Integration Test Plan Document

Authors:
Cannas, Castiglioni, Loiacono

SUMMARY

PART 1: INTRODUCTION

1.1 PURPOSE AND SCOPE

This document describes the plans for testing the integration of the Power EnJoy System's components. The purpose of this document is to plan the order and execution of the tests between the components, which were described in the Design Document [see 1.3].
The document focuses on the testing of the Server Application and on the Car Application [see Design Document] in order to create a solid base upon which a thin and usable client will be easy to develop.

1.2 LIST OF DEFINITIONS AND ABBREVIATIONS

- **RASD:** Requirements Analysis and Specifications Document.
- **DD:** Design Document.
- **API:** Application Programming Interface.
- **MVC:** Model View Controller pattern.
- **REST:** Representational State Transfer. An architectural pattern for client-server communications.
- **Green Move System:** a series of software platforms used as interface between the Car's control unit and application. It is divided between the Green E-Box (on the car) and the Green Move Center (on the central server).
- **Stripe API:** an API which enables the communication with payment providers in a standardized way.
- **EUCARIS API:** an API which enables the communication with the European Driver Licence Database, to confirm the authenticity of Users' driver licenses.

Other Definitions are provided in the RASD and in the DD document.

1.3 LIST OF REFERENCE DOCUMENTS

- Assignments AA 2016-2017.pdf
- Slides and Lectures on Software Design By Di Nitto Elisabetta
- Sample Design Deliverable Discussed on Nov. 2.pdf
- Paper on the green move project.pdf
- PowerEnJoy_RASD.pdf

- DesignDocument.pdf
- “Software testing and analysis: process, principles, and techniques”, Pezzè-Young, chapter 10
- http://mockito.github.io/mockito/docs/current/org/mockito/Mockito.html
- http://arquillian.org/
- http://jmeter.apache.org/
- https://github.com/linkedin/dexmaker

PART 2: INTEGRATION STRATEGY

2.1 ENTRY CRITERIA

For the integration testing to begin and have a meaningful purpose inside the software development process, a set of conditions have to be verified.
In first place, the RASD Document and the Design Document must be delivered, in order to give an overview of the overall architecture of the system to be tested.
Subsequently, the integration process should start only when certain conditions of completion of the software components are met: this means that every component should be at least completed for its major part (90%), while for the overall system the following percentage of development must be reached. For the Central Server:

- 100% completion for the Database and Database interfaces;
- 90% completion for the interfaces between Dispatchers and Controllers and related called methods inside these components;
- 70% completion for the interfaces and methods between Controllers and Controllers and Controllers and Helpers;
- 50% completion for the interfaces and methods between Controllers and external notification Helpers.

For the Car Application:

- 90% completion for the CarDataHandler functions and 100% of the PaymentCalculator;
- 80% completion for the NetworkController;
- 70% completion for the GreenE-Box Adapter.

This percentages allow the incremental execution of the integration testing process (as described in section 2.3), thus enabling testers and developers to find relatively early malfunctions inside the components behaviour and so timely repair the code.
Moreover, in order to accommodate drivers' and stubs' development related issues, some components need to be unit tested before the integration testing actually takes place.
All Dispatchers' interface functions provided to the high-level components external to the Central Server (mobileApp, CarApplication, service providers), should be unit

tested. The same reasoning lies behind the unit testing of the Database's interface functions, which should be unit tested before integration.
In this way, all stubs and drivers needed to the unit testing can be reused during the integration testing phase.

2.2 ELEMENTS TO BE INTEGRATED

The items to be integrated consists of the integration code of the module of the so far designed and developed PowerEnjoy software project.
Three high-level component are recognizable: the Central Server, the Car Application, and the MobileApp.
The MobileApp is purposely intended as a thin client software, whose structure is not so complex to devise a specific integration testing plan: its application logic is mainly focused on data presentation and on simple request to the Central Server, thus, an unit testing is more than sufficient for the development of our software.
Therefore, the main integration testing effort will be focused on the integration of components of the Central Server and of the Car Application.
For what concerns the Central Server, this means the integration testing of:

1. CarEvent and ClientEvent dispatchers with all the interconnected helpers and controllers;
2. all the various controllers (Registration, Login, Search, Reservation, EndRide, CarData) with the respective helpers and dispatchers;
3. all the various helpers (MoneySaving, Expiration, PaymentNotification, LicenseControl, Emergency) with their respective controllers.
4. all the various controllers with data management responsibilities with the Database.

For what concerns the Car Application, this means the integration testing of:

1. the CarData Handler with the PaymentCalculator components;
2. the PaymentCalculator with the GreenE-Box Adapter;
3. the CarData Handler with the NetworkController and the GreenE-Box Adapter;
4. the Locker/Unlocker with the NetworkController and the GreenE-Box Adapter.

2.3 INTEGRATION TESTING STRATEGY

Our testing strategy lies on the possibility of developing different testing strategies for the main high-level components of our system. All of them in fact present a clear separation in tiers, that communicate through a set of interfaces: every component then, can be integrated and tested separately, leaving the final integration to the system test.

The MobileApp and the CarApplication due to their simple structure do not require a particularly complex integration testing strategy: a simple incremental approach, as a top-down or a bottom-up, will be sufficient.

The Central Server application instead deserves a separate discussion. Since its structure does not show a clear functional subdivision of components into separate subsystems, we thought that a classical incremental approach in this case would be inappropriate and inefficient.

Thus, we preferred to resort to the use of a feature-driven approach, using a mix of functional, thread and risk-based testing. In fact, through a more detailed analysis of our specifications, three main Independently Testable Features (ITF) emerged:

- an independently testable feature related to general data management (ITF1);
- an independently testable feature related to Cars' events management (ITF2);
- an independently testable feature related to Clients' events management (ITF3).

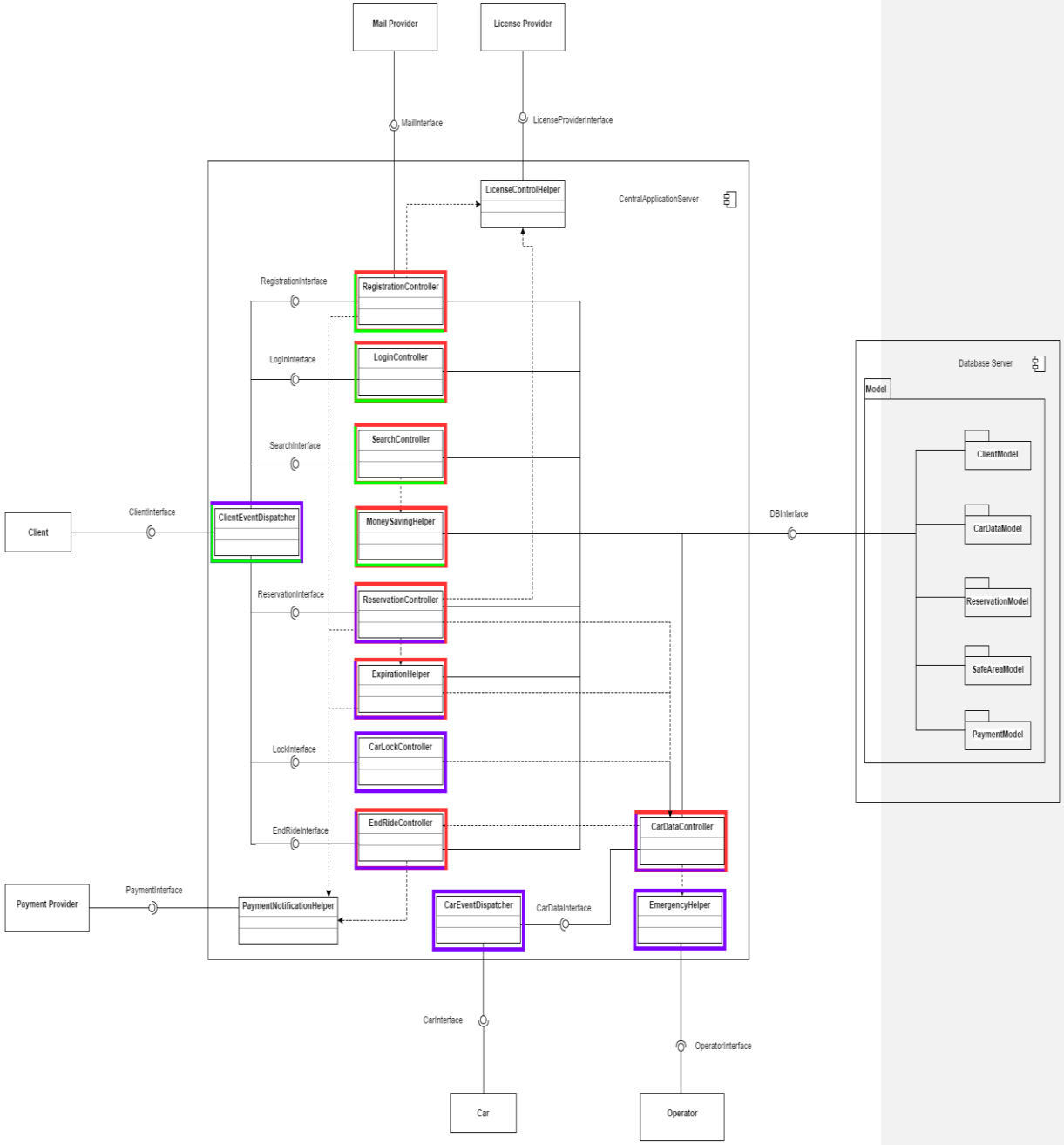

Figure 1: components are highlighted with respect to the functionalities of the ITF they are implementing. Red is for Data management, purple for Cars' events management, green for Clients' event management.

Each of these general ITF comprehends several functionalities.
For example, ITF1 comprises all those functionalities necessary for DB communication and interaction; ITF2 all those functionalities necessary for the correct management and decision taking based on the type of Event related to the Car, which in turn imply the use of functionalities for Car's data analysis, etc...
A generic component of our server, can implement functionalities belonging to one or more of these three features: however, while ITF1 is concerned more about general functionalities that are shared across the ITF2 and 3, these last two in contrast have some utilities which are not used by the other one, and vice versa.
This is due to the fact that they are related to more specific functionalities that can be handled quite independently from the ones of the other ITF, through means of separate software components.
Starting from this reasoning, our Integration Test Strategy gives precedence to the integration of those components which implement functionalities related to the access or interaction with the Database (ITF1); components characterized by the ITF2 and 3 instead, can be integrated and tested quite on their own and, when possible, in parallel at the same time.
Anyhow, to reduce the number of stubs and drivers necessary, attention has been given to the order of testing, primarily using a risk-driven approach.
Due to the central role of the Car Data Controller and Car and Client Event Dispatchers, the integration strategy has been focused in the testing step-by-step of the interactions between the components depending on these three central modules.
Since this strategy implies the testing and integration in some cases of small portions of the actual component implementation, it can be viewed as an actual thread testing strategy at a low-level of abstraction, becoming a risk-strategy inside the testing of the single ITF, and finally as a functional testing strategy at the highest level.
Please note that where possible, we tried to parallelize the integration of components that do not show direct functional dependencies within each other.
This may result in some integration test that has a bottom-up look like fashion. However, all of them are still inserted within our main functional-thread strategy.

2.4 SEQUENCE OF COMPONENT-FUNCTION INTEGRATION

Following the reasoning explained in section 2.3, our integration sequence simply goes after the functional dependency order imposed by the three main ITFs identified for the Central Server.

Unable to identify specific subsystems, our integration strategy will focus first on the integration and testing of functionalities related to the data management (ITF1), then the integration and testing of functionalities related to the Cars' events management (ITF2), and finally the integration and testing of functionalities related to the Clients' events management (ITF3).
Please notice that the integration and testing of ITFs 2 and 3 in some cases are quite independent each from the other.
This means that, despite of the integration of functionalities belonging to the ITF1, which are generally used by all the others components' functionalities and then need to be tested in first place, some of the other utilities can be tested in parallel.
We tried then to highlight this aspect, dividing our test cases not only by the identification of some functionality inside a specific ITF, but even by the possibility of integrating and testing functionally independent (loosely coupled) components when possible.

2.4.1 SOFTWARE INTEGRATION SEQUENCE

Inside the ITF number one, there is not strict integration sequence: due to the fact that we are going to use a commercial DBMS (thus, a component that has already been developed), all components implementing a data management utility can be tested independently from each other and then, in parallel.
However, the integration testing of these functionalities is crucial for the testing of the other ones belonging to ITFs number 2 and 3, thus, they must be integrated and tested first.
Integration tests of this ITF are all under the test cases family I1.

Test Case Identifier	Integration Test Items
I1T1	RegistrationController → Database
I1T2	LoginController → Database
I1T3	SearchController → Database
I1T4	MoneySavingHelper → Database
I1T5	ReservationController → Database
I1T6	ExpirationHelper → Database
I1T7	CarDataController → Database

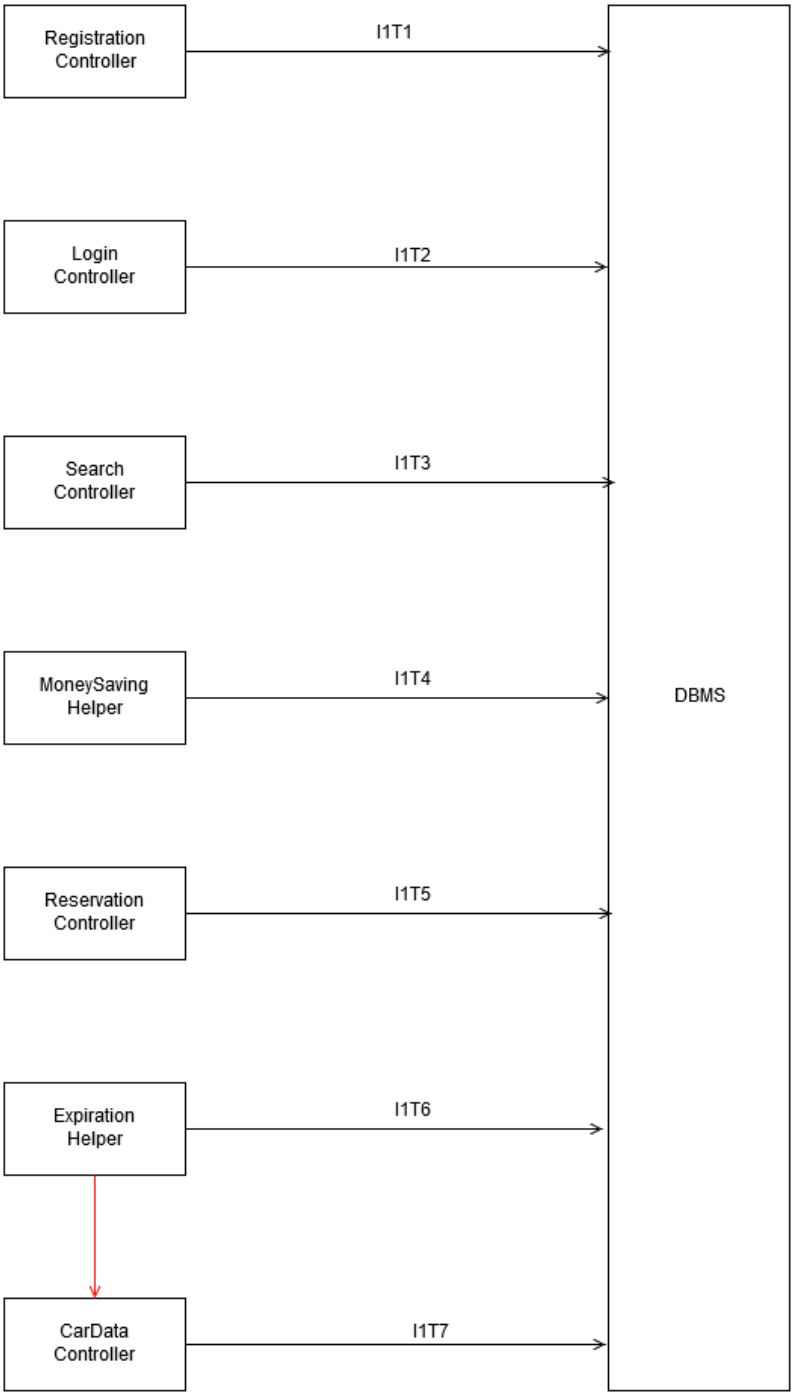

Figure 2: Central Server integration testing sequence for ITF number one. The red arrow indicates the temporal and functional dependency in the execution of test cases.

Please notice that the test I1T7 need the completion of test I1T6 for being performed.

With regards to the ITF2 and 3, the integration sequence can be divided in several steps of independent and parallelizable integration tests. This means that every step comprehends a certain number of independent integration tests, but any subsequent step integration test relies on the completion of a previous step test.
The following table illustrates the integration sequence for the ITF 2 and 3.

Test Case Identifier	**Integration Test Items**
I2T1	ReservationController → ExpirationHelper
I2T2	ClientEventDispatcher → LoginController
I2T3	ClientEventDispatcher → RegistrationController
I2T4	CarEventDispatcher → CarDataController
I3T1	ReservationController → CarDataController
I3T2	ExpirationHelper → CarDataController
I3T3	CarLockController → CarDataController
I3T4	CarDataController → EndRideController
I3T5	EmergencyHelper → CarDataController
I4T1	ReservationController → PaymentNotificationHelper
I4T2	ClientEventDispatcher → EndRideController
I4T3	ClientEventDispatcher → CarLockController
I5T1	ReservationController → LicenseControlHelper
I5T2	RegistrationController → LicenseControlHelper

All tests dependencies are indicated in the single test description in paragraph 3.
Red test case identifiers highlight the testing of functionalities belonging to ITF2, while green identifiers highlight the testing of functionalities belonging to ITF3.

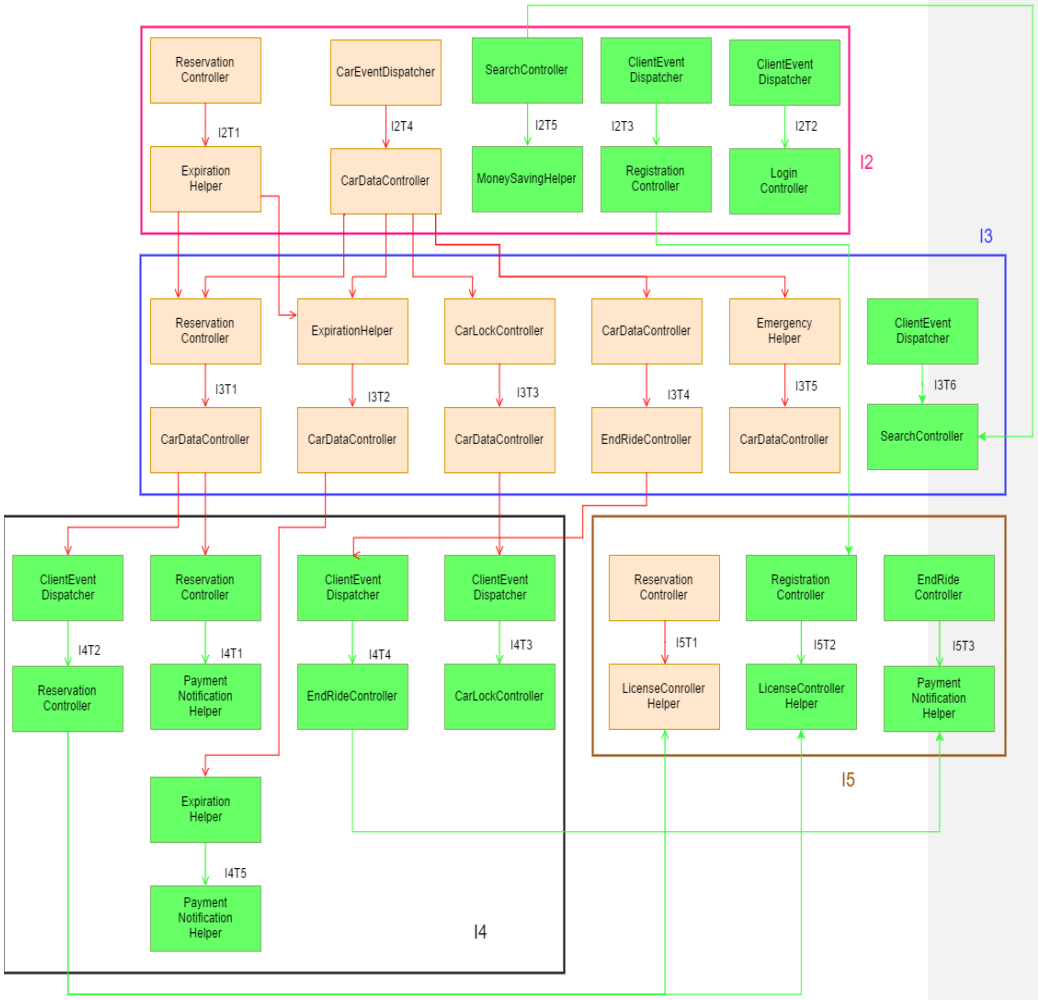

Figure 3: Central Server integration testing order for ITF2 and 3. The actual step in the incremental testing of features is indicated by the number following the letter I. For each step, more than one test case can be executed in parallel.

The numbers in the test case family indicates the actual step in the functional incremental integration testing of the system. Arrows without a test case identifier indicates the temporal and functional dependency between tests and the components involved in it.

Finally, with regards to the CarApplication, the integration sequence of its components follows a simple bottom-up approach. The table below illustrates the order of the integration tests.

Test Case Identifier	Test Case Items
IC1	DataHandler → PaymentCalculator
IC2	Locker/Unlocker → GreenE-BoxAdapter
IC3	GreenE-BoxAdapter → DataHandler
IC4	PaymentCalculator → GreenE-BoxAdapter
IC5	DataHandler → NetworkController
IC6	NetworkController → Locker/Unlocker

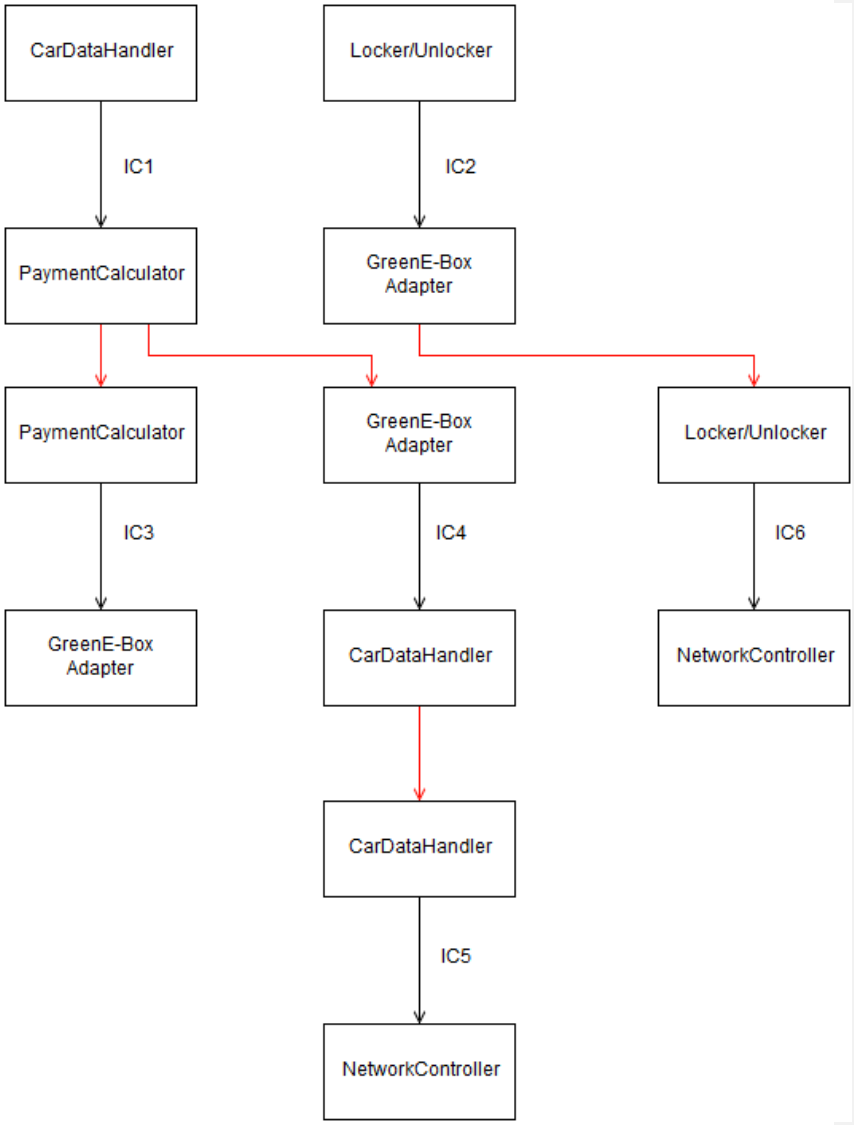

Figure 4: CarApplication components' integration order. Red arrows indicates the temporal and functional dependency between the single test cases and their components

2.4.2 SUBSYSTEM INTEGRATION SEQUENCE

As just said before, there is no clear subdivision into subsystems. However, the integration of functionalities of ITF2 and 3 depend on the integration of functionalities belonging to ITF1: thus, the integration sequence must be of ITF1 and then 2 and 3.

The integration testing of the CarApplication's components indeed can take place contemporarily to the integration testing of the Central Server components. Obviously, the integration between the two high-level components must be executed at least after the complete integration testing of the ITF2 functionalities.

PART 3: INDIVIDUAL STEPS AND TEST DESCRIPTION

3.1 Integration test case I1 (Database Interfaces)

Test Case Identifier	I1T1
Test Item(s)	RegistrationController → Database
Input Specification	Create typical Data for a query made by the RegistrationController.
Output Specification	Check if the creation of the tuples was successful or not, and why.
Purpose	The test procedure verifies if: ● the input are handled correctly by the two interface. ● the output is correct, in relation to the query made. ● the output can be returned to other components. ● errors are handled properly when the two components interact.
Procedure Steps	None
Stubs	None
Data Drivers	Data for the test, should produce these kind of queries: ● Correct Queries which create one User. ● Queries where one of the field to create a User is missing or wrong. ● Query which tries to create an already existent User.

Test Case Identifier	I1T2
Test Item(s)	LoginController → Database
Input Specification	Create typical Data for a query made by the LoginController.
Output	Check if the tuple searched is existent or not.

Specification	
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, in relation to the query made. • the output can be returned to other components. • errors are handled properly when the two components interact.
Procedure Steps	None
Stubs	None
Data Drivers	Data for the test, should produce these kind of queries: • Correct Queries which return only one User. • Incorrect Queries which return multiple Users. • Queries where one of the field is missing or wrong. • Queries which tries to return a non-existent User.

Test Case Identifier	I1T3
Test Item(s)	SearchController → Database
Input Specification	Create typical Data for a query made by the SearchController.
Output Specification	Check if the tuples returned match with the Query asked.
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, in relation to the query made. • the output can be returned to other components. • errors are handled properly when the two components interact.
Procedure Steps	None
Stubs	None
Data Drivers	Data for the test, should produce these kind of queries: • Correct Queries which return one or more Available Cars. • Queries where some of the fields are missing or wrong. • Correct Queries which return zero Available Cars.

Test Case Identifier	I1T4
Test Item(s)	MoneySavingHelper → Database
Input Specification	Create typical Data for a query made by the MoneySavingHelper.
Output Specification	Check if the tuples returned match with the Query asked.
Purpose	The test procedure verifies if: ● the input are handled correctly by the two interface. ● the output is correct, in relation to the query made. ● the output can be returned to other components. ● errors are handled properly when the two components interact.
Procedure Steps	None
Stubs	None
Data Drivers	Data for the test, should produce these kind of queries: ● Correct Queries which return one or more ChargingSafeArea. ● Queries where some of the fields are missing. ● Correct Queries which return zero SafeArea.

Test Case Identifier	I1T5
Test Item(s)	ReservationController → Database
Input Specification	Create typical Data for a query made by the ReservationController.
Output Specification	Check if the writing of the tuples was successful or not, and why.
Purpose	The test procedure verifies if: ● the input are handled correctly by the two interface. ● the output is correct, in relation to the query made. ● the output can be returned to other components. ● errors are handled properly when the two components interact.
Procedure Steps	None

Stubs	None
Data Drivers	Data for the test, should produce these kind of queries: • Correct Queries which create one Reservation from existing User and Available Car. • Queries where one of the field is missing or wrong. • Query which tries to reserve an already Reserved Car. • Query which tries to create a car for a non-existent User.

Test Case Identifier	I1T6
Test Item(s)	ExpirationController → Database
Input Specification	Create typical Data for a query made by the ExpirationController.
Output Specification	Check if the deletion of the tuples was successful or not, and why.
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, in relation to the query made. • the output can be returned to other components. • errors are handled properly when the two components interact.
Procedure Steps	None
Stubs	None
Data Drivers	Data for the test, should produce these kind of queries: • Correct Queries which delete one Reservation from existing User and Reserved Car. • Queries where one of the field is missing or wrong. • Query which tries to Delete a non-existent Reservation. • Query which tries to delete a Reservation made by a User different from the one who asked for it.

Test Case Identifier	I1T7
Test Item(s)	EndRideController → Database
Input Specification	Create typical Data for a query made by the EndRideController.
Output Specification	Check if the tuples returned match with the Query asked.
Purpose	The test procedure verifies if: ● the input are handled correctly by the two interface. ● the output is correct, in relation to the query made. ● the output can be returned to other components. ● errors are handled properly when the two components interact.
Procedure Steps	None
Stubs	None
Data Drivers	Data for the test, should produce these kind of queries: ● Correct Queries which return the amount of money the User who ended the Ride has to pay. ● Queries where some of the fields are missing or wrong. ● Queries which tries to return payment Information of a User who has not done any ride.

Test Case Identifier	I1T8
Test Item(s)	CarDataController → Database
Input Specification	Create typical Data for a query made by the CarDataController.
Output Specification	Check if the tuples written match with the Query asked.
Purpose	The test procedure verifies if: ● the input are handled correctly by the two interface. ● the output is correct, in relation to the query made. ● the output can be returned to other components. ● errors are handled properly when the two components interact.

Procedure Steps	This test should be performed after I1T7.
Stubs	None
Data Drivers	Data for the test, should produce these kind of queries: • Correct Queries which write only the Status of one Car at time. • Queries where some of the fields are missing or wrong. • Queries which tries to write a wrong Status inside a Car tuple.

3.2 Integration test case I2

Test Case Identifier	I2T1
Test Item(s)	ReservationController → ExpirationHelper
Input Specification	Create typical ReservationController Input.
Output Specification	Check if a Reservation Timer has been associated to a created Reservation, and if ended reservations are deleted.
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the output can be returned to other components. • errors are handled properly when the two components interact.
Procedure Steps	This test should be performed after all I1T5 and I1T6.
Stubs	• LicenseControlHelper • PaymentController • CarDataController
Data Drivers	Data for the test, should include: • Reservation request, for an Available Car. • Reservation request for a Car not Available. • Reservation request with some of the input missing.

Test Case Identifier	I2T2
Test Item(s)	ClientEventDispatcher → LoginController
Input Specification	Create typical ClientEventDispatcher Input, which uses the LoginInterface.
Output Specification	Check if the Interface is correctly used, users can be created, and success and failure messages are returned correctly.
Purpose	The test procedure verifies if: ● the input are handled correctly by the two interface. ● the output is correct, given a right input. ● the output can be returned to other components. ● errors are handled properly when the two components interact.
Procedure Steps	This test should be performed after I1T2.
Stubs	None
Data Drivers	Data for the test, should include: ● MobileApp request for login with all requested data. ● MobileApp request for login with missing data. ● MobileApp request for login with data already existing in the Database.

Test Case Identifier	I2T3
Test Item(s)	ClientEventDispatcher → RegistrationController
Input Specification	Create typical ClientEventDispatcher Input, which uses the RegistrationInterface.
Output Specification	Check if the Interface is correctly used, users can be created, and success and failure messages are returned correctly.
Purpose	The test procedure verifies if: ● the input are handled correctly by the two interface. ● the output is correct, given a right input. ● the output can be returned to other components. ● errors are handled properly when the two components interact.

Procedure Steps	This test should be performed after I1T1.
Stubs	None
Data Drivers	Data for the test, should include: • MobileApp request for registration with all requested data. • MobileApp request for registration with missing data. • MobileApp request for registration with data already existing in the Database.

Test Case Identifier	I2T4
Test Item(s)	CarEventDispatcher → CarDataController
Input Specification	Create typical CarEventDispatcher Input.
Output Specification	Check if the carDataController correctly update the database and send the correct output to the used component(EndRideController and EmergencyHelper)
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the correct output is sent to other components. • errors are handled properly when the two components interact.
Procedure Steps	This test should be performed after I1T8.
Stubs	• EndRideController • EmergencyHelper
Data Drivers	Data for the test, should include: • car status Informations messages. • endOfRide messages.

Test Case Identifier	I2T5
Test Item(s)	SearchController → MoneySavingHelper
Input Specification	Create typical Search with moneySaving request.
Output Specification	the correct list of available cars and destinations compatible with moneySavingOption.

Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the output can be returned to other components. • errors are handled properly when the two components interact. • only destinations compatible with MoneySavingOption are returned • only available cars are returned.
Procedure Steps	This test should be performed after all I1T3 and I1T4.
Stubs	none
Data Drivers	Data for the test, should include: • search request with moneySavingOption.

3.3 Integration test case I3

Test Case Identifier	I3T1
Test Item(s)	ReservationController → CarDataController
Input Specification	Create typical ReservationController Input.
Output Specification	Check if a Car is Reserved inside the DB, and it matches with the request made by the ReservationController.
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the output can be returned to other components. • errors are handled properly when the two components interact. • Requests from the components and changes inside the Database matches.
Procedure Steps	This test should be performed after I2T1 and I2T4.
Stubs	• LicenseControlHelper • PaymentController
Data Drivers	Data for the test, should include: • Reservation request, for an Available Car.

	• Reservation request for a Car not Available. • Reservation request with some of the input missing.

Test Case Identifier	I3T2
Test Item(s)	ExpirationHelper→ CarDataController
Input Specification	Create typical ExpirationHelper Input.
Output Specification	Check if the requested Reservation Delete is performed inside the Database.
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the output can be returned to other components. • errors are handled properly when the two components interact. • Requests from the components and changes inside the Database matches.
Procedure Steps	This test should be performed after I2T1 and I2T4.
Stubs	• PaymentController
Data Drivers	Data for the test, should include: • ExpirationHelper request for an existing reservation. • ExpirationHelper request for a non existing reservation.

Test Case Identifier	I3T3
Test Item(s)	CarLockController → CarDataController
Input Specification	Create Lock/Unlock request.
Output Specification	Check if the Lock/Unlock request is sent to the car.
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the lock/unlock operation is performed only if

	admissible. • errors are handled properly when the two components interact.
Procedure Steps	This test should be performed after I2T4.
Stubs	• CarApplication
Data Drivers	Data for the test, should include: • admissible Lock/Unlock request. • inadmissible Lock/Unlock request.

Test Case Identifier	I3T4
Test Item(s)	CarDataController → EndRideController
Input Specification	Create typical EndRide request.
Output Specification	check if the EndRideController correctly starts the EndRide procedure and communicates it to the ClientEventDispatcher .
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • errors are handled properly when the two components interact. • check if the endRide procedure correctly starts.
Procedure Steps	This test should be performed after I2T4.
Stubs	• paymentHelper
Data Drivers	Data for the test, should include: • information sent by parked or not parked cars.

Test Case Identifier	I3T5
Test Item(s)	CarDataController → EmergencyHelper
Input Specification	Create typical input data of an emergency situation.
Output Specification	check if the Emergency Helper correctly sends the emergency message with the operatorInterface.
Purpose	The test procedure verifies if:

	• the input are handled correctly by the two interface. • errors are handled properly when the two components interact. • check if the correct emergency messages are sent.
Procedure Steps	This test should be performed after I2T4.
Stubs	None
Data Drivers	Data for the test, should include: • empty battery notification • unused onRide car(parked for long time) notification.

Test Case Identifier	I3T6
Test Item(s)	ClientEventDispatcher → SearchController
Input Specification	Create typical input data of the clientEventDispatcher that include a seach request.
Output Specification	check if the list of available car (and the suggested destinations if the MoneySavingOption is requested) is correct
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • errors are handled properly when the two components interact. • check if only available car in the selected area are returned • check if the suggested destinations are correct.
Procedure Steps	This test should be performed after I2T5.
Stubs	none
Data Drivers	Data for the test, should include: • search request with and without moneySavingOption.

3.4 Integration test case I4

Test Case Identifier	I4T1
Test Item(s)	ReservationController → PaymentNotificationHelper
Input Specification	Create typical ReservationController Input.
Output Specification	Check if the amount of money requested to perform a ride of maximum length is reserved from the User Bank Account.
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the output can be returned to other components. • errors are handled properly when the two components interact. • External API are used properly to contact the Payment Provider.
Procedure Steps	This test should be performed after I3T1.
Stubs	• LicenseControlHelper
Data Drivers	Data for the test, should include: • Reservation request for a User which has enough money to perform the maximum ride. • Reservation request for a User which hasn't enough money to perform a maximum ride.

Test Case Identifier	I4T2
Test Item(s)	ClientEventDispatcher → ReservationController
Input Specification	Create typical ClientEventDispatcher Input, which uses the ReservationInterface.
Output Specification	Check if the request to reserve a Car is successful or not, and the reason for the failure. Also check if the deletion of a Reservation is notified to the ClientEventDispatcher.
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input.

	• the output can be returned to other components. • errors are handled properly when the two components interact.
Procedure Steps	This test should be performed after I3T1.
Stubs	• LicenseControlHelper • PaymentNotificationHelper
Data Drivers	Data for the test, should include: • Reservation request for a User which has enough money to perform the maximum ride. • Reservation request for a User which hasn't enough money to perform a maximum ride. • Reservation request for a user which has an invalid Driver's License.

Test Case Identifier	I4T3
Test Item(s)	ClientEventDispatcher → CarLock
Input Specification	Create typical ClientEventDispatcher Input that request to lock or unlock the car.
Output Specification	lock/unlock request to the selected car.
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the lock/Unlock requests are correctly notified to the car
Procedure Steps	This test should be performed after I3T3.
Stubs	• CarApplication
Data Drivers	Data for the test, should include: • lock/Unlock requests by an User.

Test Case Identifier	I4T4
Test Item(s)	ClientEventDispatcher → EndRideController
Input Specification	Create typical ClientEventDispatcher Input that notifies that the user is leaving the car.

Output Specification	check if the correct payment is charged and the new car status is available
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the output can be returned to other components. • errors are handled properly when the two components interact. • check if the charge is correctly calculated.
Procedure Steps	This test should be performed after I3T4.
Stubs	• PaymentHelper
Data Drivers	Data for the test, should include: • notifications about a user that is leaving the parked car.

Test Case Identifier	I4T5
Test Item(s)	ExpirationHelper → PaymentNotificationHelper
Input Specification	Create typical Expiration Input(car not taken after an hour).
Output Specification	Check if the 1€ payment is request to the PaymentNotificationHelper
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the output can be returned to other components. • errors are handled properly when the two components interact. • External API are used properly to contact the Payment Provider.
Procedure Steps	This test should be performed after I3T2.
Stubs	none
Data Drivers	Data for the test, should include: • Expiration of reservation request.

3.5 Integration test case I5

Test Case Identifier	I5T1
Test Item(s)	ReservationController → LicenseControlHelper
Input Specification	Create typical ReservationController Input.
Output Specification	Check if the output is consistent with the data inserted (i.e. if it let only Users with a valid Driver License to reserve a Car).
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the output can be returned to other components. • errors are handled properly when the two components interact. • External API are used properly to contact the License Provider.
Procedure Steps	This test should be performed after I4T2.
Stubs	None
Data Drivers	Data for the test, should include: • Reservation request for a user which has a valid Driver's License. • Reservation request for a user which has an invalid Driver's License.

Test Case Identifier	I5T2
Test Item(s)	RegistrationController → LicenseControlHelper
Input Specification	Create typical RegistrationController Input.
Output Specification	Check if the output is consistent with the data inserted (i.e. if it let only Users with a valid Driver License to register).
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the output can be returned to other components. • errors are handled properly when the two

	components interact. • External API are used properly to contact the License Provider.
Procedure Steps	This test should be performed after I4T2 and I2T3.
Stubs	None
Data Drivers	Data for the test, should include: • Registration request for a user which has a valid Driver's License. • Registration request for a user which has an invalid Driver's License.

Test Case Identifier	I5T3
Test Item(s)	EndRideController → PaymentHelper
Input Specification	Create typical Input about the end of a ride(user is leaving a parked car).
Output Specification	check if the payment request is correctly sent through the PaymentInterface
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the output can be returned to other components. • errors are handled properly when the two components interact.
Procedure Steps	This test should be performed after I4T4
Stubs	• PaymentProvider
Data Drivers	Data for the test, should include: • notification about an user that is leaving his car.

3.6 Integration test case IC1

Test Case Identifier	IC1
Test Item(s)	Data Handler → Payment Calculator
Input Specification	Create typical data Handler Input.

Output Specification	Check if the payment Calculator output is consistent with the input data
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the output is correct, given a right input. • the output is returned to the greenBoxAdapter. • the subsystem correctly identify the end of the ride
Procedure Steps	None
Stubs	• NetworkController • GreenBoxAdapter
Data Drivers	Data for the test, should produce these kind of queries: • report of typical information about car status to the data Handler • report typical car status information during the end of the ride to the data Handler

3.7 Integration test case IC2

Test Case Identifier	IC2
Test Item(s)	Locker/Unlocker→ GreenBoxAdapter
Input Specification	Create Lock/unlock request
Output Specification	check if the GreenBox receive the correct Lock/Unlock Request
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • the Lock/Unlock operation are performed correctly.
Procedure Steps	None
Stubs	None
Data Drivers	Data for the test, should produce these kind of queries: • lock/unlock requests

3.8 Integration test case IC3

Test Case Identifier	IC3T1
Test Item(s)	GreenBoxAdapter→ DataHandler
Input Specification	Create typical GreenBoxAdapter input data
Output Specification	• Network controller received information
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • check if the Network controller receive the correct information
Procedure Steps	This test should be performed after IC1
Stubs	• NetworkController
Data Drivers	Data for the test, should produce these kind of queries: • report of typical GreenBoxAdapter input Data.

Test Case Identifier	IC3T2
Test Item(s)	PaymentCalculator→ GreenBoxAdapter
Input Specification	Create tipical GreenBoxAdapter input data
Output Specification	• GreenBox received payment information
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • check if the GreenBox received the correct payment information
Procedure Steps	This test should be performed after IC1
Stubs	• NetworkController.
Data Drivers	Data for the test, should produce these kind of queries: • report of tipical GrennBoxAdapter input Data.

3.9 Integration test case IC4

Test Case Identifier	IC4T1
Test Item(s)	DataHandler→NetworkController
Input Specification	Create tipical GreenBoxAdapter input data
Output Specification	• Network controller sent information
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • check if the Network controller send the correct information
Procedure Steps	This test should be performed after IC3
Stubs	• CarEventDispatcher.
Data Drivers	Data for the test, should produce these kind of queries: • report of tipical GrennBoxAdapter input Data.

Test Case Identifier	IC4T2
Test Item(s)	NetworkController→Locker/Unlocker
Input Specification	Create tipical GreenBoxAdapter input data
Output Specification	• lock/Unlock request to GreenBox
Purpose	The test procedure verifies if: • the input are handled correctly by the two interface. • check if the GreenBox receive the correct Lock/Unlock request
Procedure Steps	This test should be performed after IC2
Stubs	None
Data Drivers	Data for the test, should produce these kind of queries: • request of Lock/Unlock by the serverApp

PART 4: TOOLS AND TEST EQUIPMENT REQUIRED

For the sake of execution of an effective integration testing, a set of automated tools are going to be exploited during this phase of the project.
With regards to the software components implementing the logic on the Central Server, since they rely on a Java Enterprise Edition runtime environment implementation, we are going to use three main tools.
The first of them is the Arquillan testing integration framework, whose use will allow us to whether check if our software components correctly interact with their surrounding environment. This aspect is particularly critical for what concerns the correct database connection management, and for other similar container-based issues, such as the right dependency injection specification.
The second of them is the Mockito framework for integration tests involving stubs.
Even though this framework was originally intended for the automation of unit tests allowing the creation of mocks, it supports scaffolding through the definition of appropriate stubs.
The possibility of using protocol invocation like HTTP for basic authentication, makes this tool a real useful instrument when testing those utilities related to the interaction with External Service Providers, or with the MobileApp or the CarApplication.
Finally, we will use the JUnit framework primarily for drivers' creation. Again, even though the predominant use of this tool is for unit testing, it offers some features that are really helpful in the debugging of an integration test (correct state of objects after method invocation, exception handling, etc…).
For what concerns the CarApplication, as we explained in the Design Document our implementation will be based on the Android OS, in order to correctly deploy our application on the Green E-box used in the Green Move Project.
Through the use of the Dexmaker API, we will still use normal Java automated testing tools, such as Mockito, for unit and integration testing, with the possibility of doing some manual testing since the actual high-level structure of our CarApplication comprehends a little number of software components.
Now, a specific and detached discussion is required in order to identify the correct and necessary equipment for our integration test to be carried out.
For the Central Server components, the setup of a separate Test Environment (TEV), that mimics the most possible the Production Environment, is warmly suggested for the sake of an easier and efficient test management, and for the testing of the components behaviour inside an environment (possibly really) close to the one where they are going to live.

This will be built on:

- the GlassFish Application Server for the server;
- the Java Enterprise Edition runtime;
- the Oracle MySQL relational database management system (RDBMS) for the database.

For the CarApplication components instead, the integration testing will be carried out using a Dealvik engine emulator running the Android Board (which provides the Software Abstraction Layer) and Embedded Board (which provides the Hardware Abstraction Layer) of the Green E-box, on which our application will be executed; then, we will connect it to an on-board diagnostic port (OBD) simulator.
In order to avoid the construction of one of our own, we will use one of the many available OBD simulator on the market (like the ECUsim 2000 OBD Simulator produced by Scantool, https://www.scantool.net/ecusim-2000/), which will provide a reliable simulation of the interaction with an electric control unit (ECU) of a real electric car.

PART 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 feature\risk-driven approach to component integration and testing.
Because of this choice, we are going to need a number of drivers and stubs to actually perform the necessary method invocations on the components to be tested; drivers will be mainly created in conjunction with the JUnit framework, while stubs will be created only during the integration tests.
Here follows a list of all the drivers and stubs that will be developed as part of the integration testing phase, together with their specific role.

5.1 Drivers

- **ClientEventDispatcher:** the purpose of this driver is to call the methods of all the component of the server that interact with it. In particular registration Request and login request for the RegistrationController and LoginController, search request for the searchController, reservation Request for the ReservationController , lock and unlock request for the CarLockController and endRide request for the EndRideController.
- **CarEventDispatcher:**the purpose of this driver is to reproduce Car information during a ride and send this information to the CarDataController, furthermore if has to send endRIde request to the EndRideController.
- **MobileApp**:the purpose of this driver is to reproduce tipical App request:Resgistration,login,search,lock,unlock,endRide.
- **CarApp**: the purpose of this driver is to produce Car information and endRide request for the CarEventDispatcher.
- **ExpirationHelper,CarLockController,ReservationController, and endRideController**: the purpose of these drivers is to call the methods of the CarData Controller with the typical request, these drivers are useful to test the interaction between the CarDataController and the database.
- **NetworkController**: the purpose of this driver is to send lock and unlock request to the locker/Unlocker.
- **GreenBoxAdapter**: the purpose of this driver is to reproduce and sent car information to the carDataHandler.
- **Car**: the purpose of this driver is to create typical car information and sends it to the GreenBoxAdapter.

Commentato [1]: in teoria non ci serve!
Ho messo come test equipment una scheda elettronica che simula il comportamento della CarControlUnit.
in teoria comunque l'adapter si interfaccia con l'embedded board della green e-box, ci servirebbe quel driver!

5.2 Stubs

- **LicenseControlHelper, PaymentController, CarDataController:** The purpose of this stubs is to reproduce the behaviour of their counterparts. They will have a set of valid data created ad-hoc in order to return the validity of data passed to them. They will also have a log where they can write the methods accessed by other classes.
- **EndRideController:** The purpose of this stub is to return true, the ID of the of a User who ends a ride. A set of data has to be created to perform different tests. It will also have a log where it can write the methods accessed by other classes.
- **EmergencyHelper:** The purpose of this stub is to have a log where it can write the methods accessed by other classes.
- **CarApplication:** The purpose of this stub is to have a log where it can write the methods accessed by other classes, and return a confirmation of received messages.
- **ClientApplication:** The purpose of this stub is to have a log where it can write the methods accessed by other classes, and return a confirmation of received messages.
- **NetworkController:** The purpose of this stub is to simulate the connection with the Application Server: a log is implemented, and a method to return the confirmation of received messages.
- **Green E-Box:** The purpose of this stub is to have a log where it can write the methods accessed by other classes, and return a confirmation of received messages.
- **External Providers:** The purpose of this stub is to have a log where it can write the methods accessed by other classes, and return a confirmation of received messages.

PART 6: EFFORT SPENT

	Cannas	Castiglioni	Loiacono
28.12	2	0	1
29.12	4	5	2
30.12	2	3	3
02.01	2	0	2
03.01	2	1	0
04.01	5	3	3
05.01	0	0	3
09.01	4	1	4
10.01	3	4	1
11.01	2	1	0
12.01	0	1	0
13.01	0	1	1
SUM	24	20	20