POLITECNICO DI MILANO
Computer Science and Engineering
Project of Software Engineering 2

Integration Test Plan Document

Authors: Falci Angelo 875123
Lanzuise Valentina 807364
Lazzaretti Simone 875326

Reference Professor: Di Nitto Elisabetta

SUMMARY

1 Introduction

In this document we are going to realize an Integration Testing, which will show that all the components of our software interacts with each others and that their functionalities actually work. This integration testing includes interaction between all layers of the system, as a complete end-to-end of the functionality.

1.1 Revision History

Version 1.0

1.2 Purpose and Scope

This document represents the Integration Testing Plan Document for PowerEnJoy. Integration testing is fundamental to ensure that every component that is in our system works well as all by itself but also with other components.
For this reason we write here in details every aspect of the integration test:

- A list of all components and their sub-components in the system that will have to be tested
- Description of the criteria that must be respected during the tests
- The approach that must be used and the logic behind it
- In which order the components and sub-components will be integrated and testing
- Expected output data for all integration test in front of determinate inputs
- The minimum performance demanded by the system requirements, listed in previous documents (specially in RASD)
- List of tools that will have to be used during the testing, together with a description of the operational environment in which the test will be executed

1.3 List of Definitions and Abbreviation

Here we write the main terms we will use in the project with their meaning.

- GUEST: identify the person not registered yet.
- REGISTERED USER: identify the person registered who can use the service.
- USER: identify the generic person who is using the service.
- SAFE AREA: identify the area where the user can leave a car to have a
- discount.
- POWER GRID: identify the power station that a safe area can have where the
- user can recharge the car to obtain a discount.
- SYSTEM: identify server and database that manage the web-application

service, and the software that manages the car.

- CAR: identify every single car provided by PowerEnJoy.

- POSITION: indicate the specific position about car, safe area and power grid using latitude and longitude.
- ASSISTANCE: identify service who user can call if it has a problem with the car.
- ASSISTANT: identify both the operator that manages the maintenance of the cars and the telephone operator who helps clients.
- HOMEPAGE: indicate the page in which are addressed either guests and registered users before signing up.
- USER HOME: identify the page in which users are addressed after log in where they have access to the services.
- PERSONAL USER DEVICE: identify a generic device used by user with an Internet connection.
- MSO: identify the “Money Saving Option” function.
- VoIP: is a acronym to identify Voice over Internet Protocol is a methodology and group of technologies for the delivery of voice communications over Internet Protocol (IP) networks, such as the Internet.
- Work Station: identify the computers in the office of PowerEnJoy with which the assistants can communicate with the server.
- Model: identify the abstraction of the data of the DBMS.
- SA: “Safe Area” abbreviation.
- PG: “Power Grid” abbreviation.
- MVC: “Model View Controller” abbreviation.

1.4 List of Reference Documents
- The project description
- The RASD
- The DD
- Examples of project of the previous year

2 Integration Strategy

2.1 Entry Criteria
Before proceeding with the integration test, we have to introduce some entry criteria dictated by the progress of the process, in order to produce meaningful results.
First of all "Requirement Analysis and Specification Document" and "Design Document" must be written completely. This is necessary because we must know the all components that our system has and how they are connected each other.
Then, the components that we want to test must be at least partially completed. For this reason we can start integrating test when at least the following percentage for

each components are completed:

- 100 % for the Model: database and all classes that interact with it must been completed because this is the most critic part and all other components depend from them
- 90% for User System: this is the components that connect our main stakeholders to the system (and all components that are included in the system). We integrate this before Car System because if nobody is connected to the system we can't test efficiently the Car System.
- 70% for Car System: this component include all the service that user can use through the car display. We let this component for last in order to test the system like in real case, in fact the user will interact first with user system (using the browser application) and the it will interact with car system (using the car display).
- 50% for User Client and Car Client: this components include the graphics that user will use to have access to the service.

The percentage written above shows the situation that we must reach before starting the integrating test. However when we integrate and test a single component, its percentage will have to be at lest 90% or we will waste time repeating the test again when the component is at an higher percentage of completeness.

2.2 Element to be Integrated

In the following paragraph we are going to provide a list of all the components that need to be integrated together.
As described in the Design Document, the system is composed by five sub-systems: *Model, User System, Car System, User Client* and *Assistant Client*. We can view DBMS and *Model* like a unique subsystem because the DBMS doesn't offer functionalities but it just contain the data, while *Model* offers different functionalities to interact with the DBMS.
At first we are going to integrate those components that strongly depend one another and that provide the main functionalities of PowerEnJoy. We obviously will start from the *Model* because all other components need to interact with the data in DBMS in order to work.
Then we will integrate and test *User System* and *Car System*, these two components are included in the server and satisfy the requests that arrive from the *User Client* or *Car Client*.
We have to integrate and test different sub-components in order to test the entire *User System*: registration request, access request, modification info request, address request, booking request and renting request.
We have to do a similar work for the *Car System:* personal code request, safe area request, path request, MSO request, current charge request and parking request.
At the end we will integrate and test *User Client* and *Car Client.* They include the graphics that user will use to do the requests (like renting or booking) and they simply will sent the request to the server.

2.3 Integration Testing Strategy

The strategy used for testing will be a mix of the bottom-up strategy and the critical module first.
We are going to use a bottom-up strategy for the sub-systems we declared before, and in particular for the server side: this choice of the bottom-up takes consideration of the necessity to have in first place a full working server. We prefer bottom-up rather than top-down one, because we think that the last one is less functional, slow and dispersive, according to the fact that in order to test the whole system it must be already completed; by the other side, the choice of the bottom-up system have some negative aspect, like the fact that we could test every single component.
The first subsystem to implement is the *Model* one, because all other components in the system have to use this component to interact with the DBMS, as we are going to explain in the following paragraph.
Then, in order to have complete and functional integration, we thought to use the critical module first, in particular testing the different *User System* and *Car* System. This strategy allows us to concentrate our testing efforts on the riskiest components first.
According to the fact that *User System* and *Car System* interact each other but the last one depends on the implementation of the first and so also the fact that a malfunctioning of the first influence a lot the operating of the last, we choose to integrate *User System* and *Car System* in the respective order.

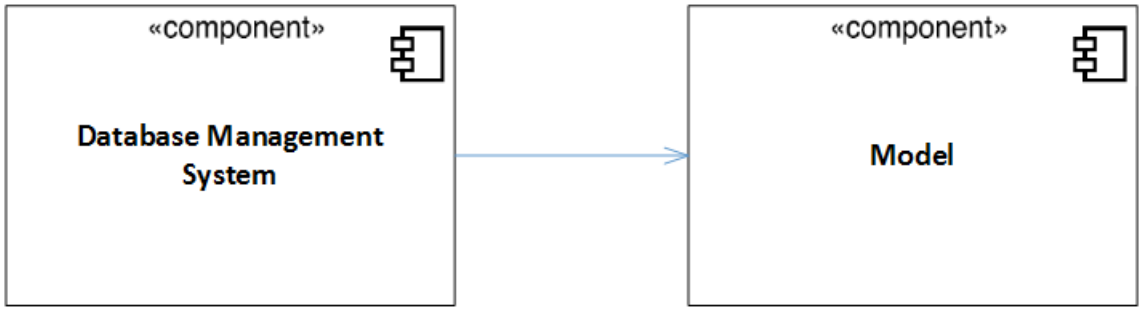
2.4 Sequence of Component/Function Integration

Software Integration Sequence

Above we already said that we will use the bottom-up approach, so in this chapter we describe in which order the sub-components are integrated in order to build the entire system.

Model

The first two components we integrate are “Database Management System” and “Model”. These must be the first components because all other components depend from them to modify the data of the system.
By “Model” we mean all classes that manage the queries on the DB; these classes will be used from the higher components in order to modify the data on the DB.



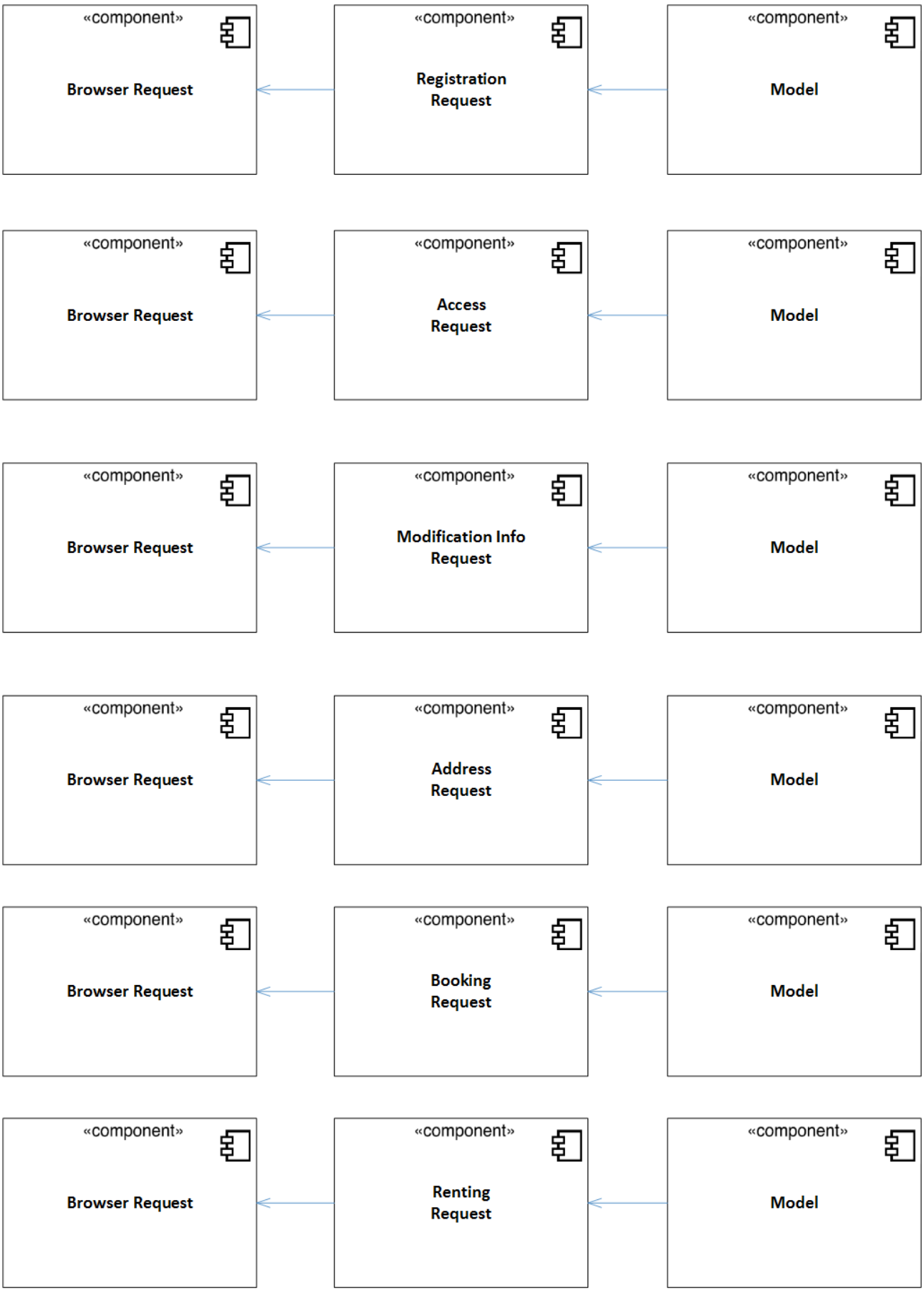
User System

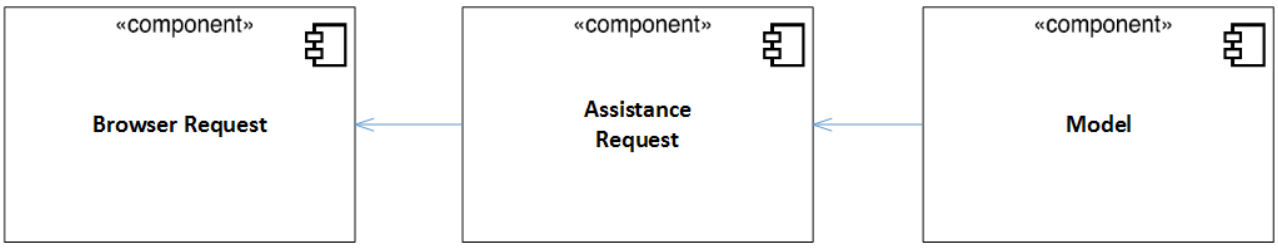
Then we integrate all components that satisfy the requests that user will do in the browser.

We integrate this before the Car System because if user wants to interact with the Car System he/she needs a personal code, that user can achieve only after the registration, which is managed by "Registration Request", a component that belongs to User System. So Car System depends in part from User System.

Now, user system includes different components, in the picture below we show the best order to integrate them:

1) Personal Information Management: this component manages the personal information of the users. It includes four sub-components: one for the registration, one for logging in, one for modifying the personal information after the registration and the last one is for the personal code. This last sub-component is very simple to integrate and test (it just verifies that the personal code inserted is right) so we can add it later, during the Car System integration (also because we can test better this sub-component in the Car System).
   The sub-components are inserted in the order shown in the picture because registration is independent, log in depends from registration (if nobody is registered, nobody can log in) and "Modify information manager" depends from registration and log in (if user wants to modify its personal information he/she has to sing up and log in)
2) Map Management: this component handles all the requests dealing to the map. To includes five sub-component: for address research, for safe area research, for the path research and for the MSO option.
   In this step we just integrate the address research because it is independent from the others, and moreover all other sub-components are used in the Car System so we can add them in the next step.
3) User-Car Management: this component manages the booking and renting functionalities. It includes four sub-component: for renting, for booking, for parking function and for viewing charge.
   Here we include the first two components, booking and renting that are more critic than other two.
   We integrate booking sub-component before, but booking and renting depend each other so it is better integrate them together before testing (system for booking a car checks if the car isn't rented or booked, in the same way the system for renting a car checks if the car isn't rented or booked).
4) Assistance Management: this is the last sub-component we integrate for user system. It is the less critical component in user system because it just manages the assistance request that user does. In details it puts in contact an user with a free assistant when user makes a request of assistance.


«component»
Browser Request
«component»
Registration Request
«component»
Model
«component»
Browser Request
«component»
Access Request
«component»
Model
«component»
Browser Request
«component»
Modification Info Request
«component»
Model
«component»
Browser Request
«component»
Address Request
«component»
Model
«component»
Browser Request
«component»
Booking Request
«component»
Model
«component»
Browser Request
«component»
Renting Request
«component»
Model



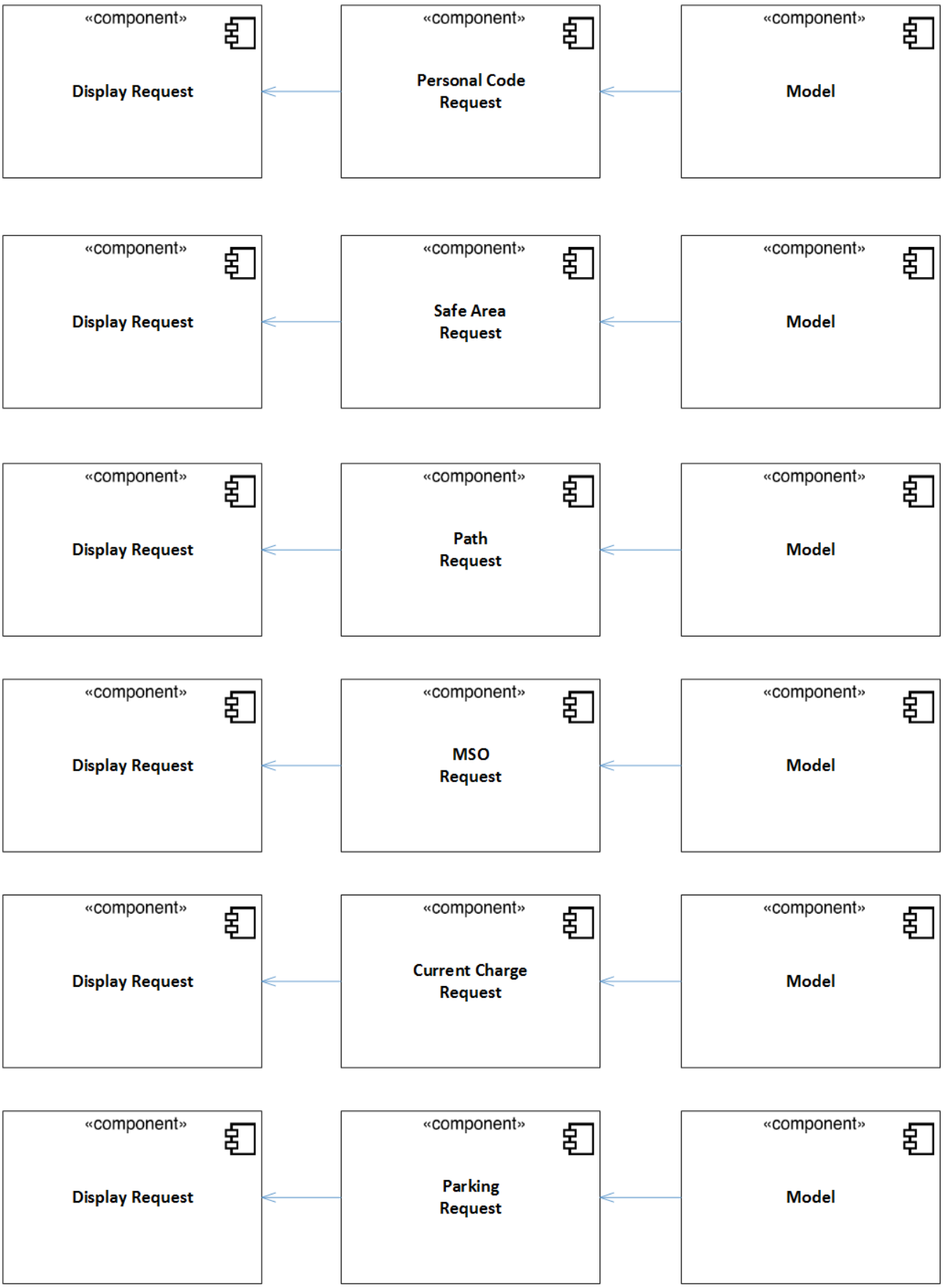
Car System

Now we have to integrate all components that satisfy the requests from the car display.

This component satisfies the requests that user does during the renting while he/she is in the car.

It includes different sub-components and we insert them in this order (shown in the following the picture):

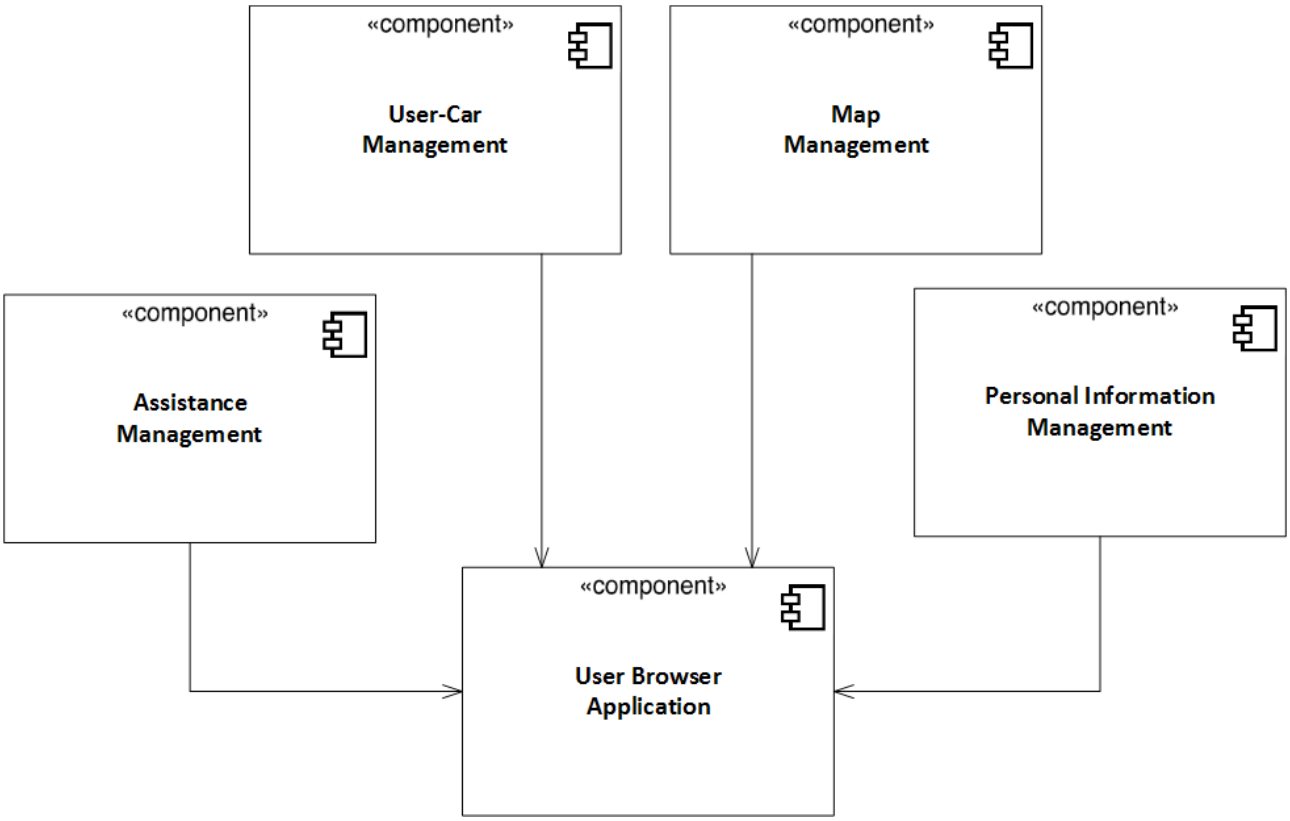
1) Personal Information Management: we start to insert the last sub-component we mentioned above because all components in Car System depend from it (in fact user must insert its right personal code to have access to the Car System).
2) Map Management: for the second we insert the sub-components we didn't insert before (as we explained previously). We respect the order shown in the picture because, in order to build the path, it needs to know the position about safe area (so "Path Request" depends from "Safe Area Request) and in the "MSO request" component we use a combination of the two sub-component mentioned above ("Path Request" and "Safe Area Request").
3) User-Car Management: at the end we integrate and test the component for viewing charge and parking function. Their order is irrelevant because they are independent each other.


«component»
Display Request
«component»
Personal Code Request
«component»
Model
«component»
Display Request
«component»
Safe Area Request
«component»
Model
«component»
Display Request
«component»
Path Request
«component»
Model
«component»
Display Request
«component»
MSO Request
«component»
Model
«component»
Display Request
«component»
Current Charge Request
«component»
Model
«component»
Display Request
«component»
Parking Request
«component»
Model

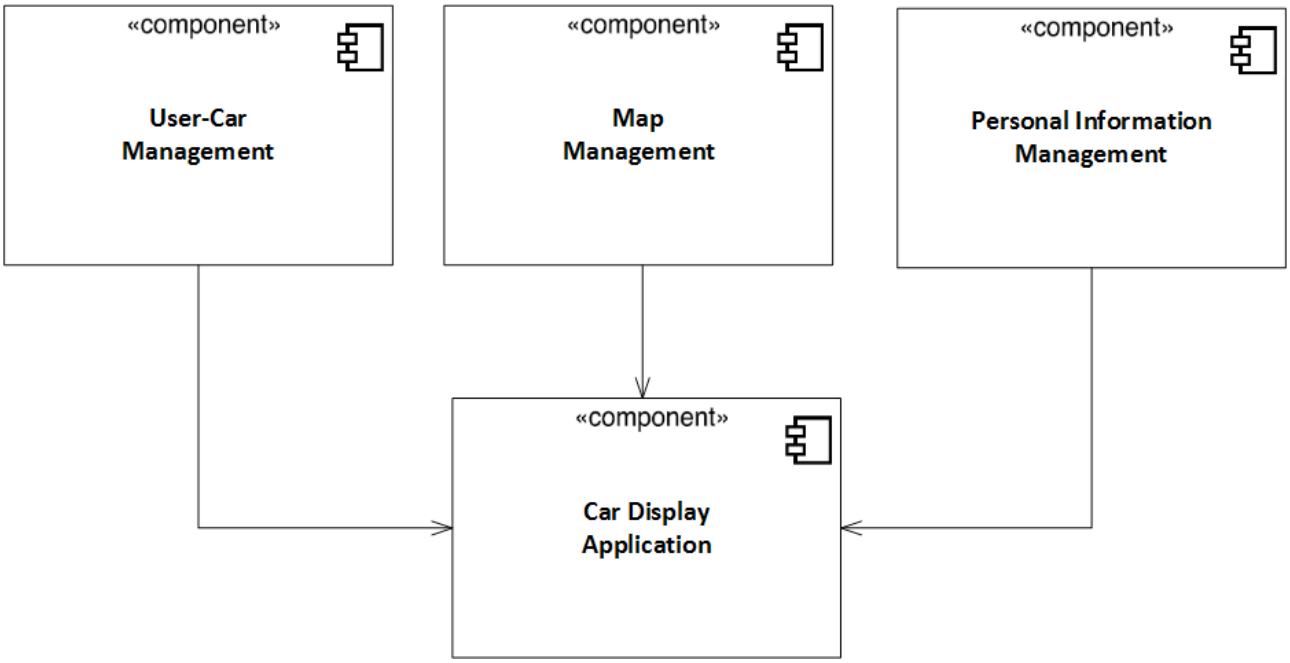
User Client

"User Browser Application" represents the web application that user can use in every device to have access to the services of PowerEnJoy.

This component interacts with User System.



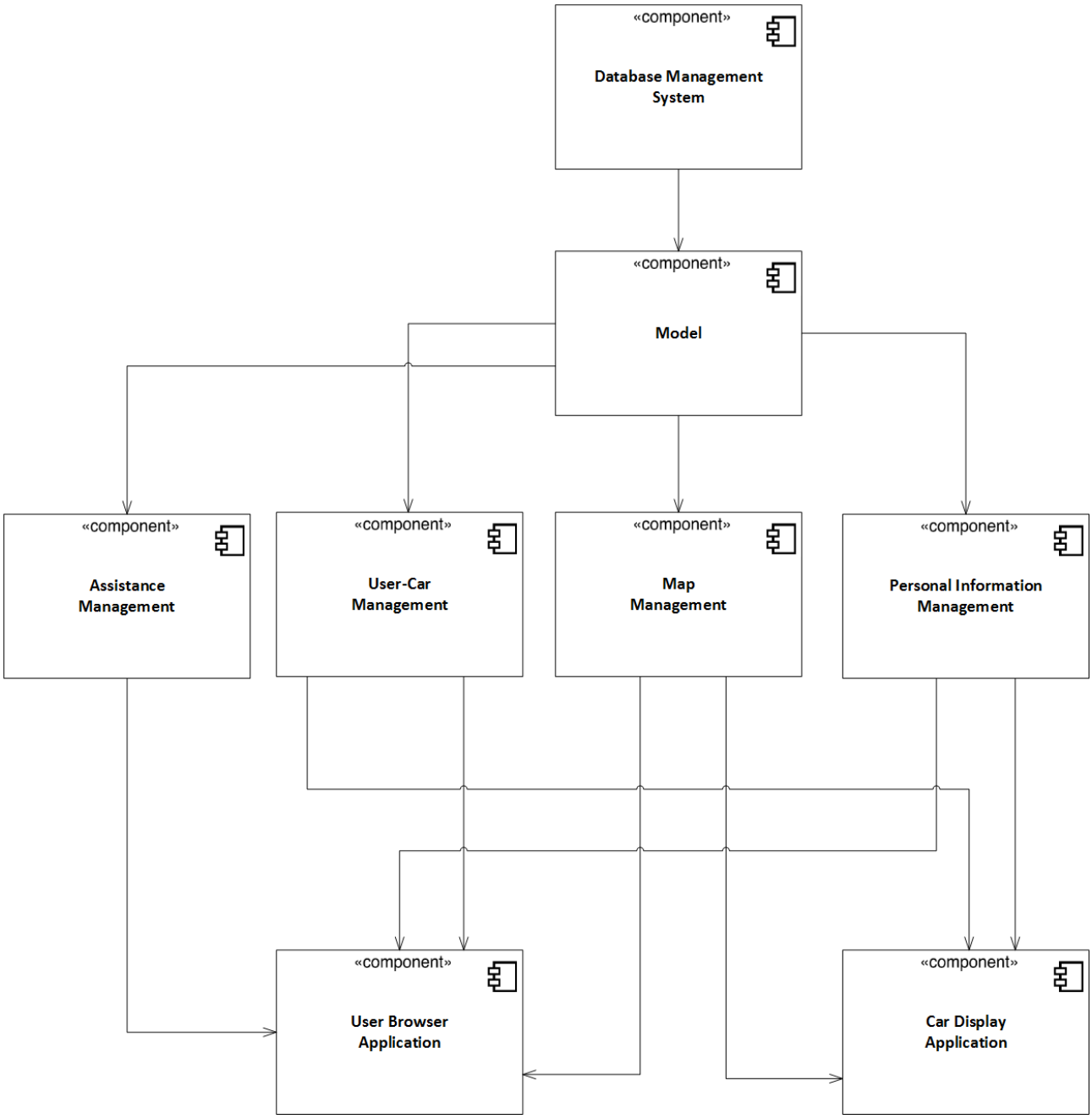
Car Client
"Car Display Application" represents the application that user can use in the car display to have access to the services in the cars of PowerEnJoy.
This component interacts with Car System.



Subsystem Integration Sequence

In the graphics below we can see how all high-level components are connected each other in order to create the entire infrastructure.

We have described each high-level component and which sub-components it includes in the previous chapter, “Software Integration Sequence”.



3 Individual Steps and Test Description

In this chapter we'll provide a detailed description of the tests to be performed on each pair of components that have to be integrated.
The methods tested that are shown below are the most important and critical.
The structure is the following: the title is composed by the name of the sub-system checked and the components that interacts with the following functions. To be precised, the left component in the rounds is the caller, while the right component in the rounds is the called.
For each method we're going to provide a brief description of the input values and the corresponding expected effects on the system. Where it's written "Nothing" under the "Input" column means that no argument has been passed in the function

Personal Information Management(Registration request → Model)

validate_newUserData(request)	
Input	Effect
A null parameter	A NullArgumentException is thrown
A request with an invalid type of data	An InvalidArgumentException is thrown
A request with a driving-license ID, e-mail and/or identify card number already registered in the database	Returns false
Formally valid request	The request is ready to be sent to the database. Returns true
Previous condition	send_to_RR(request) didn't catched any exception

adding_Info(request)	
Input	Effect
A null argument	A NullArgumentException is thrown
A formal valid request	Data are added to the database
Previous condition	validate_newUserData(request) returns true

Personal Information Management(Access Request → Model)

validate_log_in(int password, string username)	
Input	Effect
A null parameter	A NullArgumentException is thrown
A wrong combination of password and username	Returns false
A valid combination of password and username	Returns true
Previous condition	sends_to_Access(request) didn't caught any exception

Personal Information Management (Modification Information Request → Model)

modification_data(data, type)	
Input	Effect
A null data	A NullArgumentException is thrown
The same data that is going to be substituted	An InvalidArgumentException is thrown
An invalid data	An InvalidArgumentException is thrown
A formal valid argument	The already existing data is updated with the new one
Previous condition	sends_to_MIR(request) didn't caught any exception

Personal Information Management (Personal Code Request → Model)

turn_onCar(code)	
Input	Effect
A null code	A NullArgumentException is thrown
An invalid code	An InvalidArgumentException is thrown
A formal valid code	Confirm that the car can turn on
Previous condition	

Map Management (Address Request → Model)

verify_address(address)	
Input	Effect
A null address	A NullArgumentException is thrown
An invalid address	An InvalidArgumentException is thrown
A formal valid address	Returns true
Previous condition	sends_to_Access_R(address) didn't caught any exception

search_coordinates(address)	
Input	Effect
A null address	A NullArgumentException is thrown
An invalid address	An InvalidArgumentException is thrown
A formal valid address	Returning the desired address
Previous condition	verify_address(address) returns true

Map Management (Safe Area Request → Model)

getSA()	
Input	Effect
Nothing	Return the list of safe areas to be published on a car display
Previous condition	sends_to_SAR(request) didn't caught any exception

getPG()	
Input	Effect
Nothing	Return the list of safe areas with power grid station to be published on a car display
Previous condition	sends_to_SAR(request) didn't caught any exception

Map Management (Path Request → Model)

destination_address(address)	
Input	Effect
A null argument	A NullArgumentException is thrown
An invalid address	An InvalidArgumentException is thrown
A formal valid address	Itinerary is shown to the user on the display
Previous condition	sends_to_DR(request) didn't caught any exception

Map Management (MSO Request → Model)

sending_DestinationCoordinates(address)	
Input	Effect
A null argument	A NullArgumentException is thrown
An invalid address	An InvalidArgumentException is thrown
A formal valid address	Send the coordinates for the search of the optimal safe area
Previous condition	sends_to_MSOR(request) didn't caught any exception

searching_Optimal_SA(address)	
Input	Effect
A null argument	A NullArgumentException is thrown
An invalid address	An InvalidArgumentException is thrown
A formal valid address	The optimal safe area is returned
Previous condition	sending_DestinationCoordinates(address) didn't caught any exception

calculating_Itinerary(startingPoint, optSA)	
Input	Effect
One of the argument (or both) is null	A NullArgumentException is thrown
One of the argument (or both) is invalid	An InvalidArgumentException is thrown
Formal and valid arguments	The itinerary from the position of the car to the safe area is calculated
Previous condition	sending_DestinationCoordinates(address) ends successfully

User-Car Management (Booking Request → Model)

booking(request)	
Input	Effect
A null request	A NullArgumentException is thrown
The car is not available for booking	An InvalidArgumentException is thrown
A formal valid request	Confirm notification sent to the user
Previous condition	sends_to_BR(request) didn't caught any exception

delete_booking(request)	
Input	Effect
An invalid request (empty booking list)	An InvalidArgumentException is thrown
A formal valid request	The selected booking is deleted
Previous condition	sends_to_BR(request) didn't caught any exception

checking_availability(car)	
Input	Effect
A null argument	A NullArgumentException is thrown
The field "status" of the argument is set to "locked" or "booked"	Return false
The field "status" of the argument is set to "free"	Return true
Previous condition	booking(request) didn't caught any exception

updateStatus(car)	
Input	Effect
Null	A NullArgumentException is thrown
Formal valid argument and checking_availability(car) returns true	The field "status" is set to "booked"
Formal valid argument and checking_availability(car) returns false	The field "status" is set to "free”
Previous condition	checking_availability(car) returned true

User-Car Management (Renting Request → Model)

verify_existance(carPlate, carCode)	
Input	Effect
A null data in the request	A NullArgumentException is thrown
An invalid data in the request	Return false
A formal valid request	Return true
Previous condition	checking_correctness(carPlate, carCode) didn't caught any exception

verify_availability(carPlate)	
Input	Effect
A null data in the request	A NullArgumentException is thrown
An invalid data in the request	Return false
The car matched with the argument is “locked”	Return false
The car matched with the argument is “free”	Return true
Previous condition	checking_correctness(carPlate, carCode) didn't caught any exception

changing_status(carPlate)	
Input	Effect
A null argument	A NullArgumentException is thrown
A formal valid argument	The car status is switched to "locked" and it's opened
Previous condition	verify_existance(carPlate, carCode) and verify_availability(carPlate) returned true

stopping_charge(rentingID)	
Input	Effect
A null argument	A NullArgumentException is thrown
A formal valid argument	The charge of the rent is stopped
Previous condition	sends_to_RR(request) didn't caught any exception

formulating_bill(rentindID)	
Input	Effect
A null argument	A NullArgumentException is thrown
A formal valid argument	The payment is committed and a notification is sent to the renter
Previous condition	stopping_charge(rentingID) ended successfully

Charge Management (Current Charge Request → Model)

getCharge(rentingID)	
Input	Effect
A null argument	A NullArgumentException is thrown
A formal valid renting id	The charge data is returned from the server
Previous condition	send_to_CCR(request) didn't caught any exception

Assistance Management (Assistance Request → Model)

askingHelp(request)	
Input	Effect
A null argument	A NullArgumentException is thrown
A formal valid request	User's request is sent to all the assistance module in the server
Previous condition	sends_to_Assistance(request) didn't caught any exception

inqueueAssistRequest(request)	
Input	Effect
A null valid request	A NullArgumentException is thrown
A formal and valid request	Request is put in a waiting queue in order to wait to be resolved
Previous condition	askingHelp(request) didn't caught any exception

dequeueAssistRequest(request)	
Input	Effect
Nothing	Request is removed from the waiting queue because an assistant take it in charge
Previous condition	An assistant answered to a pending call

Personal Information Management (Browser Request → Registration Request)

send_request_to_RegR(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	Request is forwarded to "Registration Request" component
Previous condition	Nothing

Personal Information Management (Browser Request → Access Request)

sends_to_Access(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	Request is forwarded to "Access Request" component
Previous condition	Nothing

Personal Information Management (Browser Request → Modification Info Request)

sends_to_MIR(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	Request is forwarded to "Modification Info Request" component
Previous condition	Nothing

Map Management (Browser Request → Address Request)

sends_to_Address(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	Request is forwarded to "Address Request" component
Previous condition	Nothing

User Car Management (Browser Request → Renting Request)

sends_request_to_RentingR(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	Request is forwarded to "Renting Request" component
Previous condition	Nothing

User Car Management (Browser Request → Booking Request)

sends_request_to_BR(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	Request is forwarded to "Booking Request" component
Previous condition	Nothing

Personal Information Management (Display Request → Renting Request)

send_request_to_RentR(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	Request is forwarded to "Renting Request" component
Previous condition	Nothing

Charge Management (Display Request → Current Charge Request)

send_request_to_CCR(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	Request is forwarded to "Current Charge Request" component
Previous condition	Nothing

Map Management (Display Request → SA Request)

sending_to_SAR(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	Request is forwarded to "SA Request" component
Previous condition	Nothing

Map Management (Display Request → MSO Request)

sending_to_MSOR(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	Request is forwarded to "MSO Request" component
Previous condition	Nothing

User Client (User Browser Application → Personal Information Management)

sending_Request(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	The request is sent from the user client to the "Dispatcher Request" sub-component and forwarded to the "Browser Request" sub-component
Previous condition	Nothing

publish_user_home()	
Input	Effect
Nothing	The system shows the user home page to the user
Previous condition	validate(password, username) returned true

publish_error_message()	
Input	Effect
Nothing	The system shows an error message to the user
Previous condition	validate(password, username) returned false

show_registration_complete()	
Input	Effect
Nothing	The system says to the user with a message that the registration has been completed
Previous condition	addingInfo(request) ends successfully

User Client (User Browser Application → Map Management)

sending_Request(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	The request is sent from the user client to the "Dispatcher Request" sub-component and forwarded to the "Browser Request" sub-component
Previous condition	Nothing

center_location(coordinates)	
Input	Effect
A null argument	A NullArgumentException is thrown
A formal valid argument	The system centers the position of the user on the coordinates received and published the result to the user
Previous condition	search_coordinates(coordinates) ends successfully

showing_car_information(carPlate)	
Input	Effect
A null parameter	A NullArgumentException is thrown
CarPlate is not found in the model	An InvalidArgumentException is thrown
The parameter is valid	The system shows on the screen information of the car
Previous condition	asking_data(car) ends successfully

User Client (User Browser Application → User-Car Management)

showing_renting_screen()	
Input	Effect
Nothing	The system loads the screen where user inserts data to rent the car
Previous condition	Nothing

sending_carplate&code(request)	
Input	Effect
A null request	A NullArgumentException is thrown
A valid request	The request is sent from the user client to the "Dispatcher Request" sub-componentand forwarded to the "Browser Request" sub-component
Previous condition	Nothing

Car Client (Car Display Application → User-Car Management)

activePF()	
Input	Effect
Nothing	The engine of the auto activate the "Park Function" mode
Previous condition	User pressed “Park Function” button

checkPF(sensor)	
Input	Effect
A null argument	A NullArgumentException is thrown
The sensor is turned off	Return false
The sensor is turned on	Return true
Previous condition	User exits from car

suspend_engine()	
Input	Effect
Nothing	The engine is shut down, except for the charge service system and sensors
Previous condition	checkPF(sensor) returned false

sending_request(request)	
Input	Effect
A null request	A NullArgumentExcepion is thrown
A valid request	The request is sent from the user client to the "Dispatcher Request" sub-componentand and then forwarded to the "Display Request" sub-component
Previous condition	User performed an action

visualize_charge(charge)	
Input	Effect
A null argument	A NullArgumentException is thrown
A formal and valid argument	The system shows on the display the current charge of the service
Previous condition	searching_user_data(userID) ends successfully

showing_proximity_SA(coordinates, distance)	
Input	Effect
distance is a null argument	A NullArgumentException is thrown
distance is a valid argument and coordinates too	The system shows on the car display safe areas limited to a certain distance
Previous condition	getSA() ends successfully

showing_proximity_SA_wPG(coordinates, distance)	
Input	Effect
Distance is a null parameter	A NullArgumentException is thrown
Distance is a valid argument and coordinates too	The system shows on the car display safe areas limited to a certain distance
Previous condition	getPG() ends successfully

showing_destination_screen()	
Input	Effect
Nothing	The systems loads the screen where user has to insert the destination address
Previous condition	User presses on “MSO” button

showing_itinerary(setCoordinates)	
Input	Effect
A null parameter	A NullArgumentException is thrown
A formal and valid argument	The system shows to the user the itinerary
Previous condition	calculating_itinerary() ends successfully

4 Tools and Test Equipment Required

In order to test the various components of our project, we will use different automated testing tools. As we have written in the Design Document, our components are think in Java and Javascript, while Database is thought in MySQL. For the components that have a business logic running in Java, we are going to utilize these tools:

- JUnit framework: this tool is principally used to unit testing activity, but it is also useful to guarantee the exact execution of the components and that the produced results are correct. In particular, we are going to provide different kind of data to the methods of our main functionalities, in order to verify that wrong data raise an exception and it is managed in the right way, while inputs that are on the “domain border” and other topic data produced the results that we expected.
- Arquillian integration testing framework: this tool enables us to execute integration tests in order to verify that the components interact in the right way, so that the whole project actually works.

Instead, in order to test the components whose logic is executed in Javascript (client

side), which can be difficult to test, we are going to use the following tools:

- Qunit: it permit to execute unit tests, so on the all script functions and object methods, using and verifying some assertions, and answering with true or false, depending if it is guaranteed the right execution of the unit.
- Mockjax : it is used to test in order to test and simulate the keyboard or mouse input of user.

5 Program Stubs and Test Data Required

As we have mentioned in the Integration Testing Strategy section of this document, we are going to adopt a bottom-up approach for the component integration and testing.
Because of this choice, we need a number of drivers to actually perform the necessary method invocations on the components to be tested; this will be mainly accomplished in conjunction with the JUnit framework.
Here follows a list of all the drivers that will be developed as part of the integration testing phase, explained with their specific role:

- *Model Driver* this testing module will invoke the methods in the Model in order to verify that Model component interacts in the right way with the DBMS.
- *Personal Information Management Driver* this testing module will invoke the methods in all sub-components in the Personal Information Management in order to verify that they interact with the Model in the right way for satisfying the user requests.
- *Map Management Driver* this testing module will invoke the methods in all sub-components in the Map Management in order to verify that they interact with the Model in the right way for satisfying the user requests.
- *User-Car Management Driver* this testing module will invoke the methods in all sub-components in the User-Car Management in order to verify that they interact with the Model in the right way for satisfying the user requests.
- *Assistance Management Driver* this testing module will invoke the methods in all sub-components in the Assistance Management in order to verify that they interact with the Model in the right way for satisfying the user requests.
- *User Browser Application Driver* using Mockjax this testing module simulates the interaction between user and browser application in order to verify that all services offered for browser application are available and working.
- *Car Display Application Driver* this testing module simulates the interaction between user and the application in the car display in order to verify that all services offered for car application are available and working.

6 Effort Spent

- Falci Angelo: 16h
- Lanzuise Valentina: 11h
- Lazzaretti Simone: 13h