POLITECNICO
MILANO 1863

PROJECT for SOFTWARE ENGINEERING 2

Testing Document v1

Politecnico di Milano
A.A. 2016-2017
Prof.ssa Elisabetta Di Nitto

Students:
Diego Gaboardi
Giorgio Giardini
Riccardo Giol

INDEX

1 INTRODUCTION

1.1 PURPOSE AND SCOPE

This document is the Integration Test Plan Document for PowerEnjoy.

The main purpose is to describe in a clear and comprehensive way how we plan to accomplish the integration test, in order to integrate all different subsystems that make up our application, allowing in this way a correct and consistent realization of the whole project.

In particular, we're going to provide:

- A global vision in which there are the principal subsystems and subcomponents that we want to integrate and test;
- A description of the approach that we will follow for test implementation;
- The effective order for components and subsystems integration, with a detailed description of each integration step, including the expected output for each input data;
- Some measures of expected components performance in order to completely satisfy the requirements;
- A list of all the tools and environments that will have to be used during test activity;
- The Drivers and Stubs used to test the components of our system, grouped in sub-systems.

1.2 DEFINITIONS AND ACRONYMS

1.2.1 DEFINITIONS

- SUBCOMPONENT: the components defined in the DD document are decomposed in subcomponents that have a lower-level of abstraction.

- SUBSYSTEM: it is a high-level group of subcomponents that implements a specific function defined in the requirement part of RASD document.

1.2.2 ACRONYMS

- DD: Design Document.
- RASD: Requirement Analysis and Specifications Document.
- DBMS: DataBase Management System.
- SDK: Software Development Kit.
- GPS: Global positioning System.
- CPU: Central Processing Unit.
- RAM: Random Access Memory.
- IDE: Integrated Development Environment.
- GUI: Graphical User Interface.

1.3 REFERENCE DOCUMENTS

- RASD document ("RASD_GABOARDI_GIARDINI_GIOL_V2 document.pdf");
- DD document ("DD_Gaboardi_Giardini_Giol_v1 document.pdf");
- Specification document ("Assignments AA 2016-2017.pdf");
- Examples:
  - The Integration testing example document;
  - The Sample Integration Test Plan Document.

2 INTEGRATION STRATEGY

2.1 ENTRY CRITERIA

In this paragraph, we are going to describe all the conditions necessary for starting the Integration Testing of our project.

First of all the Requirement Analysis and Specifications Document (RASD) must be written in order to have all the functional and non-functional requirements already defined between the customers and the developers. At the same time also the Design Document (DD) is required because it is necessary to be conscious about the components of the system and their interactions.

When the Integration Testing begins we must have already started the Coding and Unit Test phase. However these two phases can be done partially in parallel, progressively integrating modules in subsystems as soon as they are implemented.

For this reason we give an estimation of the minimum percentage of completion of the components of our system required in order to start this phase. This values are calculated reflecting the order in which components are integrated.

- 100% for the DataController component;
- At least 95% for the ClientController and CarController components;
- At least 80% for the CourseController component;
- At least 60% for the ReservationController component;
- At least 40% for the AssistantController component;
- At least 40% for the client-side components.

In this way we evaluated that the two phases would be done concurrently in the most efficient way.

2.2 ELEMENTS TO BE INTEGRATED

In this paragraph, we are going to list all the components that will be incrementally integrated during the Integration Testing.

For this purpose we took into account the Component Diagram defined in the Design Document in which we showed a high-level representation of the components of our system focusing on their relationships. However, some components will be decomposed in sub-components in order to have a model of our system with a lower level of abstraction. This will help us to define the best strategy for integrating the components in subsystems.

The components are the following:

- Data Controller: it is the component that interacts with the DBMS in order to obtain the data stored or to save new tuples.
- ClientController: it is the component that handles the client accounts and so it is decomposed in several sub-components that carry out the required tasks: ClientRegistrationController, CientLoginController and ClientProfileController. At the same time it also manages the current state of the client and so it is composed also by a ClientStateController.
- CarController: it is the component which handles all the information about cars: position, state and battery. It is composed by a CarStateController and a CarBatteryController.
- CourseController: it is the component which manages the courses. This is one of the main functionality offered by our system and so it has to interact with other components such as the ParkingStationController, the SafeAreaController and the ChargeCalculator.
- ReservationController: this component is responsible for client reservations and interacts with almost the same components of the CourseController.
- AssistantController: it manages assistant accounts and so it has subcomponents similar to the ClientController component such as the AssistantRegistrationController, the AssistantLoginController and the AssistantProfileController.

- Some commercial components which have been already developed such as the DBMS and the PaymentSystem.

- The components on the client-side of our system and so the ClientApplication, the AssistantApplication and the CarApplication.

2.3 INTEGRATION TESTING STRATEGY

In this paragraph we are going to explain the strategy that we will use for performing the Integration Testing describing the order in which components are considered.

In particular our main goal is to parallelize as much as possible this phase with the Coding phase and at the same discover bugs early in the most critical components of our system. For this reason we will partially adopt a bottom-up approach and so we will start integrating components that depends only on already developed and integrated components. This approach will be combined with a critical module testing and so we will start by integrating the most important sub-systems of components.

This is the order chosen:

- The DataController with the DBMS because these components are used in almost all the tasks. The DBMS is an external and commercial component and so we considered it already developed.

- The subsystem about clients and so the ClientController and its sub-components: the ClientRegistrationController, the CientLoginController, the ClientProfileController and the ClientStateController.

- The subsystem about the cars and so the CarController, the CarStateController and the CarBatteryController.

- The components relative to courses. Even if this is the most critical subsystem we decided to integrate it at this point in order to have all the components necessary for this task already developed and integrated.

- The components relative to reservations. They are considered now for the same reasons of the previous point.

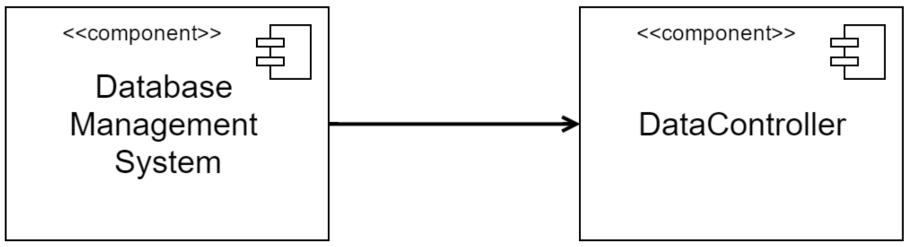
- The subsystem about assistants and so the AssistantController and its sub-components: the AssistantRegistrationController, the AssistantLoginController and the AssistantProfileController.

2.4 SEQUENCE OF COMPONENT / FUNCTION INTEGRATION

Following the strategy defined in the previous paragraph we are now going to describe how we integrate together our components. In order to show that a component is necessary to another one we use an arrow that connects the two components.

2.4.1 DATA CONTROLLER

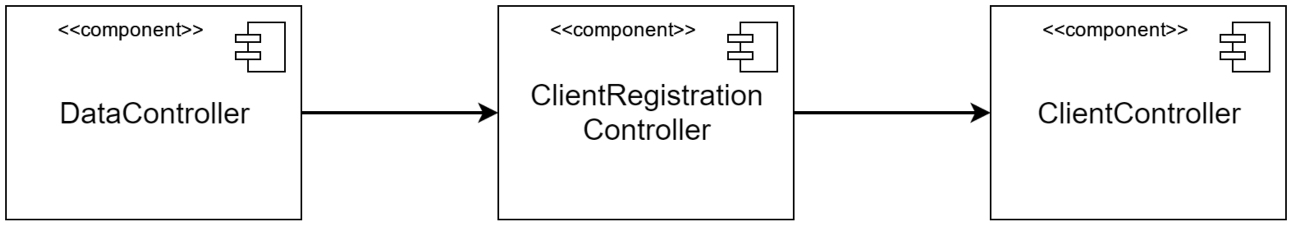
As we said, we start by integrating the DataController component with the Database Management System (DBMS). This is an obvious choice because these components are required almost everywhere in our systems and so it is a direct consequence of our bottom up approach.

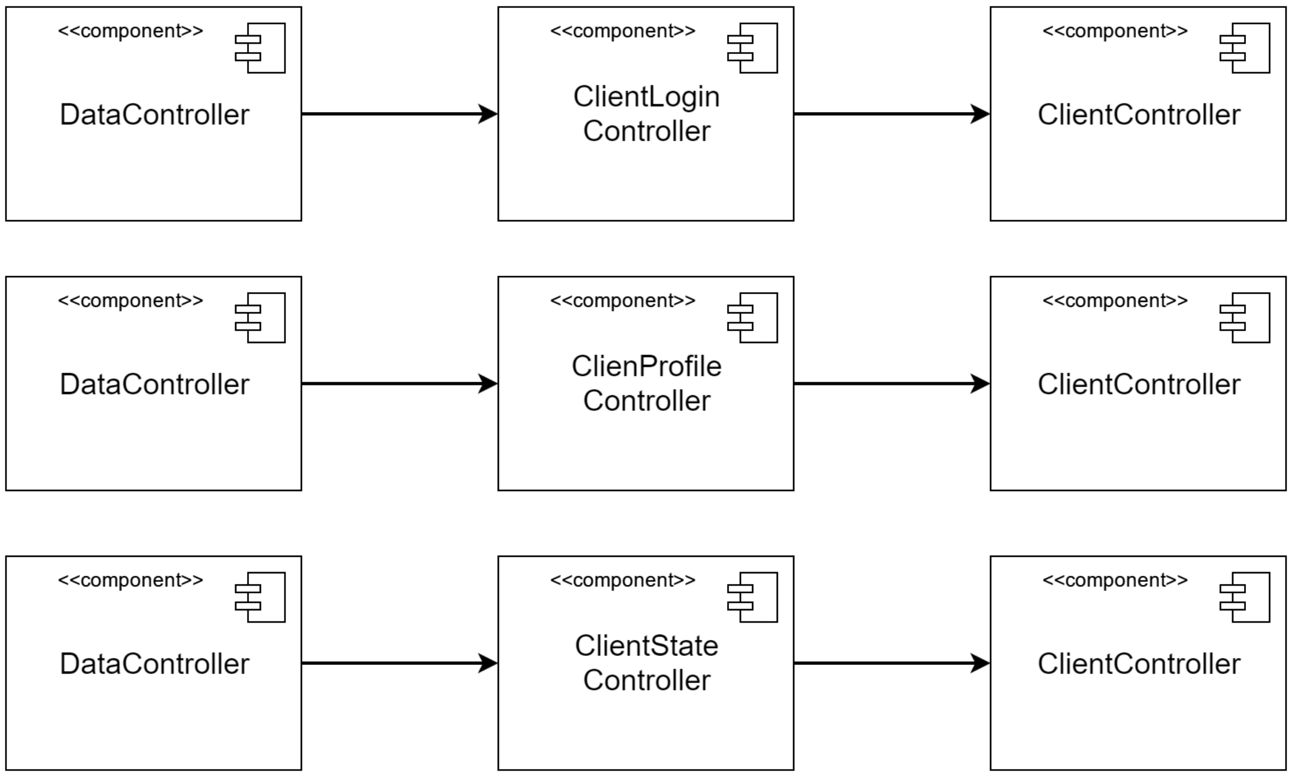


2.4.2 CLIENT MANAGEMENT SYSTEM

Then we consider the subsystem that manages clients. It is composed by several subcomponents which cover different aspects of the client administration and so can be integrated independently from one another. In all their functions they need to access or modify the data stored in the database and so they will be considered also with the DataController component.

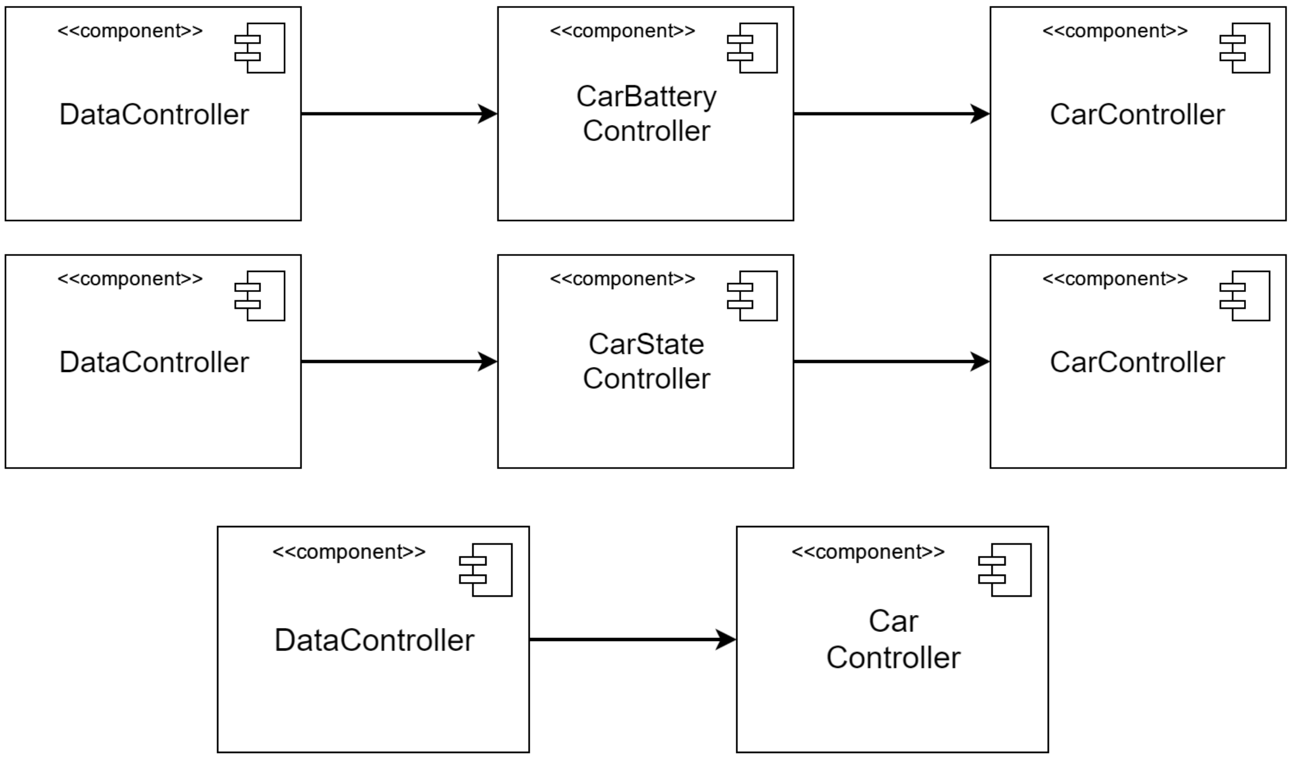
So we list the groups of components that are integrated.





2.4.3 CAR MANAGEMENT SYSTEM

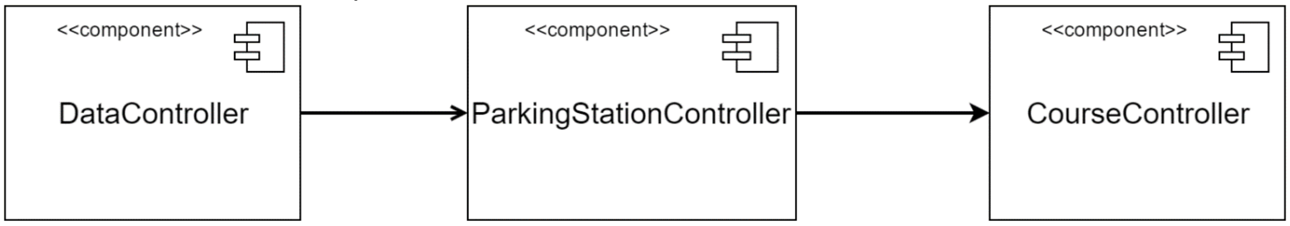
At this point the only component that can be integrated using only components that have been already developed is the CarController. It manages all the information about cars and so it has to interact with the database through the DataController, the CarStateController and the CarBatteryController.

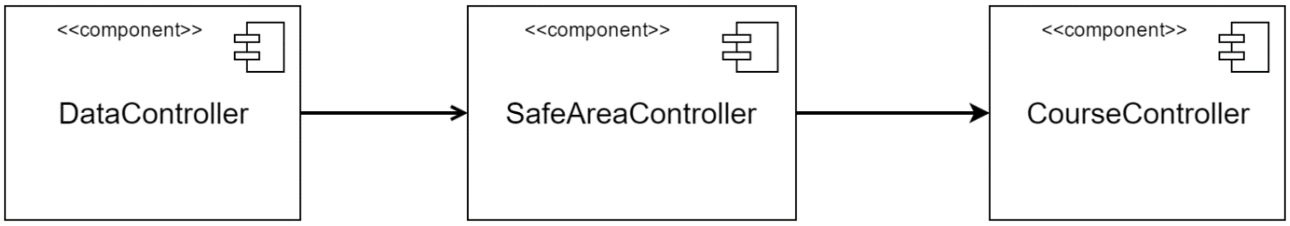


2.4.4 COURSE MANAGEMENT SYSTEM

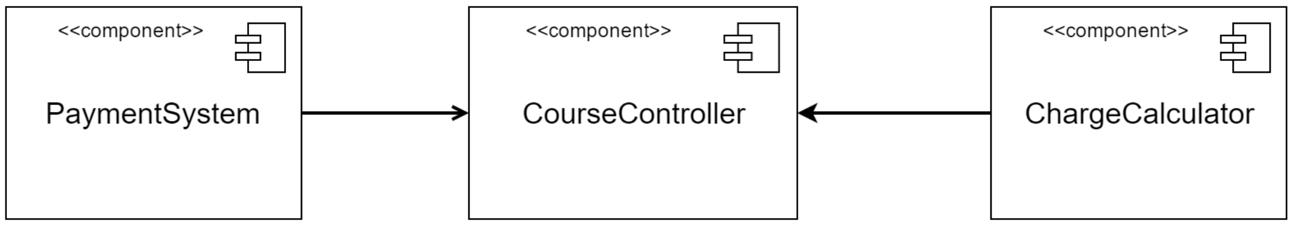
In this step, we have already integrated the subsystems about clients and cars and so now following the bottom-up strategy we could consider those of courses or assistants or reservation. We decided to start from the subsystem about courses because it certainly represents one of the most important functionalities of our system and so we applied a critical-module criterion.

In this subsystem there are several subcomponents that need to be integrated incrementally and the order will be described by the following diagrams. We start by considering the SafeAreaController and the ParkingStationcontroller whose functionalities are independent.

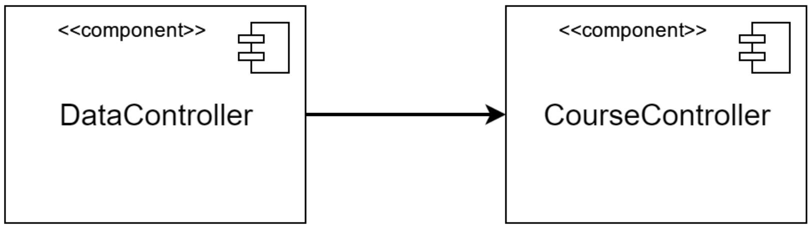




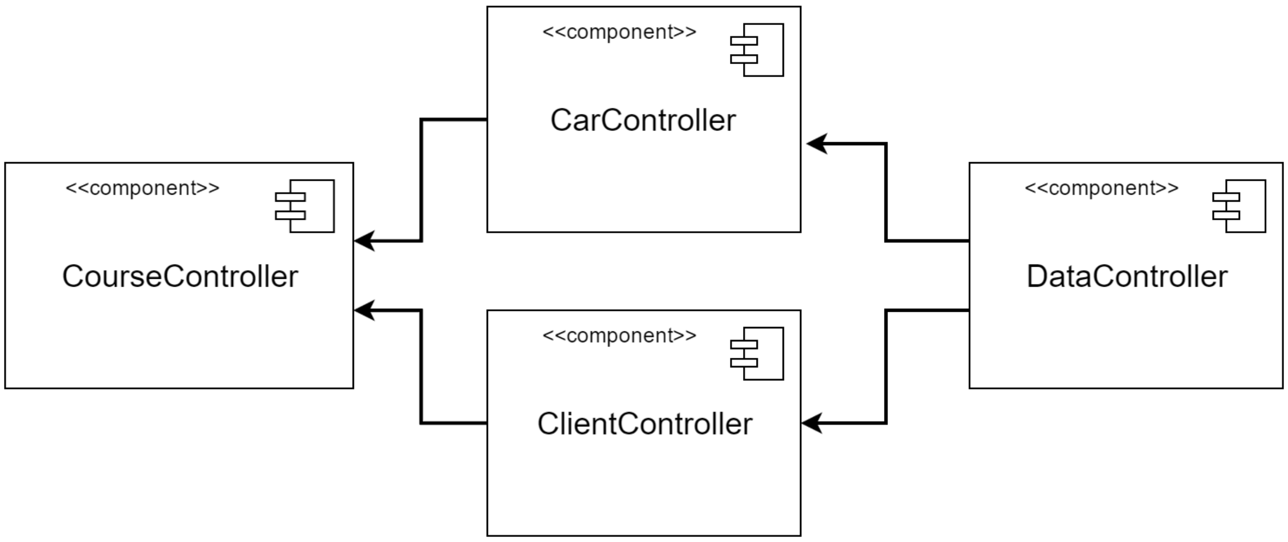
Then we consider the subcomponents that are necessary for the client payment. In this case we can use the PaymentSystem in a bottom-up approach because it is a commercial and external component that we considered already implemented and tested.



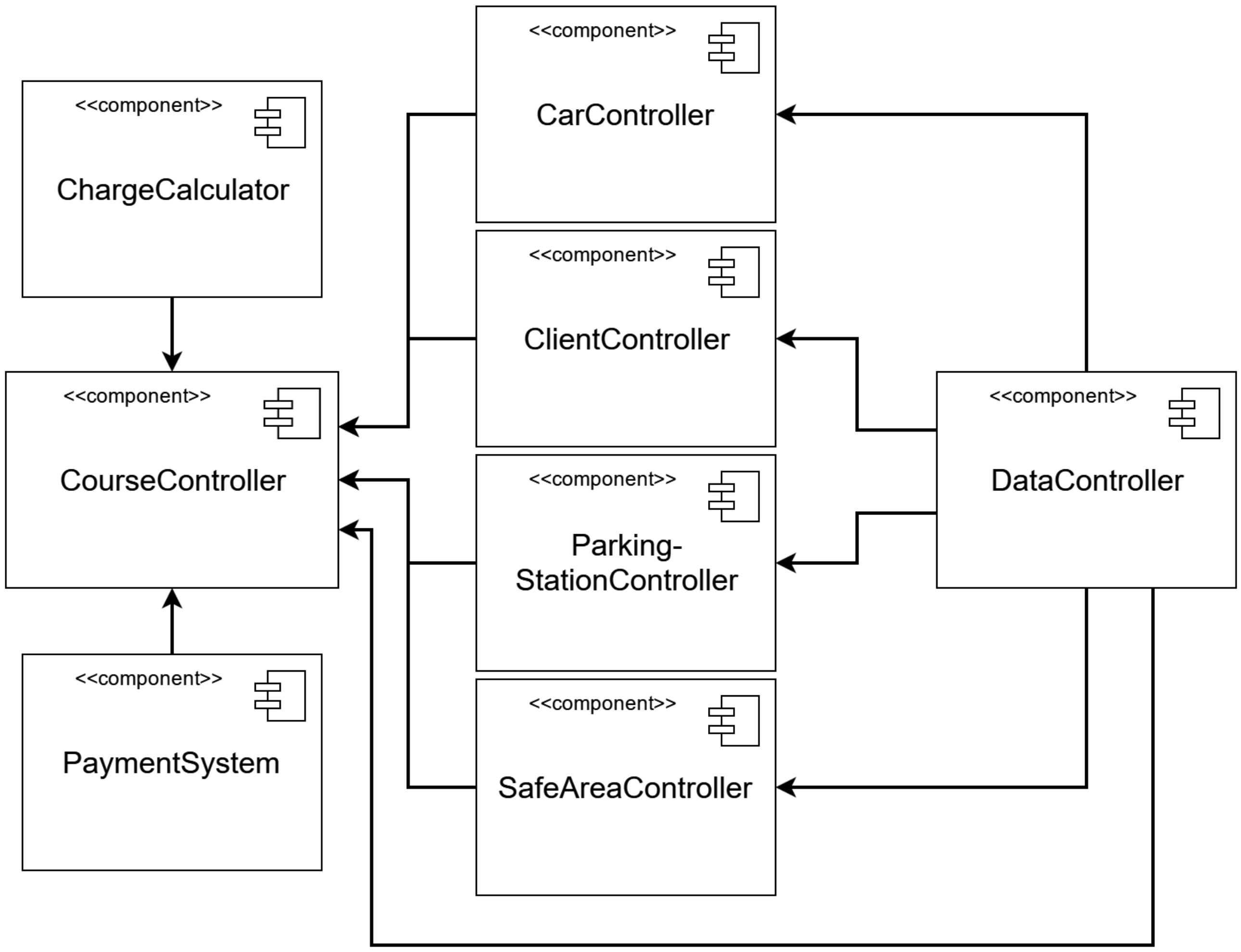
The CourseController can also interact directly with the DataController for example to insert a new Course in the database. So also these two components need to be considered.



Then the course controller has to modify the client and car state for example when the user decides to end his course. So we must integrate the CourseController also with the ClientController and the CarController.



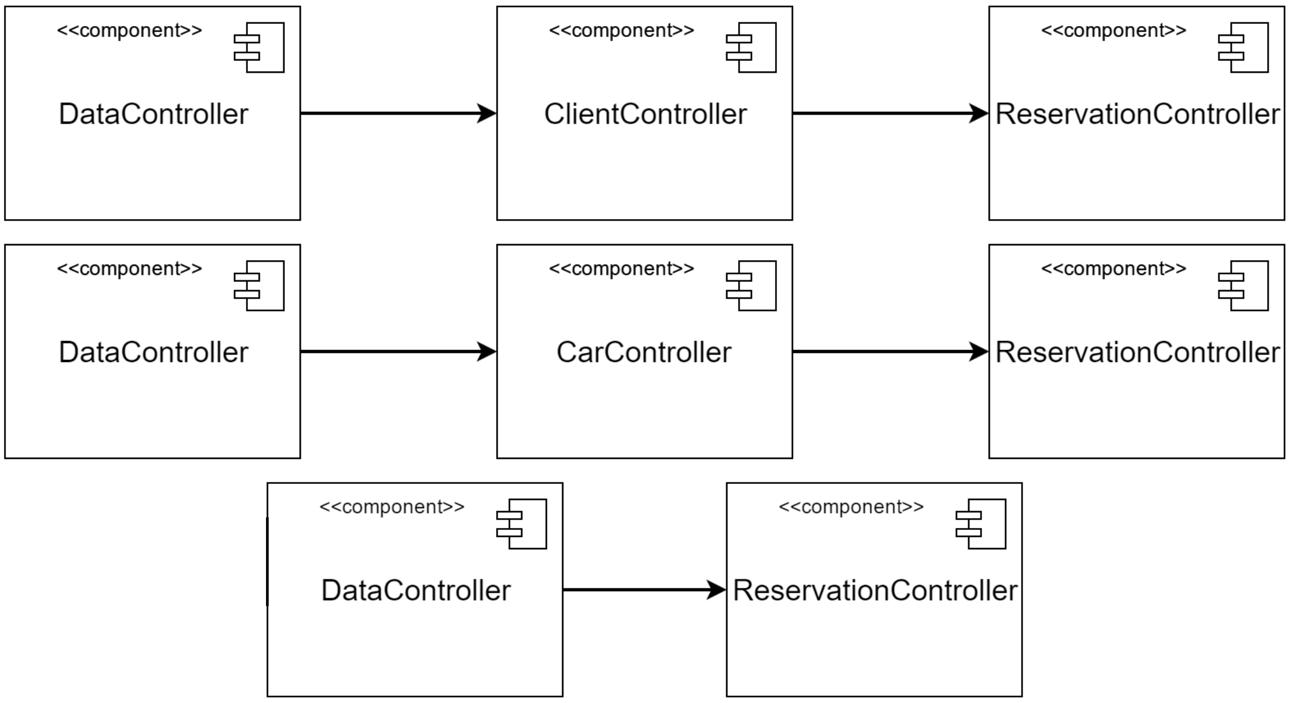
At the end the subcomponents of this system are integrated together.



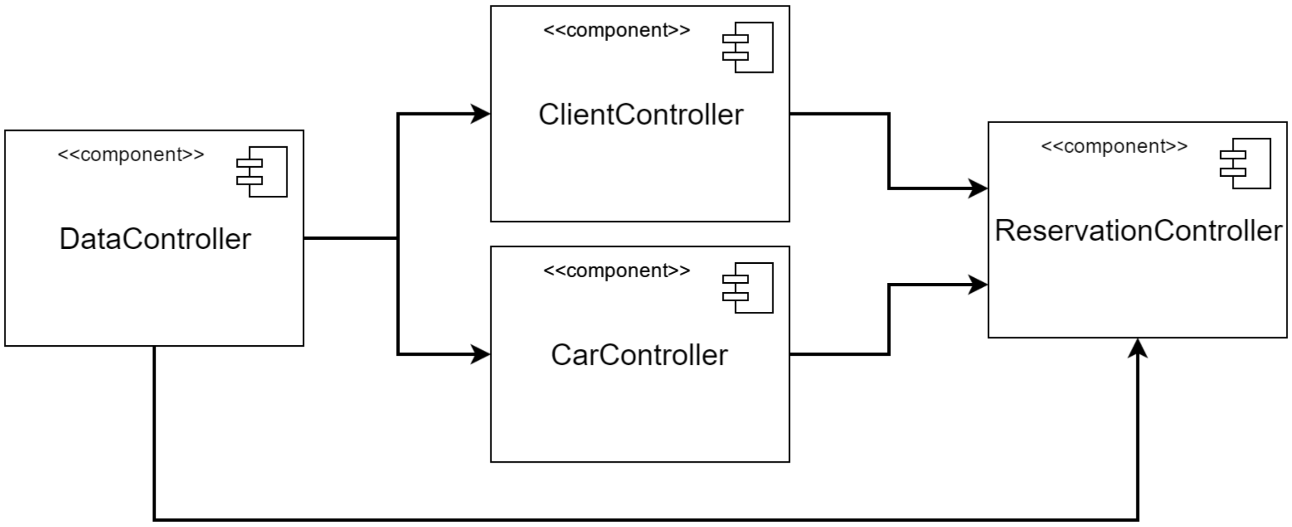
2.4.5 RESERVATION MANAGEMENT SYSTEM

Continuing to follow the critical-module approach we now consider the reservation subsystem. Reserving a car is one of the most important tasks of our system and the integration procedure has some similarities with the courses management system because some subcomponents are used in both cases.

The integration starts by considering the CarController and the ClientController which are necessary when the ReservationController needs to know or to modify the state of clients or cars. The ReservationController must interact also directly with the DataController in order to insert new reservation into the database.



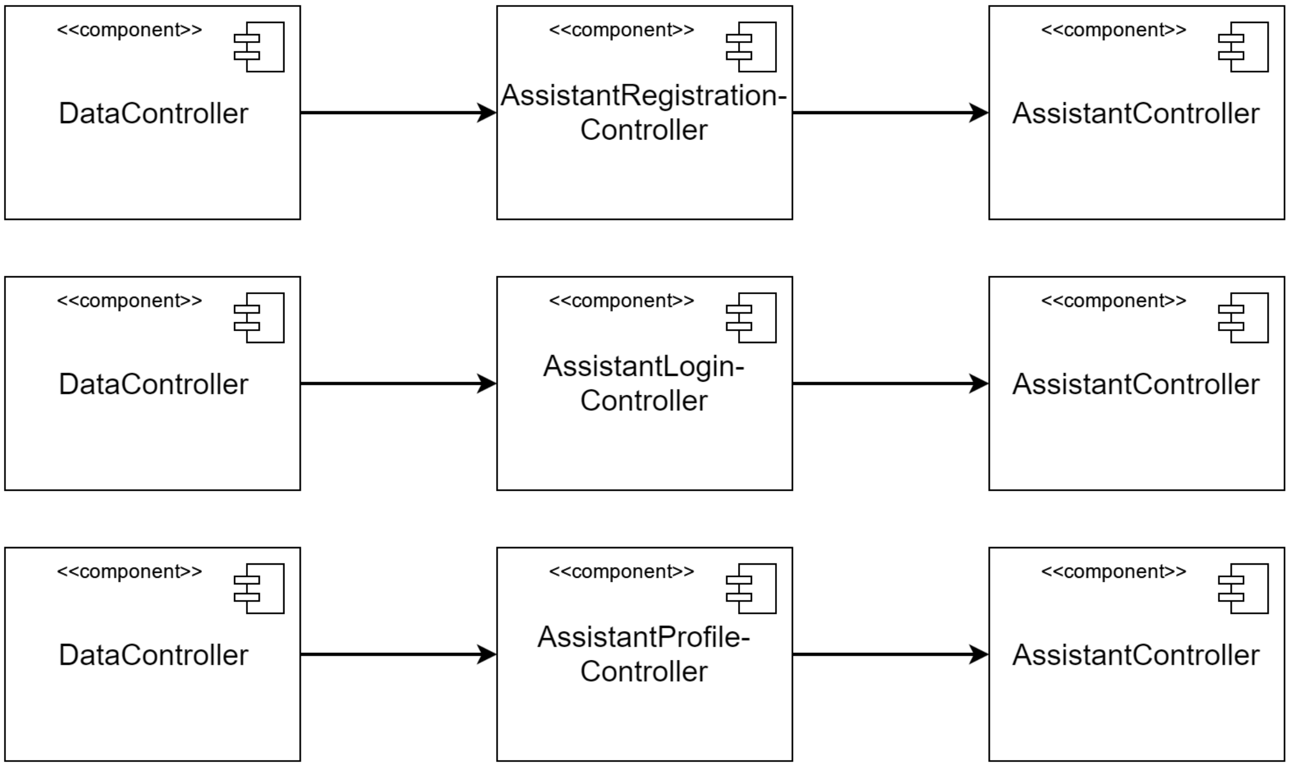
At the end these components are also integrated together.



2.4.6 ASSISTANT MANAGEMENT SYSTEM

At the end we consider the subsystem with the task of managing the assistants. It has some similarities with the components that manages clients because also here there are subcomponents that handle registration, login and profile modification.

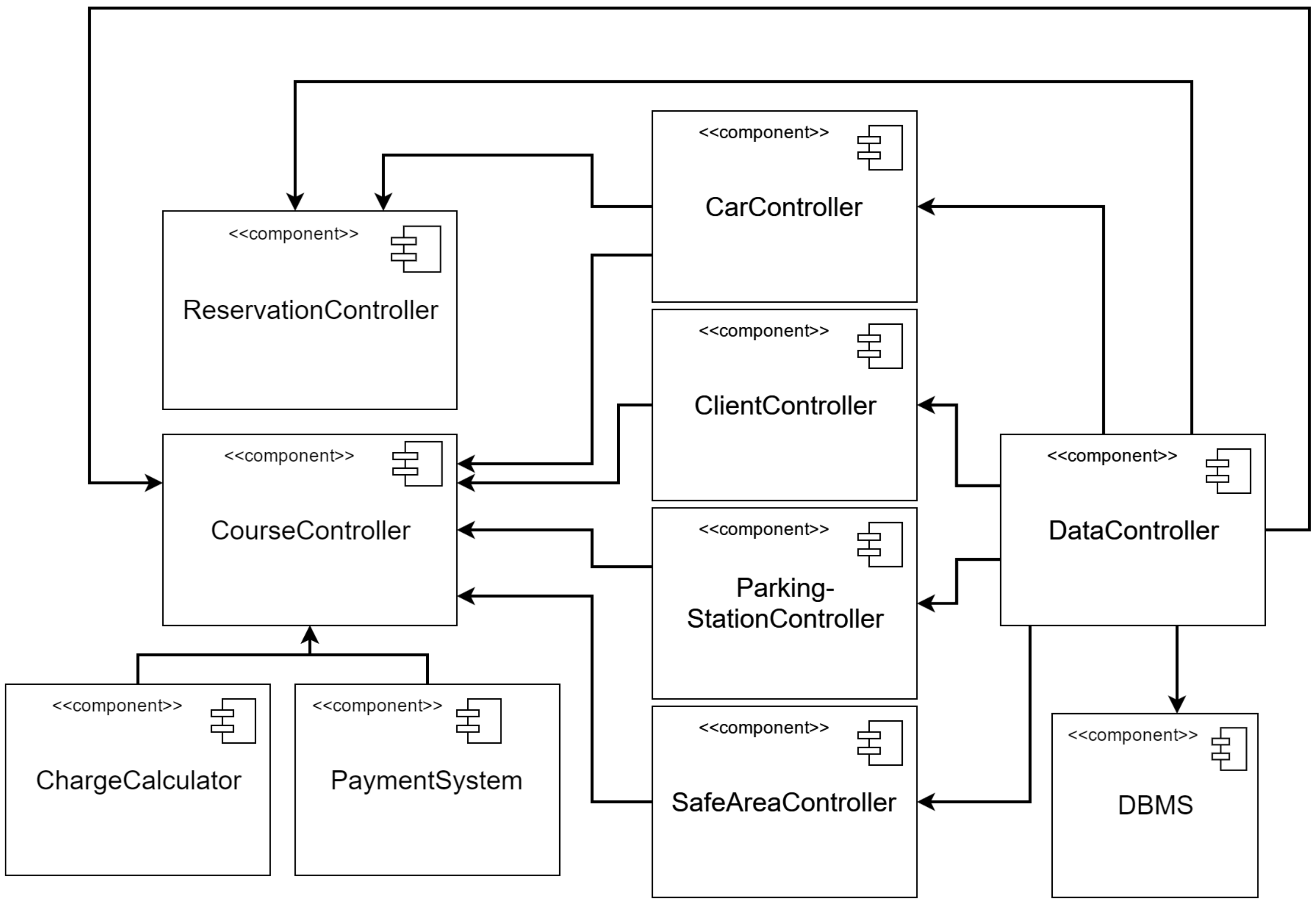
So also this subsystem is composed by subcomponents that are independent from one another and as a consequence it is not necessary to consider them all together.



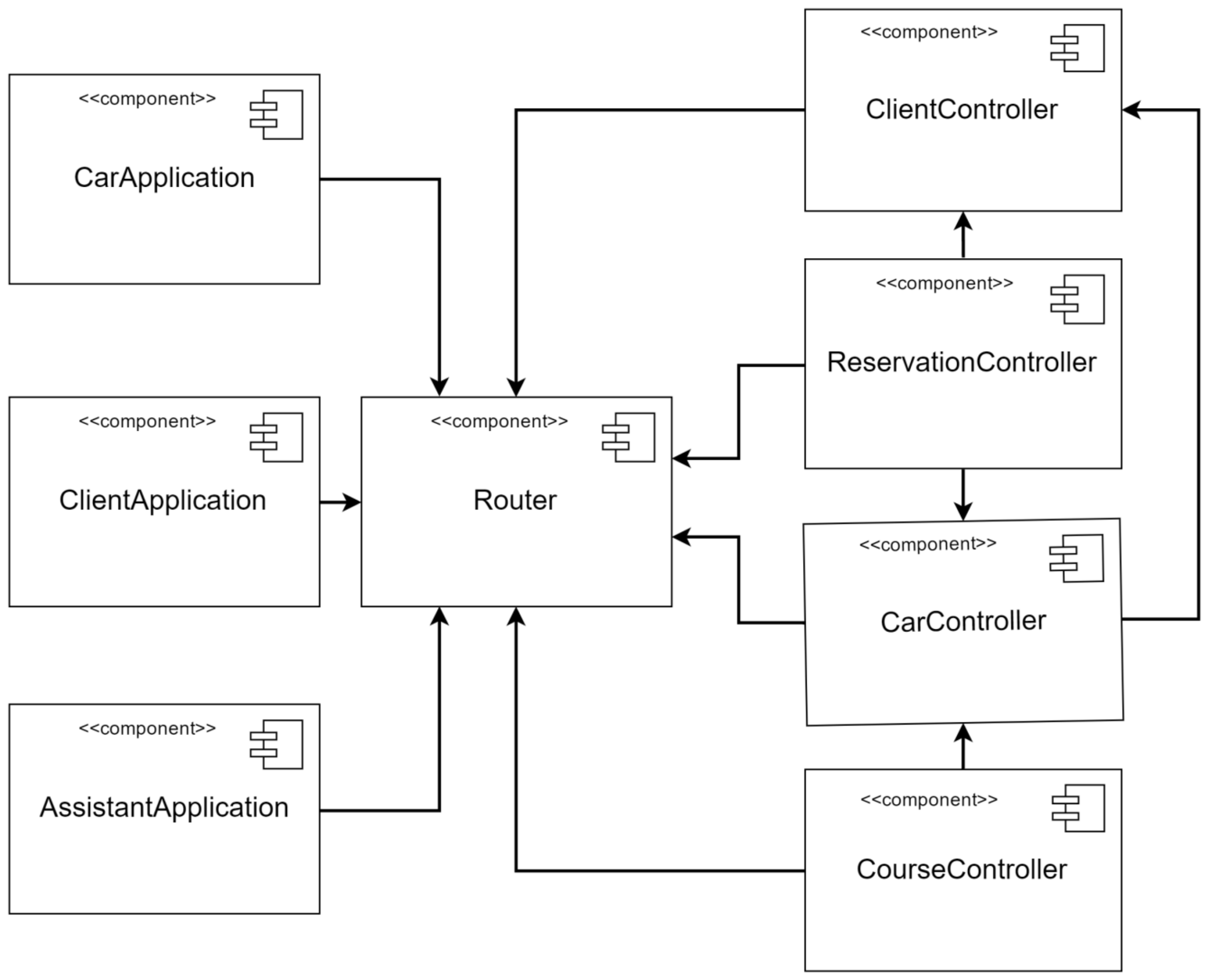
2.5 SUBSYSTEM INTEGRATION SEQUENCE

In the following paragraph we provide some diagrams in which we show how the subsystems integrated in the previous section can be integrated together in order to obtain the complete application.

In the first one there are all the components that manages the services that our system offers. In order to make the diagram more readable we did not insert some low-level subcomponents such as those about client and assistant management.



In this schema, we show how the previous components interact with the client-side components.



3 INDIVIDUAL STEP - TEST DESCRIPTION

In this chapter we'll analyse, for each pair of component that have to be integrated, the functionality that must be tested in order to guarantee a correct behaviour of the subsystem that they generate. In particular, for each possible input of the analysed functions, we'll focus on the expected output, providing a brief description of the effects on the system.

3.1 Client management system

3.1.1 ClientRegistrationController : DataController

insertClient(clientData)	
Input	*Effects*
Null parameters	NullArgumentException is throwed
An object containing some null attributes	NullArgumentException is throwed
An email already existing	InsertionFailedException is throwed
An invalid credit card	InvalidCreditCardException is throwed
An Invalid driving licence	InvalidDrivingLicenceException is throwed
All valid data	A new client is inserted in the database, with a unique code

deleteClient(clientCode)	
Input	*Effects*
Null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code	The client is removed from the database

3.1.2 ClientLoginController : DataController

verifyCredential(username,password)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing username	InvalidArgumentException is throwed
A valid and existing username, but an incorrect password	LoginFailedException is throwed
A valid and existing username with the corresponding password	Login is successfully done

getHomeProfile(clientCode)	
Input	*Effects*
Null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code	Client home is loaded

3.1.3 ClientProfileController : DataController

getClientData(clientCode)	
Input	*Effects*
Null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code	Client data are read from the database

modifyClientData(clientCode, newClientData)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
Invalid or incomplete new data	InvalidArgumentException is throwed
A valid code and valid data	Client old data are replaced with new data in the database

3.1.4 ClientStateController : DataController

getClientState(clientCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code	Client state is read from database

changeClientState(clientCode, newState)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
An invalid new state	InvalidArgumentException is throwed
A valid code and a valid state	State of the corresponding client is updated in the database

3.2 Car management system

3.2.1 CarStateController : DataController

getCarState(carCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code	Car state is read from database

changeCarState(carCode, newState, damage)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
An invalid new state	InvalidArgumentException is throwed
A valid code and a valid state	State of the corresponding client is updated in the database, if newState is “not available” the state will include also the damage entity

lockCar(carCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code which refer to a locked car	CarAlreadyLockedException is throwed
A valid and existing code which refer to an unlocked car	Car is set locked in the database and a message is sent to carApplication

unlockCar(carCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code which refer to an unlocked car	CarAlreadyUnlockedException is throwed
A valid and existing code which refer to a locked car	Car is set unlocked in the database and a message is sent to carApplication

3.2.2 CarBatteryController : DataController

getCarBattery(carCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code	Car battery level is read from database

setOnCharge(carCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A code corresponding to a car which is already on charge	CarAlreadyChargingException is throwed
A valid and existing code, corresponding to a car which is not on charge	Car is set on charge in the database

disconnectCarCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A code corresponding to a car which is already disconnected	CarAlreadyDisconnectedException is throwed
A valid and existing code, corresponding to a car which is on charge	Car is setted not on charge in the database

3.2.3 CarController : DataController

getNearestCarAvailable(position)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing position	InvalidArgumentException is throwed
A valid and existing position with no nearby available cars	Empty list is returned
A valid and existing position with some nearby available cars	A list with nearest available cars to the specified position is returned

getCar(carCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code	Car with the corresponding code is returned

3.3 Course management system

3.3.1 ParkingStationController : DataController

getNearestParkStationAvailable(position)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing position	InvalidArgumentException is throwed
A valid and existing position with no nearby available park stations	Empty list is returned
A valid and existing position with some nearby available park stations	A list with nearest available park stations to the specified position is returned

getRechargerAvailable(parkStationCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing park station code with no recharger available	Empty list is returned
A valid and existing park station code with some recharger available	A list of available recharger in the corresponding park station is returned

setRechargerOccupied(parkStationCode, rechargerCode)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing park station code	InvalidArgumentException is throwed
An invalid or not existing recharger code in the corresponding park station	InvalidArgumentException is throwed
All valid codes, corresponding to a recharger that is already occupied	RechergerAlreadyOccupiedException is throwed
All valid codes, corresponding to a recharger that is not occupied	Recharger is set occupied in the database

setRechargerFree(parkStationCode, rechargerCode)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing park station code	InvalidArgumentException is throwed
An invalid or not existing recharger code in the corresponding park station	InvalidArgumentException is throwed
All valid codes, corresponding to a recharger that is already free	RechergerAlreadyFreeException is throwed
All valid codes, corresponding to a recharger that is occupied	Recharger is set free in the database

3.3.2 SafeAreaController : DataController

verifySafeArea(position)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing position	InvalidArgumentException is throwed
A position which is not a safe area	Return false
A position which is a safe area	Return true

3.3.3 CourseController : PaymentSystem

makePayment(paymentData)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
Invalid payment data	InvalidArgumentException is throwed
Valid payment data	The transaction is executed

3.3.4 CourseController : ChargeCalculator

calculateCharge(courseTime)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
Invalid course time	InvalidArgumentException is throwed
Valid course time in second	Course charge is calculated proportionally to the course time and the value is returned

applyDiscount(discount[], courseCharge)	
Input	*Effects*
Some null parameter	NullArgumentException is throwed
Invalid course charge	InvalidArgumentException is throwed
Array of discount with at least one invalid element	InvalidArgumentException is throwed
Array of valid discounts and valid course charge	The total amount of discount is calculated and is applied to the course charge

3.3.5 CourseController : CarController

createCourseCarRelation(course, carCode)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A code corresponding to a car which already have a course relation	InvalidCreationException is throwed
A course which already have a relation with a car	InvalidCreationException is throwed
A valid and existing code corresponding to a car with no course relation, a course with no car relation	Database is updated adding the relation between course and car

deleteCourseCarRelation(course, carCode)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A code corresponding to a car which isn't in relation with course	InvalidArgumentException is throwed
A course which isn't in relation with the car	InvalidArgumentException is throwed
A valid and existing code corresponding to a car in relation with course	Database is updated removing the relation between course and car and setting to null the corresponding fields

3.3.6 CourseController : ClientController

createCourseClientRelation(course, clientCode)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A code corresponding to a client which already have a course relation	InvalidCreationException is throwed
A course which already have a relation with a client	InvalidCreationException is throwed
A valid and existing code corresponding to a client with no course relation, a course with no client relation	Database is updated adding the relation between course and client

deleteCourseClientRelation(course, clientCode)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A code corresponding to a client which isn’t in relation with course	InvalidArgumentException is throwed
A course which isn’t in relation with the client	InvalidArgumentException is throwed
A valid and existing code corresponding to a client in relation with course	Database is updated removing the relation between course and client and setting to null the corresponding fields

3.3.7 CourseController : DataController

createCourse(dataStart, timeStart)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid time or an invalid data	InvalidArgumentException is throwed
A valid time and a valid data	A new course with the corresponding dataStart and timeStart is created and added to the database

endCourse(course)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
A course already terminated	CourseAlreadyTerminatedException is throwed
A course not already terminated	Time end and data end are registered to the corresponding course in the database

3.4 Reservation management system

3.4.1 ReservationController : CarController

createReservationCarRelation(reservation, carCode)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A code corresponding to a car which already have a reservation relation	InvalidCreationException is throwed
A reservation which already have a relation with a car	InvalidCreationException is throwed
A valid and existing code corresponding to a car with no reservation relation, a course with no car relation	Database is updated adding the relation between reservation and car

deleteReservationCarRelation(reservation, carCode)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A code corresponding to a car which isn't in relation with reservation	InvalidArgumentException is throwed
A reservation which isn't in relation with the car	InvalidArgumentException is throwed
A valid and existing code corresponding to a car in relation with reservation	Database is updated removing the relation between reservation and car and setting to null the corresponding fields

3.4.2 ReservationController : ClientController

createReservationClientRelation(reservation, clientCode)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A code corresponding to a client which already have a reservation relation	InvalidCreationException is throwed
A reservation which already have a relation with a client	InvalidCreationException is throwed
A valid and existing code corresponding to a client with no reservation relation, a course with no client relation	Database is updated adding the relation between reservation and client

deleteReservationClientRelation(reservation, clientCode)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A code corresponding to a client which isn’t in relation with reservation	InvalidArgumentException is throwed
A reservation which isn’t in relation with the client	InvalidArgumentException is throwed
A valid and existing code corresponding to a client in relation with reservation	Database is updated removing the relation between reservation and client and setting to null the corresponding fields

3.4.3 ReservationController : DataController

startReservationCountdown(reservation)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
A reservation which already has an active reservation countdown	InvalidArgumentException is throwed
A reservation which not already has an active reservation countdown	A one-hour reservation countdown is started on the corresponding reservation

stopReservationCountdown(reservation)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
A reservation which not already has an active reservation countdown	InvalidArgumentException is throwed
A reservation which already has an active reservation countdown	The one-hour reservation countdown is stopped and reset

startCourtesyCountdown(reservation)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
A reservation which already has an active courtesy countdown	InvalidArgumentException is throwed
A reservation which not already has an active courtesy countdown	A five-minutes courtesy countdown is started on the corresponding reservation

stopCourtesyCountdown(reservation)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
A reservation which not already has an active courtesy countdown	InvalidArgumentException is throwed
A reservation which already has an active courtesy countdown	The five-minutes courtesy countdown is stopped and reset

3.5 Assistant management

3.5.1 AssistantRegistrationController : DataController

insertAssistant(assistantData)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An object containing some null attributes	NullArgumentException is throwed
An email already existing	InsertionFailedException is throwed
A name of an assistant which is not recognised as an employee of PowerEnjoy society	InsertionFailedException is throwed
All valid data	A new assistant is inserted in the database, with a unique code

deleteAssistant(assistantCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code	The assistant is removed from the database

3.5.2 AssistantLoginController : DataController

verifyCredential(username,password)	
Input	*Effects*
Some null parameters	NullArgumentException is throwed
An invalid or not existing username	InvalidArgumentException is throwed
A valid and existing username, but an incorrect password	LoginFailedException is throwed
A valid and existing username with the corresponding password	Login is successfully done

getHomeProfile(assistantCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code	Assistant home is loaded

3.5.3 AssistantProfileController : DataController

getAssistantData(assistantCode)	
Input	*Effects*
Null parameter	NullArgumentException is throwed
An invalid or not existing code	InvalidArgumentException is throwed
A valid and existing code	Assistant data are read from the database

4 PERFORMANCE ANALYSIS

With the performance analysis we evaluate in which conditions and with which tools the system works properly. A complete analysis can be performed during the implementation of the infrastructure, monitoring how the system responds with the structures used. However, it is useful to perform a preliminary analysis for the infrastructures required from the performances requested.

For the correct running of the mobile application, the smartphone has to provide at least:

- The correct tracking through GPS;
- Camera with 5Mpx;
- 3,5 MB of memory for the app;
- 80 MB of RAM to be execute;
- CPU with 1GHz in one core.

All these constraints are approximate, and some more close to real values can be detected in the future, with the implementation of the infrastructure.

The application will be developed with the framework Apache-Cordova for all the platforms. Than it is tested with a specific SDK for different smartphones.

5 TOOLS - TEST EQUIPMENT REQUIRED

5.1 TOOLS

The tests are all executed with some automatized tools, that decrease the writing time for the test and simplify and get more readable the solutions. It is even easier to find the error in the code, thanks to the debugging offer by the IDE Netbeans. The tools used will be:

- Junit;
- Mockito;
- Arquillian;
- Apache JMeter.

For the unit test and components test will be used JUnit and Mockito, while for the subsystem test will be used Arquillian too. These three are the tools for testing the Functional Requirements.

For testing the NonFunctional Requirements Apache Meter will be used.

The roles of these tools are different, and each one is relative to a different scope:

- Junit: is a unit testing framework. Its scope is to verify the correct behaviour, and so the result of a function through assertions. With this tool the interactions between components can be verified checking if they produce the expected result. We will use it even to verify if the methods will throw the correct exceptions with some specific parameters.

- Mockito: is a tool that permit to mock an environment for test a unit. It allows the interaction test between objects. It provides a mocking of some methods creating a scaffolding defined with stubs.

- Arquillian: is used for tests with containers. The test is similar to the Junit test, but it adds the container and so the environment in which the class is run. It also uses archives, defined using ShrinkWrap, that defines the micro-environment with only the strict data necessary to the class in test to work. With Arquillian it can be choose the container in which execute the class to test.

- Apache JMeter: is used to evaluate performances of a system. It has a GUI and can evaluate performances relative to any software (web applications) about heavy load on a DB or similar. It is possible to check even the behaviour of multiple test servers in a single interface. It is possible to create test plans that emulate the interaction of a user with the GUI of the application, and so evaluate the performances with many users.

We will evaluate the different performances in the different devices with some appropriate tools. For Android we use Systrace, for iOS XCode and for WindowsPhone a specific tool of Visual Studio.

5.2 TEST EQUIPMENT

Our system involves many different devices with different features, all to be tested to monitoring the performances.

For how concerns the mobile app, the possible different devices that can be involved are the smartphones with the different operative systems. We will try to cover all the possible smartphones that can be used, so for each operative system we test some smartphones with different screen widths and different CPU performances, in which the app can have different behaviours.

For the on board car system, we will use a specific tablet with OS Android, so the app is built for this specific tablet model.

For testing the backend of the system, we have to simulate the Application Server with Apache and the Data Server with MySQL. The server infrastructure is simulated as it will be in order to testing the performance. It has to run the same software of the original infrastructure.

6 STUBS, DRIVERS AND TEST DATA

6.1 PROGRAM DRIVERS

For the component integration and testing we have decided to adopt a bottom-up approach. For testing the components, we call their methods with the drivers, as Junit works. The drivers for the testing are:

- Data Controller Driver: this testing module invokes the methods of the Data Controller and their interaction with the Database Management System.

- Client Management Driver: this testing module invokes the methods of the subsystem that manages clients, so the methods of the Client Controller, Client Registration Controller, Client Login Controller, Client Profile Controller, Client State Controller and their interaction with the Data Controller.

- Car Management Driver: this testing module invokes the methods of the Car Controller, Car Battery Controller, Car State Controller and their interaction with the Data Controller.

- Course Management Driver: this testing module has the subsystems of Car and Client integrated. It invokes the methods of the Course Controller, Parking Station Controller, Safe Area Controller and their interaction with the Data Controller. It invokes also the methods of the subsystem relative to the Payment and so the Payment System and the Charge Calculator.

- Reservation Management Driver: this testing module integrates the subsystem of Car and Client. It invokes the methods of the Reservation Controller and its interaction with the Car Controller, Client Controller and Data Controller.

- Assistant Management Driver: this testing module invokes the methods of the Assistant Controller, Assistant Registration Controller, Assistant Login Controller, Assistant Profile Controller and their interaction with the Data Controller.

The stubs are not used indeed the testing processing follow the bottom-up criterion. A stub emulates a component not yet tested, but in the bottom-up criterion the simpler components are the first to be tested, so gone on the more complex components that use the simpler components found these already tested, without need to emulate them.

6.2 TEST DATA

To perform the tests explained in this document we need:

- A list of both valid and invalid candidate clients to test the Client Controller component. The set should contain instances exhibiting the following problems:
    - Null Objects
    - Null Fields
    - Not legal Driver Licence
    - Not legal Credit Card
    - Not valid Personal Data
    - Not valid email
    - Not valid Credentials
    - Not valid Code
    - Not valid State

- A list of both valid and invalid candidate cars to test the Car Controller component. The set should contain instances exhibiting the following problems:
    - Null Objects
    - Null Fields
    - Not valid Code
    - Not valid State
    - Not valid Position

- A list of both valid and invalid candidate assistants to test the Assistant Controller component. The set should contain instances exhibiting the following problems:
    - Null Objects
    - Null Fields
    - Not legal Driver Licence
    - Not valid Credentials
    - Not valid Code

- A list of both valid and invalid candidates course to test the Course Management Controller component. The set should contain instances exhibiting the following problems:
    - Null Objects
    - Null Fields

  - Not valid Recharger
  - Not valid Payment Data
  - Not valid Date
  - Not valid Time
  - Not valid Car Parameters
  - Not valid Client Parameters

- A list of both valid and invalid candidate parking station to test the Parking Station Controller component. The set should contain instances exhibiting the following problems:
  - Null Objects
  - Null Fields
  - Not valid Position
  - Not valid Code
  - Not valid Recharger

- A list of both valid and invalid candidate safe areas to test the Safe area Controller component. The set should contain instances exhibiting the following problems:
  - Null Objects
  - Null Fields
  - Not valid Position

- A list of both valid and invalid candidate reservations to test the Reservation Management Controller component. The set should contain instances exhibiting the following problems:
  - Null Objects
  - Null Fields
  - Not valid Client Parameters
  - Not valid Car Parameters
  - Not valid Code
  - Not valid Countdown

7 HOURS OF WORK

The writing of this document took overall about 10 hours of head-work.

The amount of work has been divided equally between the 3 members of the group. About 2 hours of 10 were spent working together in order to set up correctly and better coordinate each part of the work.
Furthermore, last 1 hours of head-work have been spent for rereading, revising and assembling the whole document.

8 USED TOOLS

The tools we used to create this document are:

- Microsoft Word 2016: to write and assemble the document
- Dropbox e GitHub: to share work
- Draw.io: for component diagrams