test its integration with the **Model**. The driver has to emulate the environment of a end rent request, emulating the behavior of the **CarRemoteController**.
- **CarStatusManagerDriver**: used in the test **I2T3**, this driver calls the methods of the **CarController** in order to test its integration with the **Model**. The driver has to emulate the environment of a new Reservation request, emulating the behavior of the **ReservationController**.
- **DistributionManagerDriver**: used in the test **I2T4**, this driver calls the methods of the **DistributionOptimizer** in order to test its integration with the **Model**. The driver emulate the environment of a request of a possible car distribution made by the **CarController**
- **ReservationServiceDriver**: used in the tests **I2T5** ,**I3T2** this driver calls the methods of the **ReservationController** in order to test its integration with the **Model** and with the **CarController**. It emulates the behavior of a client that tries to do a reservation on a car.
- **ReservationManagerDriver**: used in the test **I2T5** this driver calls the methods of the **ReservationController** in order to test its integration with the **Model**.It emulates the behavior of **RentController** when it receives an new rend request.
- **RentServiceDriver**: used in the tests **I2T6**, **I3T3**, **I4T1** and **I5T1** this driver calls the methods of the **RentController** in order to test its integration with the Model, with the **CarController**, with **ReservationController** and with the **FareController**.It emulates the behavior of a client that tries to rent a car he has previously reserved .
- **RentManagerDriver**: used in the test **I2T6** this driver calls the methods of the **RentController** in order to test its integration with the **Model**.It emulates the behavior **CarController** when it receives an end rend request.
- **FareManagerDriver**: used in the test **I2T7**, this driver calls the methods of the **FareController** in order to test its integration with the **Model**.It emulates the behavior of **RentController** when it receives an end rend request.
- **CarRequestHandlerDriver**: used in the tests **I3T1** and **I3T3**, this driver calls the methods of the **CarController** in order to test its integration with the **DistributionOptimizer** and **RentController**. It emulates the requests made by **CarRemoteController**.

5.3 Test Data

In order to produce some significant tests we will need to populate our database with several fake data for users, requests(past and running),rents(past and running), charging stations, cars, map and all other entities that will populate the database. The generation of such data can be made using some faker library that will allow us to populate the database.

6 Effort spent

During the whole project the team worked with the following schedule:

Claudio Salvatore Arcidiacono

- 9 January : 16.30-20.30 4 hours
- 10 January : 16.30-20.30 4 hours
- 12 January : 17-20 3 hours
- 13 January : 11-13 2 hours
- 13 January : 15-18 3 hours
- 14 January : 17-20 3 hours
- 15 January : 17-20 3 hours

Total hours: 22 hours.

Antonio Di Bello

- 9 January : 4 hours
- 10 January : 4 hours
- 12 January : 2 hours
- 13 January : 2 hours
- 14 January : 6 hours
- 15 January : 3 hours

Total hours: 21 hours.

Denis Dushi

- 9 January : 4 hours
- 10 January : 4 hours
- 12 January : 3 hours
- 13 January : 2 hours
- 14 January : 6 hours
- 15 January : 7 hours

Total hours: 26 hours.

7 References

7.1 Software and tool used

The tools we used to create this document are:

- Overleaf (for latex writing in parallel)