Integration Test Plan Document- PowerEnJoy

POLITECNICO
MILANO 1863

Version 1.0

- Bagna Francesco Matteo (mat. 878556)
- Barletta Carmen (mat. 877129)

Commit date: 15/01/17

Summary

1: Introduction

1.1: Revision history

This document had no revisions.

1.2: Purpose and scope

The purpose of the ITPD (Integration Test Plan Document) document is to describe the plans made for testing the components and the interfaces of the PowerEnJoy system. This document is addressed to the developers of the system.
In this document we will present the main aspects of the organization of the test planning, along with a detailed description of the main test cases, the tools to be used during the tests and the integration of our system's components. The document will be so organized:

- Section 2: The integration strategy for our system: the choice of the strategy with its motivations, the elements that need to be integrated, the sequence of integration and the entry criteria for our system.
- Section 3: The test description for the main methods of our system and the tests of integration between components.
- Section 4: The test equipment (hardware) and tools (software) that need to be used during the testing phase.
- Section 5: All the driver/stubs for our system, plus the test data to be used.
- Section 6: The effort each member of the group put in doing this document.

1.3: Acronyms, abbreviations and definitions

For other acronyms, abbreviations and definitions please refer to the DD and RASD documents.

- ITPD: Integration Test Plan Document
- Driver: A driver is a program that accepts test data and passes it to the components to be tested, then acquires and shows the results. It is used to call methods from components that are not already finished to the component to be tested.
- Subsystem: A high level unit of the system, made by subcomponents. A collection of subsystem forms the complete system.

1.4: References

Documents that were referenced in doing this document are:

- The RASD document for our project
- The DD document for our project
- Sample Integration Test Plan Document.pdf

2: Integration Strategy

2.1: Entry Criteria

Before integration can begin, there are some documents and components must be delivered:

- This document.
- The DD and RASD Documents.
- The components, at least the firsts that need to be integrated (All the third-party components, like the Database, the GEB, the Payment Manager and Driving License Manager, plus the first components of our system, like the Car Controller, the Location Manager, the Safe Area Component, etc. See section 2.4 for the full chronological list of integration).
- The drivers for specific integration tests (again, see sections 2.4 and 3 for the first driver).
- All input data for tests.

Also, Unit Testing must have already been done.

2.2: Elements to be integrated

In this paragraph we will provide a complete list of all the components that must be integrated in our system; in paragraph 2.4.1 we will provide the exact steps of integration.
Our system is built upon many high-level components, which are, for the sake of modularity and order, split in sub-components, even if many of them weren't completely specified for the sake of brevity.
The first subsystem to be considered is Car Controller, that is created by the integration of the Car Manager and the Car Search subcomponents.
We can then individuate the Central System subsystem, composed by the Car Controller subsystem plus the User Manager, Ride Manager, Reservation Manager, Location Manager, Safe Area and Administration Manager subcomponents.
All the other subsystems are made by a collection of components that don't interact with each other, because all of them exclusively interact with the Central System; In this case the integration will be limited to making the subsystems correctly expose the functionalities of its subcomponents.
The Subsystems involved are:

- The Users subsystem, made by the User App, Operator App and Administrator App subcomponents.
- The Database;
- The External Interfaces subsystem, made by the Payment Manager and Driving License Manager subcomponents (both external-party components).
- The Car subsystem, which main subcomponent is the GEB or Car App component.

Moreover, nearly all these subsystems (Database, External Interfaces and Car) are external-party subsystems (or are only made of external-party components), so they should be already correctly integrated.
Some subcomponents (like the Car Manager) directly interact with higher-level subsystems (In this case the Database); these dependencies will be modeled in other paragraphs.
The final integration process will see as protagonists all the individuated subsystems, which will be integrated between themselves in order to create our final system.

2.3: Integration Testing Strategy

We chose to use a bottom-up testing strategy for our system. We will assume that the external components, like the GEB system, the Database and the Payment and Driving License managers, have already been tested and are ready to be integrated with our system. We will then start to examine those components that mostly interact with those external components, while using drivers to simulate the other components' calls.
Given the structure that we have chosen and described in the DD document, we thought that the bottom-up approach would have been the most appropriate to test the components as soon as they are developed and to develop the others at the same time, so that we could work in parallel and optimize the time at our disposal, minimizing the simulation of components not already developed.

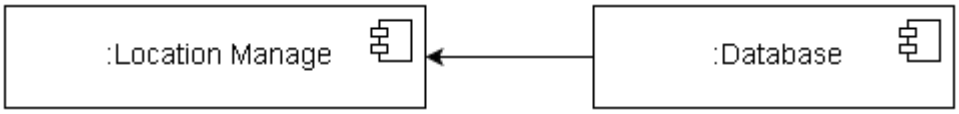
2.4: Sequence of Component/Function integration

In this section it will be described the sequence of component integration in the subsystems, and then the integration between subsystems of our system. An arrow from a component to another means that the second component needs the first to be fully used.

2.4.1: Software Integration Sequence

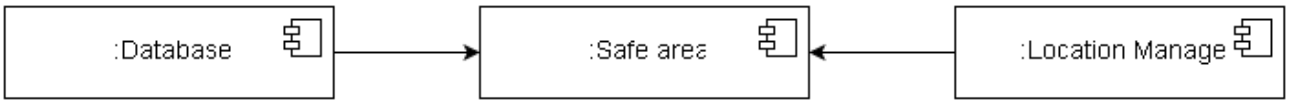
2.4.1.1: Location Manager

The first two components to be integrated are the Location Manager component and the Database; This is our starting point, because the main component of the Central System subsystem, the Car Manager, needs this component to properly be used. Location manager will need a driver of the Safe Area and Car Search components in order to make tests.



2.4.1.2: Safe Area

The second component to be integrated is the Safe Area component, which needs both the Database and the Location Manager to properly function. It will need no drivers.

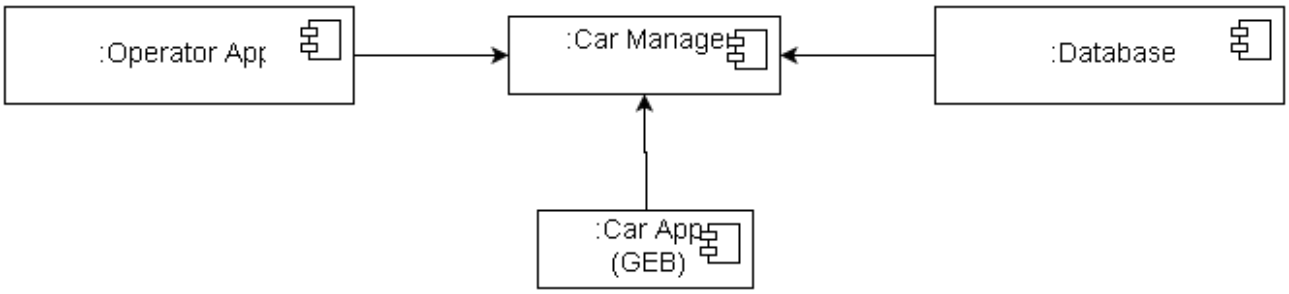


2.4.1.3: Operator App

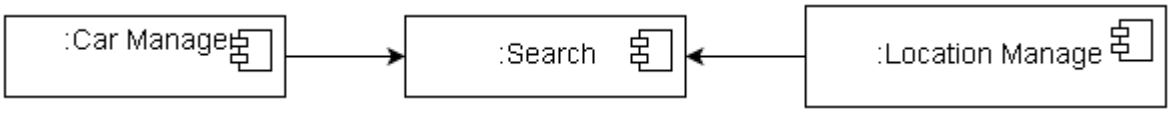
We then take a slight detour, exiting from the Central System in order to build the Operator App component. Why is this? Because our main subsystem, the Car Manager, needs this component in order to send recharging requests. The Operator App is a component that does not need any other component to properly function, but will be tested with a Central System driver (more specifically, a Car Manager driver.)

2.4.1.4: Car Controller.

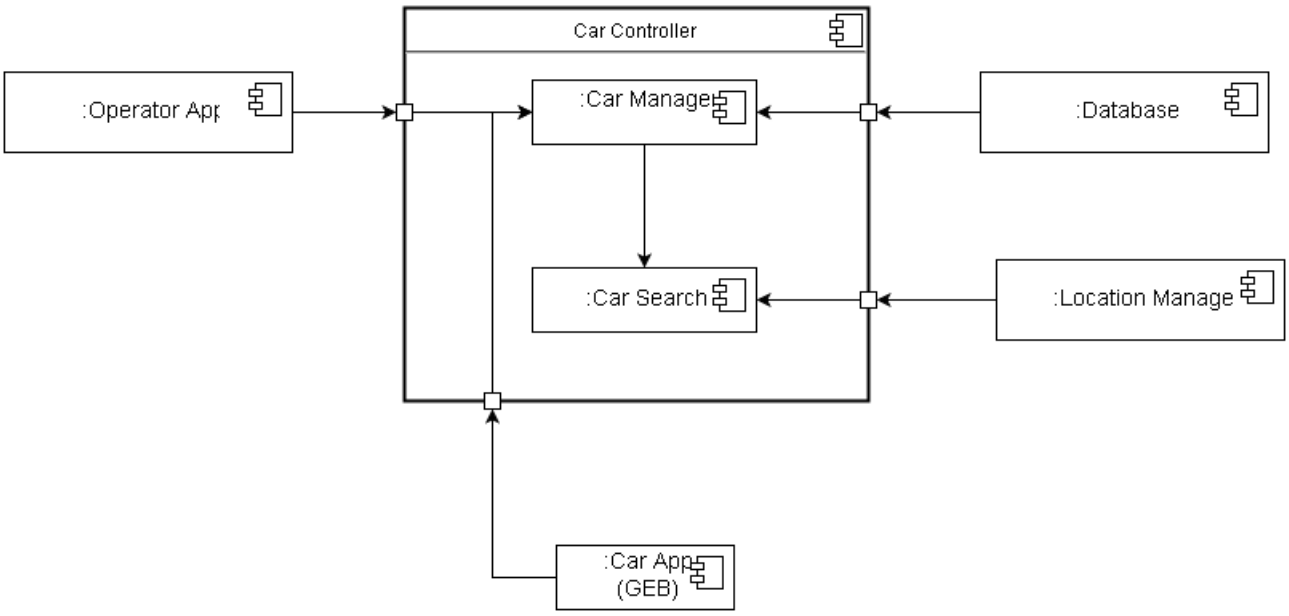
It is now time to integrate the main component of the central system. This will be done in two phases.
First we integrate the Car Manager: it needs the Operator App, the Car App (GEB) and the Database to properly function, while it will be tested by using a driver for Reservation Manager and Car Search.



After having built the Car Manager, we can integrate the Car Search subcomponent: it will need the Car Manager and Location Manager component to be integrated, and will need no drivers.

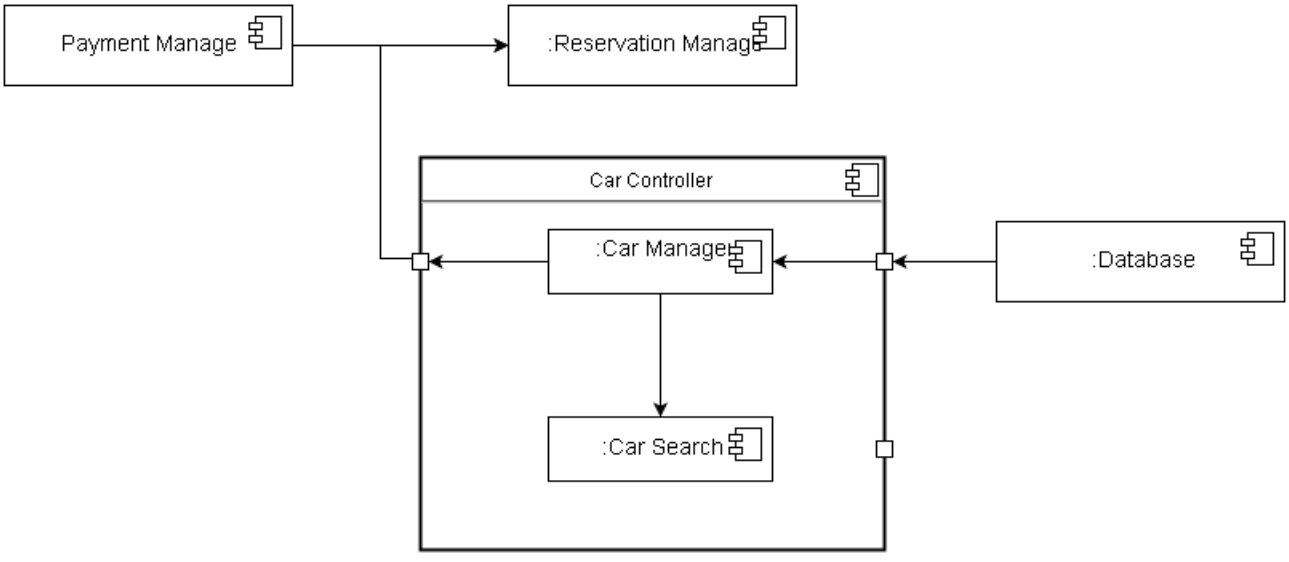


Since we want to have a wider vision of our subsystem, here is a schema for the integration of the whole Car Controller Subsystem.



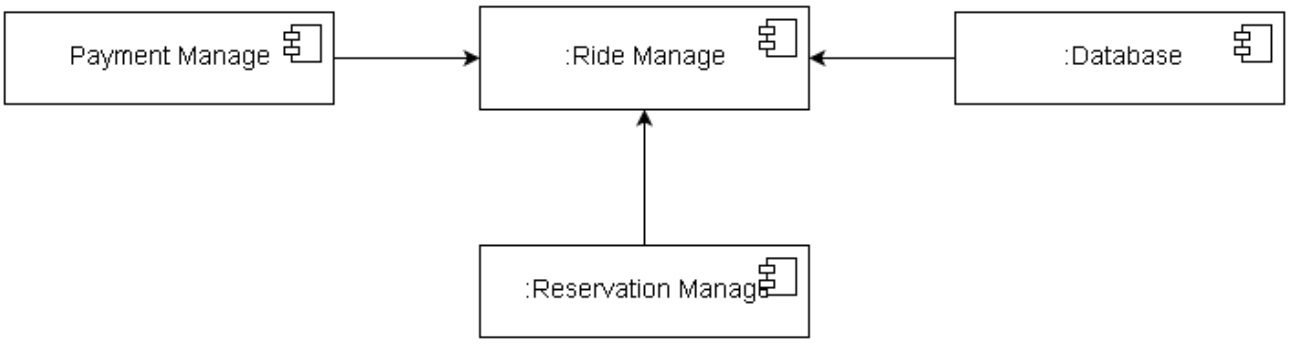
2.4.1.5: Reservation Manager

Now that we've done the main component for our Central System, we can continue with the other components. First there is the Reservation Manager. The reason is simple: it's a component that only needs the Database, the Payment Manager and the Car controller to function, while all the others (except for the Administration Manager) need it to be used.
The Reservation Manager will need a Ride Manager Driver and a User Manager Driver to be tested.



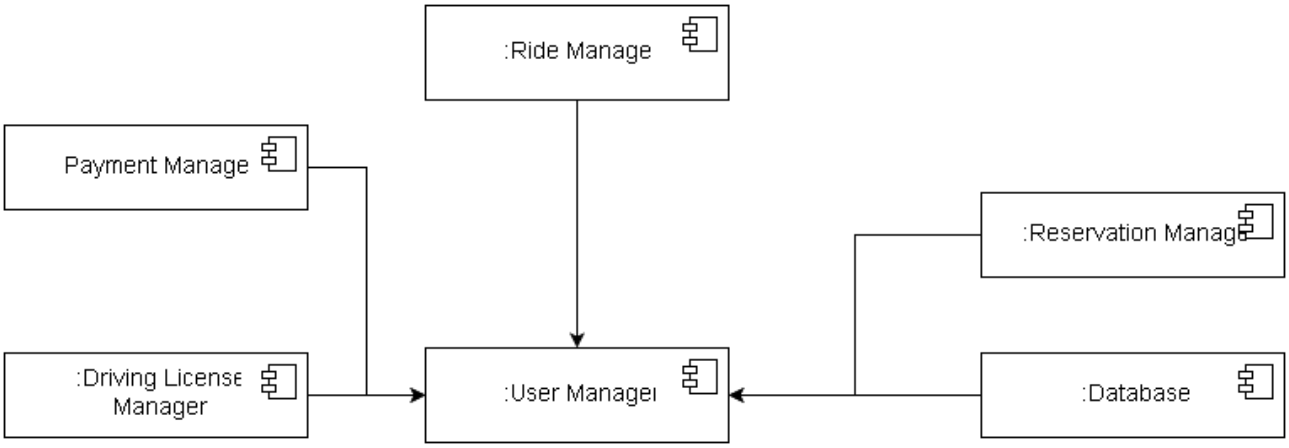
2.4.1.6: Ride Manager

We can now integrate the Ride manager, since it needs only the Reservation Manager, the Payment Manager and the Database to be used. It will need a User Manager Driver.



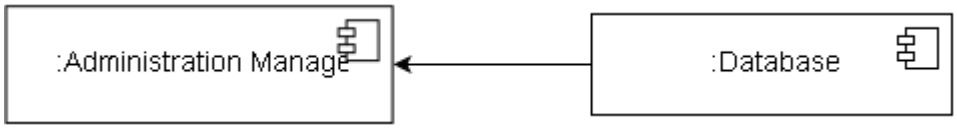
2.4.1.7: User Manager

The next component is the User Manager. It's one of the last component to be done, because it needs a lot of other components to be used: Ride Manager, Reservation Manager, Database, Payment Manager and Driving License Manager. It will also need a User App Driver.



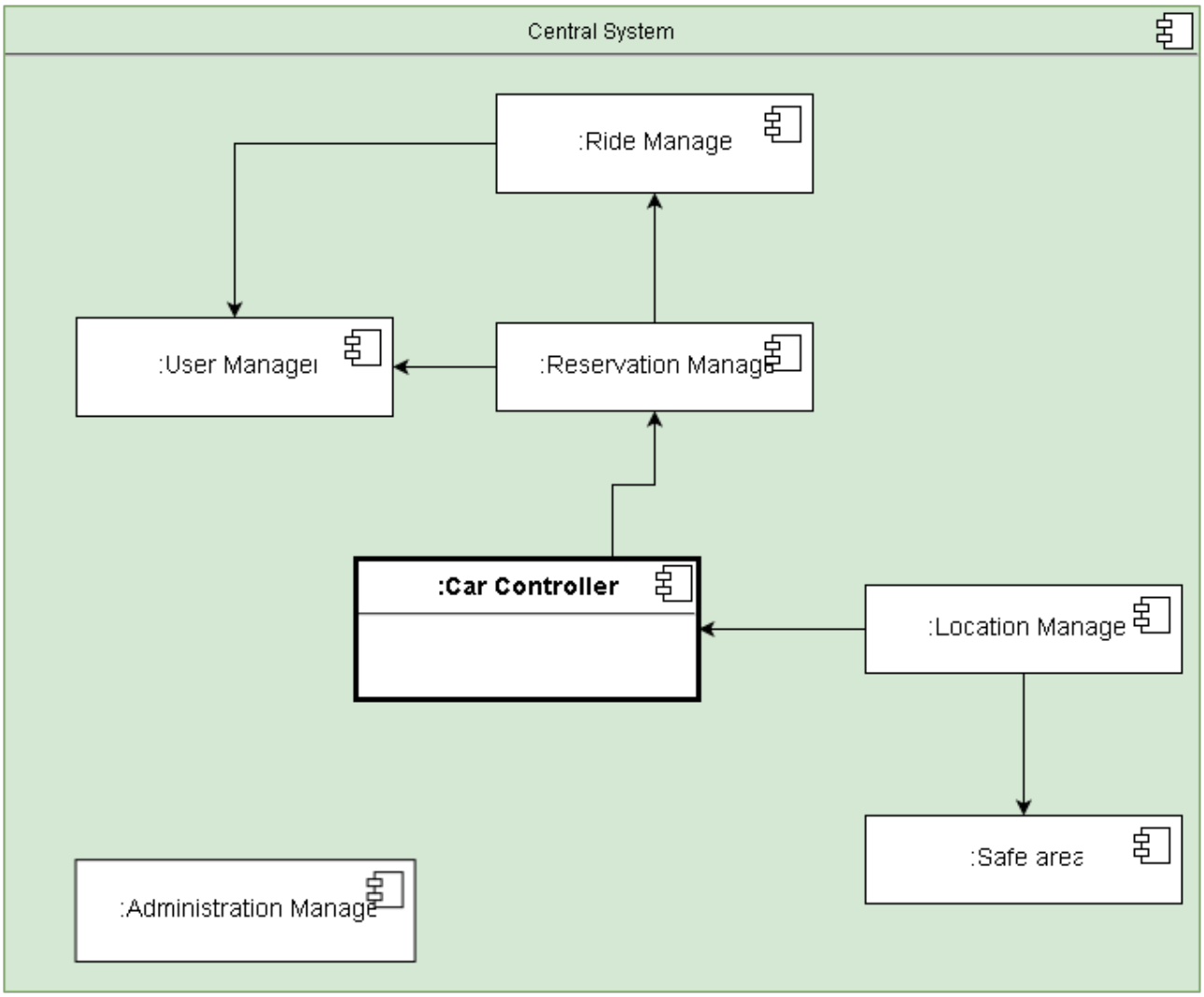
2.4.1.8: Administration Manager

This could have been done even at the beginning of these operations, but we decided to do it after all the others were done. This component will need the Database and a driver for the Administrator App.



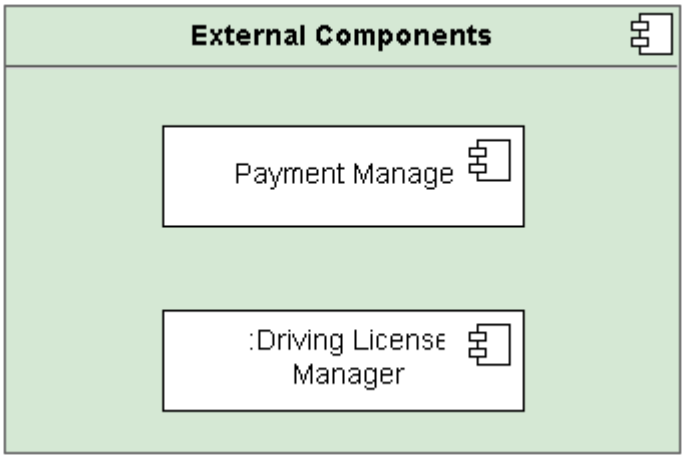
2.4.1.9: Central System

Here is a complete view of all the Central system and its internal dependencies. Please note that the dependencies with components that do not belong to the Central System (Database, Operator App, Car App, Payment Manager, Driver License Manager) will not be modeled here.



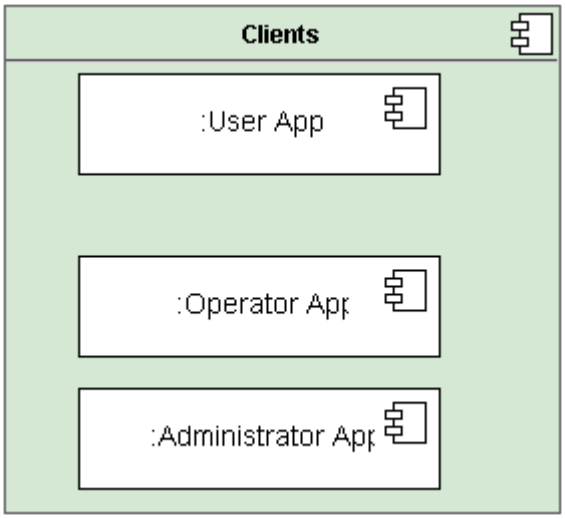
2.4.10: External Components

Here is the External Components subsystem, which subcomponents were already built and only needed to be correctly “integrated” by letting the subsystem correctly show their functionalities.



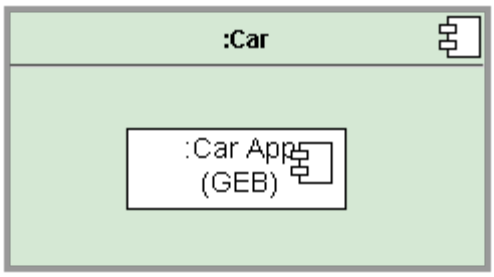
2.4.11: Clients

There is the Clients subsystem, made by the User App, Operator App and Administrator App.



2.4.12: Car

Lastly there is the Car subsystem, which only subcomponent is the Car App(GEB) component.

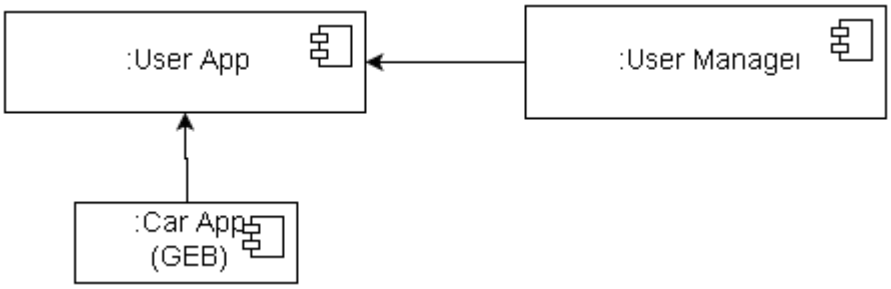


2.4.2: Subsystem Integration Sequence

In this section we will see the integration between components that are of different subsystems (we will not see again the already formulated dependencies) and a general sight of all the subsystems together.

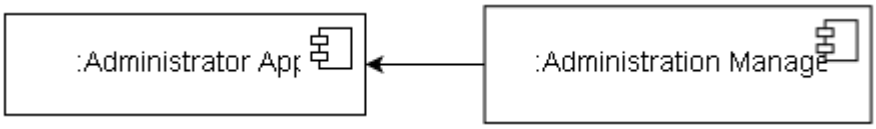
2.4.2.1: User App

User app needs the User Manager to properly function, and needs no Drivers.



2.4.2.2: Administrator App

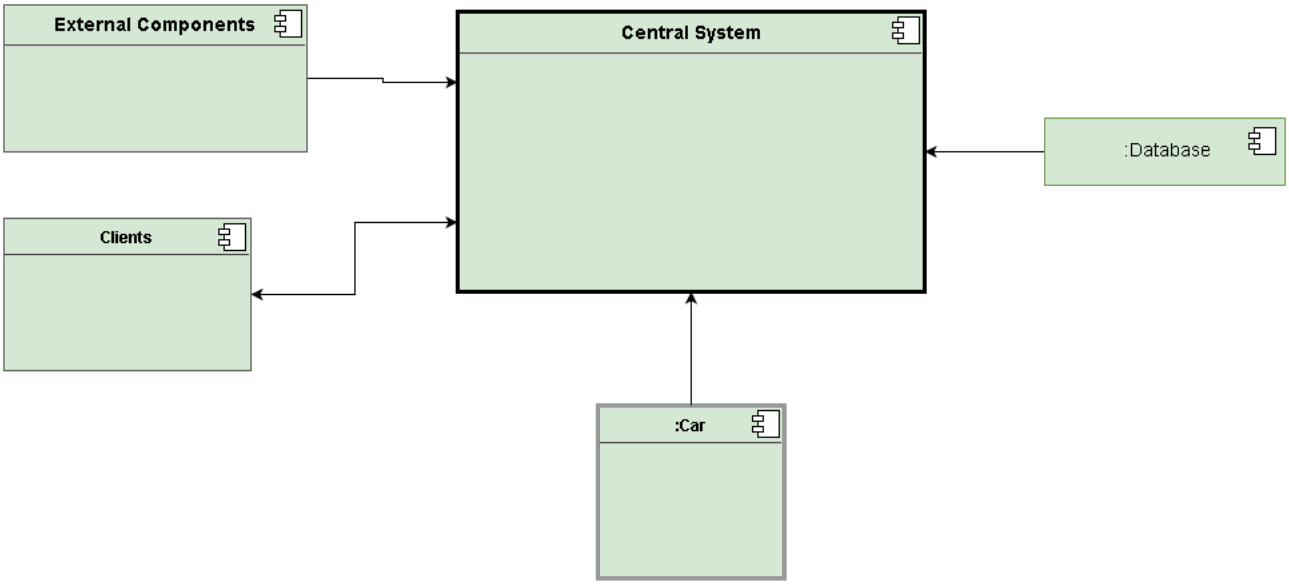
Administrator App only needs Administration Manager to be used, and no Drivers.



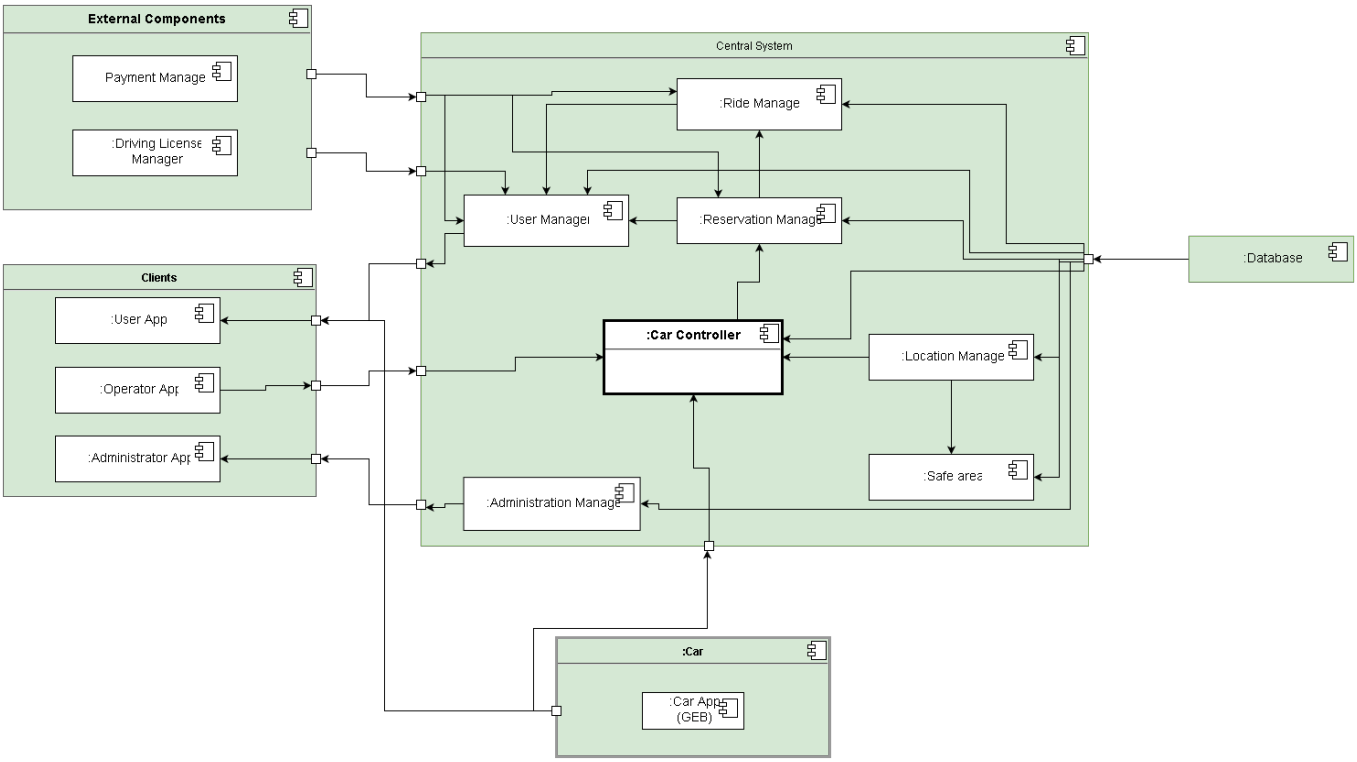
2.4.2.3: General views

Here are the general views for our complete system.

The first is generic and shows only the interactions between subsystems:



The second is a specific view of all the interactions between components (All these interactions have already been discussed before):


External Components
Payment Manage
:Driving License Manager
Central System
:Ride Manage
:User Manager
:Reservation Manage
:Database
Clients
:User App
:Operator Apr
:Administrator Apr
:Car Controller
:Location Manage
:Safe area
:Administration Manage
:Car
:Car App (GEB)

3: Individual Steps and Test Description

In this section we will describe the kind of tests that there will be executed on the integrated components, briefly listing the input and expected output and then focusing on the most important functions that should be tested during the integration test process.

3.1: Test Case Specification

3.1.1: Integration test case T1

Test Case Identifier	T1
Test Items	Location Manager → Database Interface
Input specification	Create typical request of location searching
Output specification	Check if the correct queries are executed on the Database
Environmental Needs	Car Controller Search Driver

3.1.2: Integration test case T2

Test Case Identifier	T2
Test Items	Safe Area Manager → Location Manager
Input specification	Create typical request of safe area searching
Output specification	Check if locations are found and correct methods in the Location Manager are called
Environmental Needs	T1 passed successfully

3.1.3: Integration test case T3

Test Case Identifier	T3
Test Items	Operator client application
Input specification	Create typical request of maintenance
Output specification	Check if operator can answer by calling his methods correctly
Environmental Needs	Car Controller Manager Driver

3.1.4: Integration test case T4: Car Controller Subsystem

3.1.4.1: Integration test case T4.1

Test Case Identifier	T4.1
Test Items	Car Controller Manager → Operator client application
Input specification	Create typical request of maintenance
Output specification	Check if methods concerning the operator calling are called properly
Environmental Needs	T3 passed successfully

3.1.4.2: Integration test case T4.2

Test Case Identifier	T4.2
Test Items	Car Controller Manager → Database Interface
Input specification	Create typical input to ask info about a certain car
Output specification	Check if the correct info can be retrieved by the Database
Environmental Needs	Reservation Manager Driver

3.1.4.3: Integration test case T4.3

Test Case Identifier	T4.3
Test Items	Car Controller Manager → Car Interface (GEB)
Input specification	Create typical communication between Car Controller of Central System and GEB
Output specification	Check if the communication works and the correct methods of the car interface are called
Environmental Needs	T4.2 passed successfully

3.1.4.4: Integration test case T4.4

Test Case Identifier	T4.4
Test Items	Car Search Manager → Location Manager
Input specification	Create typical request of car searching
Output specification	Check if the Location Manager is able to answer through its methods
Environmental Needs	T1 passed successfully, Reservation Manager Driver

3.1.5: Integration test case T8

Test Case Identifier	T8
Test Items	Ride Manager→ Reservation Manager
Input specification	Create typical inputs to start a ride
Output specification	Check if the correct methods are called in the Reservation Manager
Environmental Needs	T4.4 passed successfully

3.1.6: Integration test case T9

Test Case Identifier	T9
Test Items	User Manager→ Reservation Manager
Input specification	Create typical inputs to make a reservation
Output specification	Check if the correct methods are called in the Reservation Manager

Environmental Needs	T8 passed successfully

3.1.7: Integration test case T10

Test Case Identifier	T10
Test Items	Administration Manager → Database Interface
Input specification	Create typical input to update and query the Database
Output specification	Check if the correct changes are applied and functions of the Database are called
Environmental Needs	N/A

3.1.8: Integration test case T11

Test Case Identifier	T11
Test Items	User client application→ User Manager
Input specification	Create typical inputs of the User
Output specification	Check if the correct methods are called in the User Manager
Environmental Needs	T9 passed successfully

3.1.9: Integration test case T12

Test Case Identifier	T12
Test Items	User client application→ Car Interface (GEB)
Input specification	Create typical communication between the user and the car
Output specification	Check if the Car is able to answer to user requests
Environmental Needs	T4.3 passed successfully

3.1.10: Integration test case T13

Test Case Identifier	T13
Test Items	Administrator client application→ Administrator Manager
Input specification	Create typical administrator input
Output specification	Check if the methods of the Administrator Manager are called properly
Environmental Needs	T10 passed successfully

3.1.11: Integration test case T14

Test Case Identifier	T14
Test Items	Ride Manager→ Payment Manager
Input specification	Create typical request of payment

Output specification	Check if the payment transaction can be made correctly
Environmental Needs	N/A

3.1.12: Integration test case T15

Test Case Identifier	T15
Test Items	User Manager→ Payment Manager
Input specification	Create typical request to check payment info
Output specification	Check if the payment info are valid
Environmental Needs	T11 passed successfully

3.1.13: Integration test case T16

Test Case Identifier	T16
Test Items	User Manager→ Driving License Manager
Input specification	Create typical request to check driving license info
Output specification	Check if the driving license info are valid
Environmental Needs	T11 passed successfully

3.2: Test Description

3.2.1: Location Manager, Database Interface (T1)

`setGPS(car,position)`

Input	*Output*
A null parameter for the position	Null Argument Exception raised
An invalid position	InvalidPositionException raised
A null parameter for the car id	Null Argument Exception raised
A valid car id and a valid position	The tuple containing the position of the corresponding car in the database is updated

3.2.2: Safe Area Manager,Location Manager (T2)

`getSafeAreaPosition(safeArea)`

Input	*Output*
A null parameter	Null Argument Exception raised
An invalid safe area id	InvalidArgumentException raised
A valid safe area id	The corresponding position of the safe area is returned

3.2.3: Operator client application (T3)

`answerToRequest(answer)`

Input	*Output*
A null parameter for the answer	Null Argument Exception raised
An invalid answer	InvalidArgumentException raised
A negative valid answer (False)	NegativeAnswerException raised
A positive valid answer (True)	The operator accepts the maintenance request

3.2.4: Car Controller Subsystem

3.2.4.1: Car Controller Manager, Operator Client Application (T4.1)

`sendRequestOfMaintenance(car)`

Input	***Output***
A null parameter	NullArgumentException raised
A car whose position is unavailable	InvalidPositionException raised
A valid id of a car	A valid request of maintenance is sent to the operator

3.2.4.2: Car Controller Manager, Database Interface (T4.2)

`setState(car,state)`

Input	*Output*
A null parameter	NullArgumentException raised
An invalid state	InvalidArgumentException raised
A null parameter for the car id	Null Argument Exception raised
A valid car id and a valid state	The tuple containing the state of the corresponding car in the database is updated

`getState(car)`

Input	*Output*
A null parameter	Null Argument Exception raised
An inexistent car id	NullArgumentException raised
A formally valid id of a car	Returns the stored car state

`getGPS(car)`

Input	*Output*
A null parameter	Null Argument Exception raised
An inexistent car id	NullArgumentException raised
A formally valid id of a car	Returns the stored car position

3.2.4.2: Car Controller Manager, Car Interface(GEB) (T4.3)

`getState(car)`

Input	*Output*
A null parameter	Null Argument Exception raised
The wrong car id	InvalidCarIdException raised
The formally valid id of the car	Returns the actual car state

`getPassengers(car)`

Input	*Output*
A null parameter	Null Argument Exception raised
The wrong car id	InvalidCarIdException raised
The formally valid id of the car	Returns the actual number of passengers

`setAccessibility(car,"OutOfService")`

Input	*Output*
A null parameter	Null Argument Exception raised
The wrong car id	InvalidCarIdException raised
The formally valid id of a car	The car's state is set to Out of Service

3.2.4.3: Car Controller Search Manager, Location Manager(T4.4)

`getGPS(car)`

Input	*Output*
A null parameter	Null Argument Exception raised
An inexistent car id	NullArgumentException raised
A formally valid id of a car	Returns the actual car position

`near(position,car)`

Input	*Output*
A null parameter for the position	Null Argument Exception raised
An invalid position	InvalidPositionException raised
A null parameter for the car id	Null Argument Exception raised
A position far from the car	False
A position near (<=1km) the car	True

`inSafeArea(carPosition)`

Input	*Output*
A null parameter for the position	Null Argument Exception raised
An invalid position	InvalidPositionException raised
A position corresponding to a Safe Area	True
A position not corresponding to a Safe Area	False

`inSpecialSafeArea(carPosition)`

Input	*Output*
A null parameter for the position	Null Argument Exception raised
An invalid position	InvalidPositionException raised
A position corresponding to a Special Safe Area	True
A position not corresponding to a Special Safe Area	False

`farFromSpecialSafeArea(carPosition)`

Input	*Output*
A null parameter for the position	Null Argument Exception raised
An invalid position	InvalidPositionException raised
A position further than 3km to the nearest Special Safe Area	True
A position closer than 3km to the nearest Special Safe Area	False

`findReservableCars(position)`

Input	*Output*
A null parameter for the position	Null Argument Exception raised
An invalid position	InvalidPositionException raised
A valid position	Available and charging cars “near” the position are searched and returned

3.2.5: Ride Manger, Reservation Manager (T8)

`startRide(reservation)`

Input	*Output*
A null parameter for reservation	Null Argument Exception raised
An expired reservation	InvalidReservationException raised
A valid reservation	Car is set to “Reserved”, doors are set to “Closed” and engine is set to “on”, while the reservation timer is interrupted and the charging starts to be calculated

3.2.6: User Manager, Reservation Manager (T9)

`makeReservation(position,user)`

Input	*Output*
A null parameter for position	Null Argument Exception raised
A null parameter for user	Null Argument Exception raised
A valid user id with a jet existing valid reservation active	OneReservationPossibleException raised
A valid user id, but blocked (he has a pending payment)	BlockedUserException raised
An invalid position	InvalidPositionException raised
A position with no car available nearby	NoCarAvailableException raised

A valid user id and a valid positon	Reservable cars are searched and the user is asked to choose

`cancelReservation(user)`

Input	*Output*
A null parameter for user	Null Argument Exception raised
A valid user id with no active reservations	NoReservationException raised
A valid user id, but blocked (he has a pending payment)	BlockedUserException raised
A valid user id with an active reservation (timer <60min)	The user's reservation is delated

`userHasReservationActive(user)`

Input	*Output*
A null parameter for user	Null Argument Exception raised
A valid user id with no active reservations	False
A valid user id, but blocked (he has a pending payment)	BlockedUserException raised
A valid user id with an active reservation (timer <60min)	True

3.2.7: Administration Manager, Database Interface (T10)

`insertCar(car)`

Input	*Output*
A null parameter	NullArgumentException raised
A request with a car id already existent in the database	InvalidArgumentException raised
A valid id of a new car	A new row containing the car's information is inserted into the database

`delateCar(car)`

Input	*Output*
A null parameter	Null Argument Exception raised
An inexistent car id	NullArgumentException raised
A formally valid id of a car	The correspondent row containing car's info is delated

`setState(car,state)`

Input	*Output*
A null parameter	NullArgumentException raised
An invalid state	InvalidArgumentException raised
A null parameter for the car id	Null Argument Exception raised

A valid car id and a valid state	The tuple containing the state of the corresponding car in the database is updated

`getState(car)`

Input	*Output*
A null parameter	NullArgumentException raised
An inexistent car id	InvalidArgumentException raised
A valid car id	The tuple containing the state of the corresponding car in the database is returned

`insertSafeArea(position,safearea)`

Input	*Output*
A null parameter for the position	NullArgumentException raised
A null parameter for the safe area	NullArgumentException raised
An invalid position	InvalidPositionException raised
A request with a safe area already existent in the database	InvalidArgumentException raised
A valid id of a new safe area and a valid position	A new row containing the position of the safe area is inserted into the database

`insertSpecialSafeArea(position,specialSafeArea)`

Input	*Output*
A null parameter for the position	NullArgumentException raised
A null parameter for the safe area	NullArgumentException raised
An invalid position	InvalidPositionException raised
A request with a special safe area already existent in the database	InvalidArgumentException raised
A valid id of a new special safe area and a valid position	A new row containing the position of the special safe area is inserted into the database

`insertOperator(operator)`

Input	*Output*
A null parameter	NullArgumentException raised
A request with an already existent operator in the database	InvalidArgumentException raised

A valid id of a new operator	A new row containing the new operator's info is inserted

`getOperator(operator)`

Input	*Output*
A null parameter	NullArgumentException raised
An inexistent operator id	InvalidArgumentException raised
A valid operator id	The tuple containing the info of the corresponding operator in the database is returned

`delateOperator(operator)`

Input	*Output*
A null parameter	Null Argument Exception raised
An inexistent operator id	NullArgumentException raised
A formally valid operator id	The correspondent row containing operator's info is delated

`getUser(user)`

Input	*Output*
A null parameter	NullArgumentException raised
An inexistent user id	InvalidArgumentException raised
A valid user id	The tuple containing the info of the corresponding user in the database is returned

3.2.8: User Client Application, User Manager(T11)

`checkCredentials(user,password)`

Input	*Output*
A null parameter	NullArgumentException raised
An inexistent user id	InvalidArgumentException raised
An empty password	InvalidArgumentException raised
A valid user and password combination, which how- ever is not the correct one	InvalidCredentialException raised
A correct and valid user and password combination	Returns a session token

`getPassword(user)`

Input	*Output*
A null parameter //	NullArgumentException raised
An inexistent user id	InvalidArgumentException raised

A correct and valid user	Sends an email to the user to complete the registration

`register(userInfo)`

Input	*Output*
A null parameter	NullArgumentException raised
An already existent user	InvalidArgumentException raised
Some formally correct user info	Interact with the database to create a new user entry and a new password

`makeReservation(position,user)`

Input	*Output*
A null parameter for position	Null Argument Exception raised
A null parameter for user	Null Argument Exception raised
A valid user id with a jet existing valid reservation active	OneReservationPossibleException raised
A valid user id, but blocked (he has a pending payment)	BlockedUserException raised
An invalid position	InvalidPositionException raised
A position with no car available nearby	NoCarAvailableException raised
A valid user id and a valid positon	Reservable cars are searched and the user is asked to choose

`cancelReservation(user)`

Input	*Output*
A null parameter for user	Null Argument Exception raised
A valid user id with no active reservations	NoReservationException raised
A valid user id, but blocked (he has a pending payment)	BlockedUserException raised
A valid user id with an active reservation (timer <60min)	The user's reservation is delated

3.2.9: User Client Application, Car Interface (GEB)(T12)

`openCar(user)`

Input	*Output*
A null parameter	NullArgumentException raised
A user without a reservation for that car	NoReservationxception raised
A valid user	Car is opened

3.2.10: Administrator Client Application, Administrator Manager (T13)

`checkCredentials(administrator,password)`

Input	*Output*
A null parameter	NullArgumentException raised
An inexistent administrator id	InvalidArgumentException raised
An empty password	InvalidArgumentException raised
A valid administrator and password combination, which how- ever is not the correct one	InvalidCredentialException raised
A correct and valid administrator and password combination	Returns a session token

3.2.11: Ride Manager, Payment Manager (T14)

`getPayment(user,amount)`

Input	*Output*
A null parameter user	NullArgumentException raised
An inexistent user id	InvalidArgumentException raised
A valid user id	Request of payment is sent, and the user is blocked if he doesn't have money to pay

3.2.12: User Manager, Payment Manager (T15)

`getPaymentInfo(user)`

Input	*Output*
A null parameter user	NullArgumentException raised
An inexistent user id	InvalidArgumentException raised
A valid user id	Returns user's payment info

3.2.13: User Manager, Driving License Manager (T16)

`getDrivingLicenseInfo(user)`

Input	*Output*
A null parameter user	NullArgumentException raised
An inexistent user id	InvalidArgumentException raised
A valid user id	Returns user's driving license info

4: Tools and Test Equipment Required

4.1: Tools

In order to test the components of our system we will use various automatic tools, and some other techniques in order to obtain the widest and deepest possible set of tests for our system.
The main one will be RSpec, a testing software explicitly developed for Ruby on Rails Framework, which we have used for the main server of our system. Being an already compatible software, it's the best choice for testing the components of our system, at least in the server environment.
We will then use the JUnit framework, useful for testing the interactions between components and unit testing. Junit will be used to test methods, responses and exceptions raised in order to be sure that every one of them is correctly formed.

We have also decided to use manual testing, being it a good method to find faults and errors in the code, before passing the code to an automatic tester.
In particular we will do sessions of inspections and walkthroughs, where our code will be scanned for errors in order to be fixed later. This in accordance with all the main rules of walkthroughs and inspections.

4.2: Equipment

All the testing activities will be done in the necessary testing environment, using the necessary equipment. This will include smartphones and computers, as indicated below:

- At least one Android smartphone with the supported version of Android installed, an NFC or Bluetooth connection, for display size from 3" to 6".
- At least one Android tablet with the supported version of Android installed, an NFC or Bluetooth connection, for display size from 7" to 12".
- At least one computer, not dependent from display size or OS, since the application will be entirely web based.

The tests on the computers will only regard the registration and login functionalities, since it's been decided that these will be the only functions available for laptop and computer's users.
The tests on smartphones and tablets will test all the main functionalities of the system, both on the web and on the application designed for Android.
The backend testing will be done in a cloud environment similar to the one that will be used on the final release of the project, with the same Java Enterprise Application Server and the same Database.

5: Program Stubs and Test Data Required

5.1: Program Stubs and Drivers

Since we have decided to use a bottom-up approach in component integration and testing, we will need a few drivers to perform the necessary method invocation on the components to be tested. These drivers have already been declared during the integration test in section 2.4, but will be here amplified.
Here is a list of drivers and their functions:

- Safe Area Driver: This will be used to invoke methods in Location Manager, like retrieving the coordinates of the car to see if the user is in a safe area.
- Car Search Driver: This driver will invoke methods on Location Manager, like retrieving the coordinates of cars in a certain area or for the GPS system. It will also invoke methods on the Car Manager subcomponent.
- Car Manager Driver: Used to test the Operator App component, it will send recharging request to the Operator App.
- Reservation Manager Driver: It will invoke methods, like the retrieval of the information on the current ride, from the Car Manager.
- Ride Manager Driver: It will be used while testing the Reservation Manager component, like calling (for example) for info on the ride.
- User Manager Driver: This driver will invoke methods on the Reservation Manager when trying to make a new reservation and on Ride Manager when retrieving the information on the last ride of the user.
- User App Driver: it will call for User Manager in order to login, register, make new reservation and all the other operations related to the user.
- Administrator App Driver: This driver will invoke methods on the Administration Manager component.

5.2: Test Data

To perform the integration tests that we specified before, we need different sets of data that allow to excite the system in its most crucial parts.

- A set of correct and incorrect values for the car identifier, in order to perform:
  - ✓ T1: Location Manager→ Database Interface
  - ✓ T4: Car Controller subsystem with all its interactions
  - ✓ T10: Administration Manager→ Database Interface

  This set has to contain at least the following type of data:

  - Null object

  - Null fields
  - Car id not legally valid
- A set of correct and incorrect values for the user identifier, in order to perform:
  - ✓ T9: User Manager→ Reservation Manager
  - ✓ T10: Administration Manager→ Database Interface
  - ✓ T11: User Client Application→ User Manager
  - ✓ T12: User Client Application→ Car Interface (GEB)
  - ✓ T15: User Manger→ Payment Manager
  - ✓ T16: User Manger→ Driving License Manager
  - ✓ T14: Ride Manager→ Payment Manager

  This set has to contain at least the following type of data:

  - Null object
  - Null fields
  - Invalid Username/email address
- A set of correct and incorrect values for the operator identifier, in order to perform:
  - ✓ T10: Administration Manager→ Database Interface

  This set has to contain at least the following type of data:

  - Null object
  - Null fields
  - Invalid Username/email address

- A set of correct and incorrect values for the position, in order to perform:
  - ✓ T1: Location Manager→ Database interface
  - ✓ T4: Car Controller subsystem with all its interactions
  - ✓ T9: User Manager→ Reservation Manager
  - ✓ T10: Administration Manager→ Database Interface
  - ✓ T11: User Client Application→ User Manager

  This set has to contain at least the following type of data:

  - Null object
  - Null fields
  - Location outside the PowerEnJoy service
  - Incorrect coordinates
- A set of correct and incorrect values for the [special] safe area position, in order to perform:
  - ✓ T2: Safe Area Manger → Location Manager
  - ✓ T10: Administration Manager→ Database Interface

This set has to contain at least the following type of data:

- Null object
- Null fields
- Incorrect coordinates

- A set of correct and incorrect values for the administrator identifier, in order to perform:
  - ✓ T13: Administrator Client Application → Application Manager

  This set has to contain at least the following type of data:

  - Null object
  - Null fields
  - Invalid Username/email address
- A set of correct and incorrect values for the reservation, in order to perform:
  - ✓ T8: Ride Manager→ Reservation Manager

  This set has to contain at least the following type of data:

  - Null object
  - Null fields
  - Reservation with an expired timer
  - Reservation for the same user at the same time

- A set of correct and incorrect values for the car's state, in order to perform:
  - ✓ T4: Car Controller subsystem with all its interactions
  - ✓ T10: Administration Manager→ Database Interface

  This set has to contain at least the following type of data:

  - Null object
  - Null fields
  - Incompatible state (e.g. engine="on" and action="still", or accessibility="out of service" and engine="on" etc.)

6: Effort spent

6.1: Hours of work

6.1.1: Barletta Carmen

09/01/17 → 3h

10/01/17 → 2h

11/01/17 → 3h

12/01/17 → 1h30m

6.1.2: Bagna Francesco Matteo

9/01/17 → 2h

10/01/17 → 3h

11/01/17 → 1h

12/01/17 → 2h