Power EnJoy

Integration **T**est **P**lan **D**ocument

Software Engineering 2

A.A. 2016/2017

Version 1.0

Emanuele Ghelfi (mat. 875550)
Emiliano Gagliardi (mat. 878539)

January 11, 2017

Contents

1 Introduction

1.1 Revision history

Version	Date	Authors	Description
1.0	/*TODO*/	Emanuele Ghelfi, Emiliano Gagliardi	Initial release

1.2 Purpose and scope

This document aims to the organization of the integration test for the PowerEnjoy project.
The main purpose of this document are:

- The decomposition of the components set defined in the Design Document in subsystems.
- Foreach subsystem, the definition of an integration strategy to be applied internally.
- The definition of an integration strategy between the subcomponents.

In the definition of an integration strategy, the most relevant points are:

- The entry criteria, that specifies the status that the coding phase have to reach before carry out the test.
- The order in which the subcomponents have to be integrated, directly followed by the definition of drivers and stubs if needed.

1.3 Definition, Acronyms, Abbreviations

- RASD: requirement analysis and specification document.
- DD: design document.
- ITPD: integration test plan document.
- API: application programming interface.
- IT: Integration test

1.4 Reference Documents

The reference documents used for the development of the integration test planning are:

- The assignment document
- The PowerEnjoy RASD
- The PowerEnjoy DD

2 Integration Strategy

2.1 Entry Criteria

Entry criteria are used to determine when integration testing activity should start. Criteria are necessary conditions on the project status in order to start integration testing.

The Entry Criteria for Power EnJoy integration test are the following:

1. Unit testing must have been performed
2. Environment for integration testing is complete.
3. All previous documents have been released (RASD and DD).
4. For each integration the sub-integrations must have been correctly tested.

Point 1 is because there is the need to be sure that sub-modules are internally correct before trying to integrate them. In this way bugs can only derive from interactions between sub-modules.

Point 2 is because before integration testing there is the need to have all required data source that will be find in the production environment. In this way testing will be efficient and effective. Integration testing need also Tools and Test Equipment specified in section 4.

Point 3 is because all the requirements of the software to be need to be specified (from RASD). Also modules and their interactions in order to fulfill requirements need to be stated (from DD).

Point 4 means that if a test involves the integration of component A with component B (when A depends on B), the last must have been correctly integrated with all its dependencies before starting the integration test. This is due to the bottom-up approach selected later (Section 2.3).

Before a specific integration test can begin the following things must have been delivered:

- Input Data that cover all test cases.
- Target Data (desired output for the Input Data).
- Drivers/stubs (if necessary).
- System status (involves DB status and components status).

2.2 Elements to be integrated

This subsection refers to component described in the DD and explain which components need to be integrated in order to have the complete Power EnJoy system.

In the system to be there are some components (high level components) that rely on lower lever components.

The latter are shared among different subsystems in a way to offer the same functionality to all components. The elements to be integrated in to ensure functionalities of our system are overlapping because of the software architecture of the system.

This is not a problem and the components with the highest fan-in of dependencies are the most critical of the system so they need to be tested with several components.

Here are listed all groups of components that need to be integrated with the related function:

Car Communication: includes Car Proxy, CarOS and Car Listener component of Ride Manager.

Registration Functionality: includes Registration Manager and Payment manager.

Reservation Functionality: includes the Reservation Manager and the Ride Manager.

Ride Functionality: includes Ride Manager, Payment Manager and Car Communication.

Search Functionality: includes Car Search Engine and Car Communication.

Car Management Functionality: includes Car Manager and Car Communication.

Notification Functionality: includes Notification Manager and Notification Receiver.

Monitoring Functionality: includes Car Monitoring, Car Communication and Notification Functionality.

Customer Communication: includes Customer Message Handler and Notification Functionality.

Other components are **Administrative Functionality Provider**, **Profile Manager** and **Authentication Manager**, these components do not depend on other components and for them integration it's only with respect to the entire system. They are autonomous and atomic components.

Notice that all components rely on the Data Model that relies on the DBMS API offered by the DBMS.

All components listed below needs to be integrated together in order to obtain the **Application Server** Subsystem.

The last step is to integrate all client components (Mobile and Web App) together with the Application Server.

After all the integrations the system is complete.

2.3 Integration Testing Strategy

The integration testing strategy used in this project is a mixture of critical-module-first and bottom-up.

Integration testing and build plan are strictly related so the development should also follow these approaches.

Critical-Module-First Rationale

Critical Module First is the strategy used to support the integration of higher level subsystems . Since higher level subsystem are independent one from the other there isn't an established precedence ordering. Critical Module First has been selected because in this way riskiest component are developed first and tested first. Riskiest components are components with important or complex functionalities and components with the highest fan-in. An error or a malfunction in these components will compromise all the software to be. These malfunctions need to be discovered (and corrected) as soon as possible so these components have to be tested a lot.

Bottom-up Rationale

Bottom Up approach is used internally with respect to subsystems. In this way the the integration proceed from lower level subcomponents. By doing this there is the possibility to test component's behavior in early stage of development and improve parallelism and efficiency. Bottom-up approach requires the development of drivers for higher level components not yet developed.

The decision to write drivers instead of stub is because, in practical scenarios, behavior of stubs is not that simple as it seems. The called module, most of the time involves complex business logic like connecting to a database. As a result creating Stubs becomes as complex and time taking as the real module. In some cases, Stub module may turn out to be bigger than stimulated module.

Drivers, instead, are the dummy programs which are used to call the functions of the lowest module in case when the calling function does not exists.

2.4 Sequence of Component/Function Integration

This section is strictly related to the integration testing strategy described in section 2.3. In this section is explained the sequence in which components need to be integrated and tested.

Since it's used a bottom up approach to integrate subsystems, it's important to define a precedence relationship between components. In this section a dotted arrow from component C to component C' means that component C depends on component C'. This means that component C' needs to be developed and integrated before C in a bottom-up approach.

2.4.1 Software Integration Sequence

This subsection explains how subcomponents are integrated together in order to obtain higher level components defined in section 2.2.

Data Model

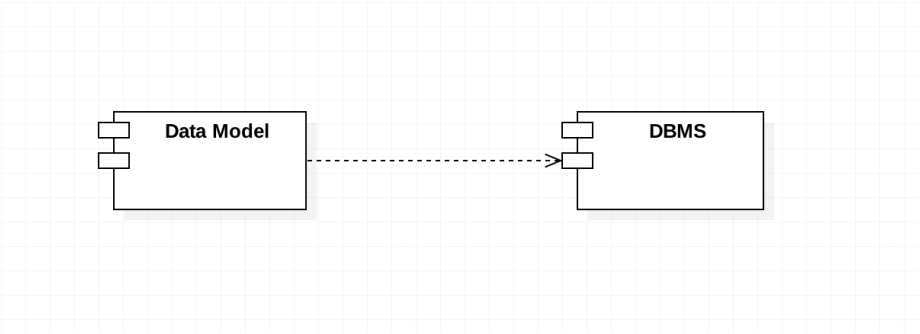

Figure 1: Data Model

The Data Model depends on the DBMS and its schema. This component is one of the most important component of the software architecture since all other components rely on it in order to perform their tasks. . So the choice of starting from this component comes from the critical module first approach.

The Data model needs to use in a proper way the APIs provided by DBMS. The DBMS schema is designed in parallel with respect to the data model since they are highly inter-connected.

Car Communication

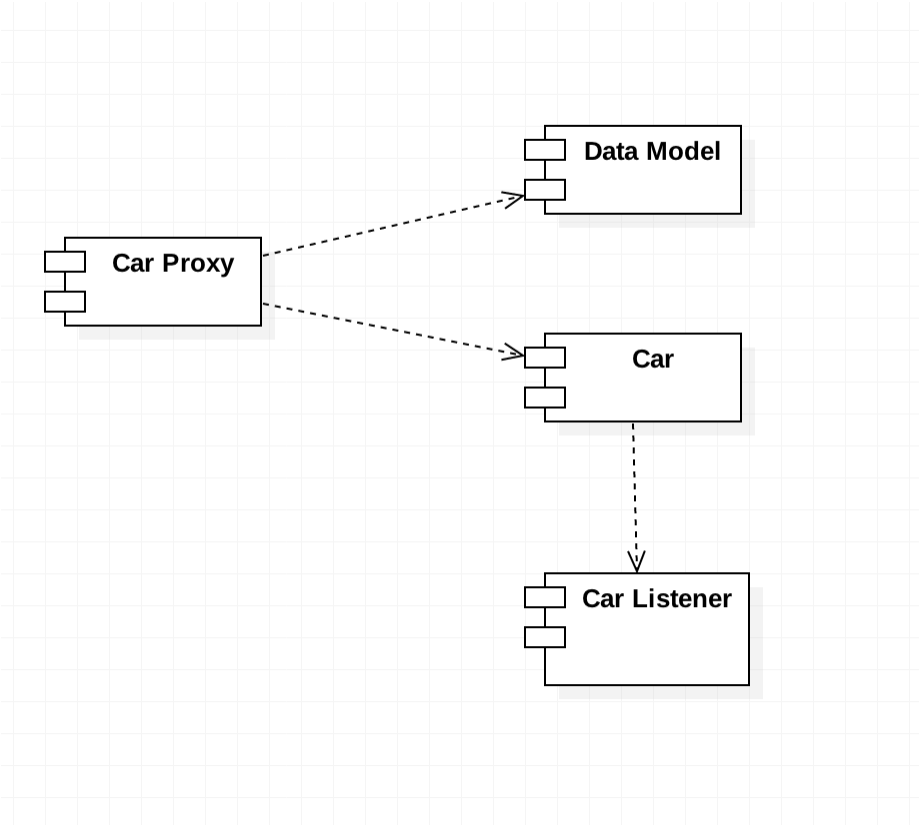

Figure 2: Car Communication

Car Communication is the most critical module, almost all components rely on it. Car proxy relies on the Data Model so the Data Model is the first component that needs to be developed (as before). The CarOS component depends on the Car Listener that is a subcomponent of the Ride Manager but for now it's a stub because the ride manager will be developed later. This is an exception of the bottom up approach but Car Listener at this moment is useless for the integration because it receives only callback from the car event.

Test Case: IT1 in Section 3.1

Registration Functionality

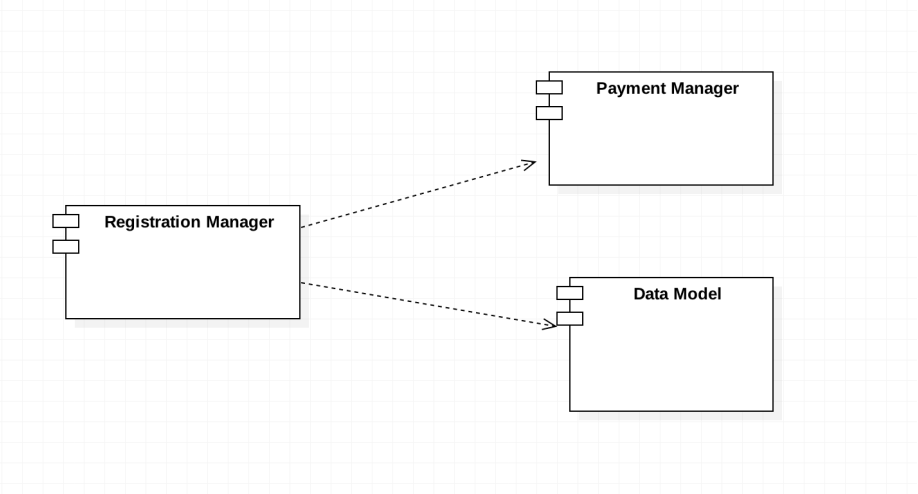

Figure 3: Registration Functionality

The Registration Functionality is a subcomponent independent from the previous subcomponents so it can be developed and integrated in parallel with respect to them.

Test Case: IT2 in Section 3.2

Ride Functionality

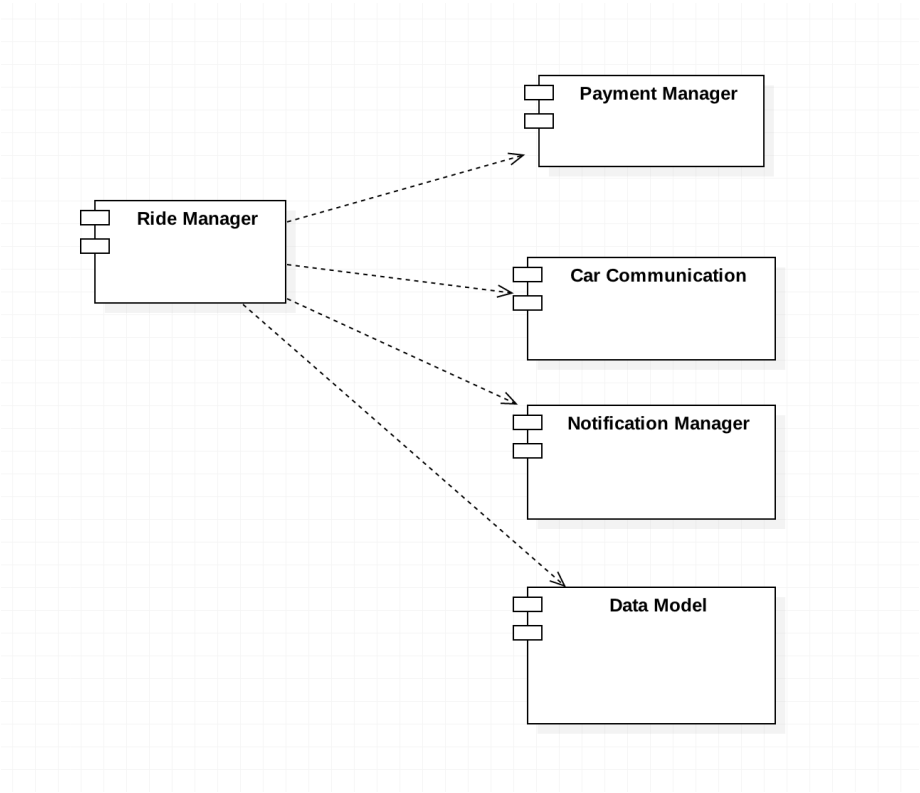

Figure 4: Ride Functionality

Ride Functionality is the subcomponent that manages all the flow of the ride. It needs to be integrated with Payment Manager, Car Communication, Notification Manager and, of course, with the Data Model. In this integration there is the necessity to test the payment part, the unlock and lock calls and the notification to the user. At this level of integration the Notification Manager could be a stub since it's not an important component in order to test the ride flow.

Test Case: IT3 in Section 3.3

Reservation Functionality

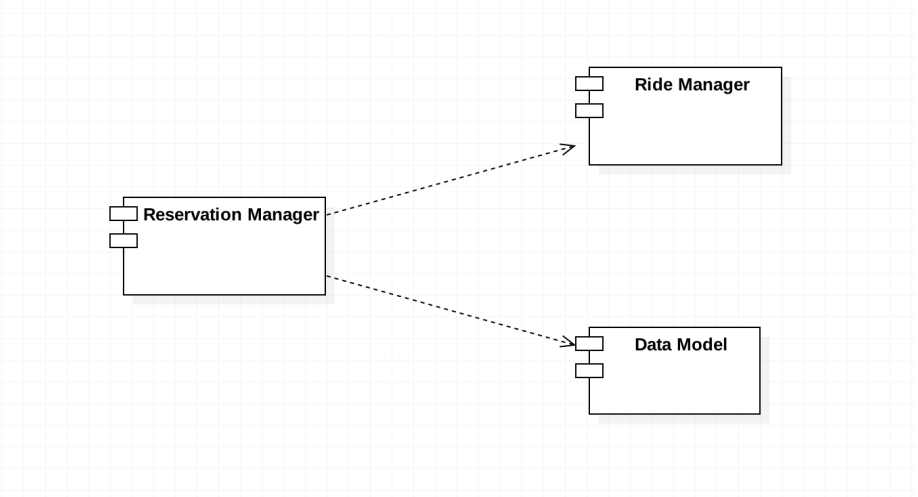

Figure 5: Reservation Functionality

The Reservation Manager and the Ride Manager needs to be integrated together because of the dependency between them.

Test Case: IT4 in Section 3.4

Search Functionality

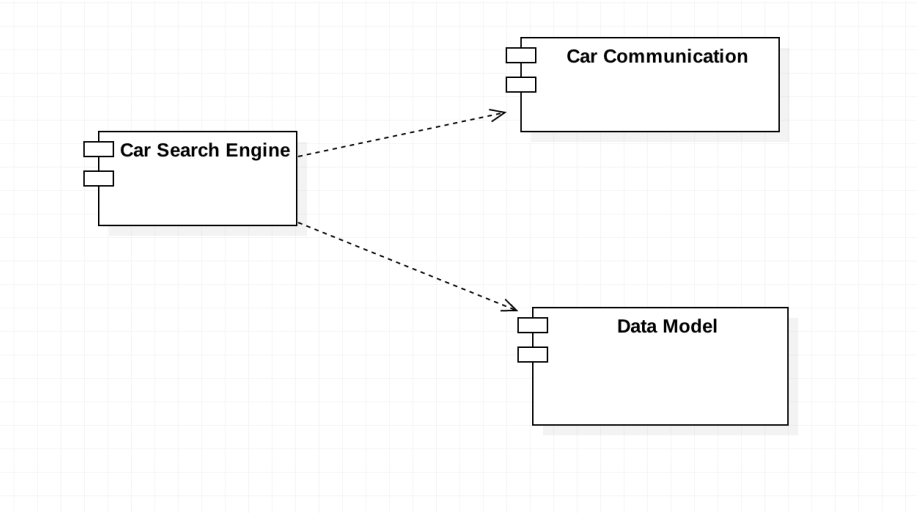

Figure 6: Search Functionality

In order to test the search functionality the Car Search Engine needs to be integrated with the Car Communication subcomponent.

Test Case: IT5 in Section 3.5

Notification Functionality

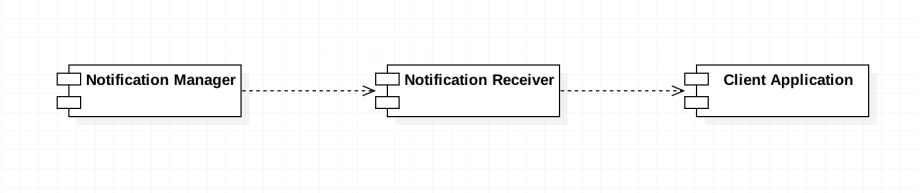

Figure 7: Notification Functionality

At this point all critical subcomponents have been integrated. The integration can proceed with the Notification Functionality. Since the client application could be not yet ready it can be a stub. The important test here is that the notification are correctly received by the Notification Receiver and that the response is received correctly by the Notification Manager.

Customer Communication

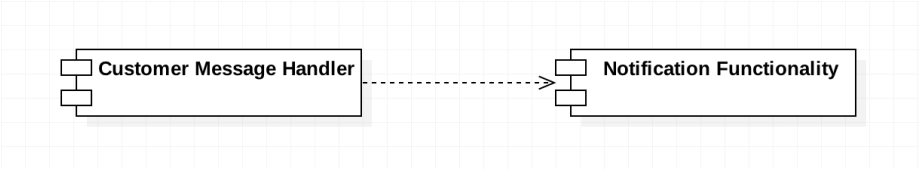

Figure 8: Customer Communication

The Customer Message Handler component depends only on Notification Functionality and no other component depend on it. Like Monitoring Functionality component it can be integrated lastly following the critical Module First Approach.

Test Case: IT6 in Section 3.6

Monitoring Functionality

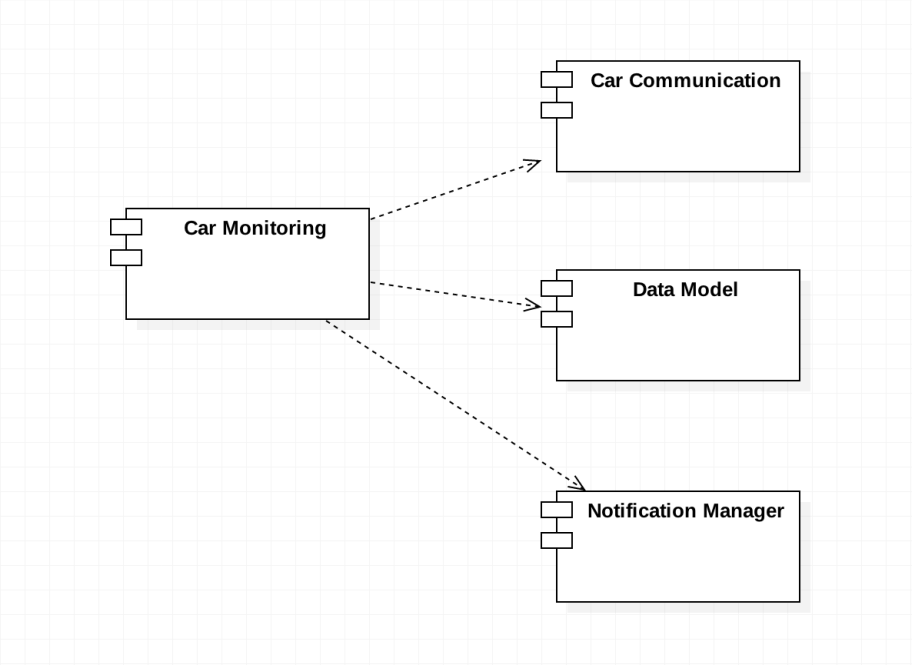

Figure 9: Monitoring Functionality

No components depends on the Car Monitoring component so it can be integrated lastly.

Test Case: IT7 in Section 3.7

Car Management Functionality

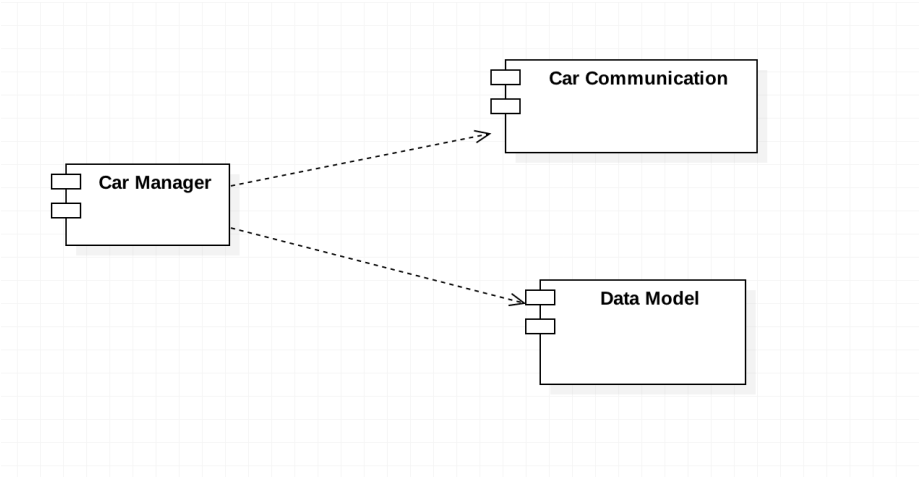

Figure 10: Car Management Functionality

Car management functionality is a backend-component necessary for administrators and operators, it's independent for the user-part of the system.

Test Case: IT8 in Section 3.8

Other Integrations

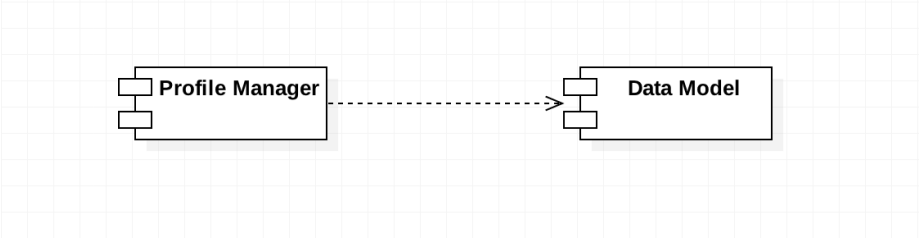

Figure 11: Profile Manager

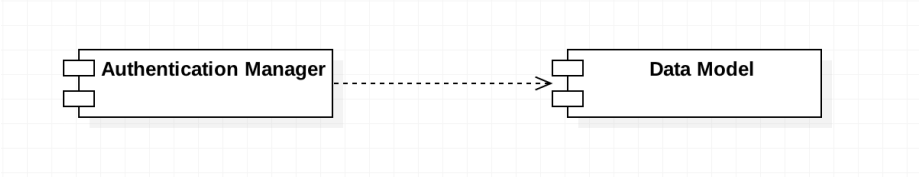

Figure 12: Authentication Manager

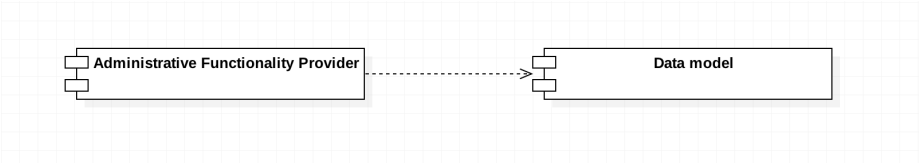

Figure 13: Administrative Functionality Provider

2.4.2 Subsystem Integration Sequence

In this part for simplicity the server part is divided into three main components: the User-part, the Administrative-part and the Operator-part.

The components share some subcomponents between them.

The **User − part** is made of the following components:

- Ride Functionality
- Registration Functionality
- Reservation Functionality
- Search Functionality
- Notification Functionality
- Customer Communication
- Profile Manager
- Authentication Manager

The **Administrative − part** of the system is made of the following components:

- Administrative Functionality Provider
- Car Manager Functionality
- Monitoring Functionality
- Authentication Manager

The **Operator − part** of the system is made the following components:

- Car Manager Functionality
- Monitoring Functionality
- Notification Functionality
- Search Functionality
- Authentication Manager

At this point the client applications of the Users, Operators and Administrators need to be integrated with the relative parts of the system.

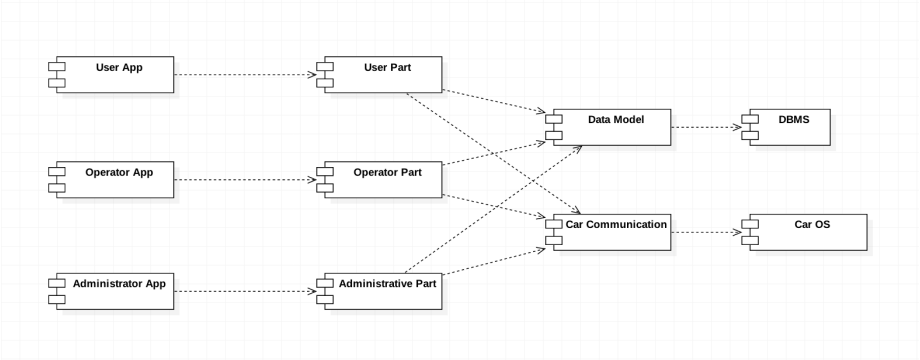

Figure 14: Subsystems Integration

3 Individual steps and test description

In this section are described the tests that have to be performed to integrate component belonging to the same subsystem. Each step is characterized by the caller and the called component, and by all the methods that are invoked by the caller on the called. Foreach method are defined some representative inputs and the corresponding expected effects.

3.1 Car communication subsystem (IT1)

3.1.1 Car uses the CarlListener component callback methods

startEngineCallback(carId)	
INPUT	EFFECT
A null parameter	A null value exception is raised
An invalid parameter	An invalid parameter exception is raised
A valid parameter	The ride manager is notified that the engine of the given car started

stopEngineCallback(carId)	
INPUT	EFFECT
A null parameter	A null value exception is raised
An invalid parameter	An invalid parameter exception is raised
A valid parameter	The ride manager is notified that the engine of the given car stopped

An invalid parameter could be:

- A carId that doesn't exist
- An empty carId
- A carId of a car that is under maintenance
- A carId of a car that is not in a ride
- A carId of a car that have not yet called startEngineCallback

3.1.2 Call to car proxy from another subsystem

getCarStatus(carId)	
INPUT	EFFECT
A null parameter	A null value exception is raised
An invalid parameter	An invalid parameter exception is raised
The id of a car that have not been asked recently	The car API are correctly called
The id of a car that have been asked recently	The car API are not called, instead the db is interrogated

unlockCar(carId)	
INPUT	EFFECT
A null parameter	A null value exception is raised
An invalid parameter	An invalid parameter exception is raised
A valid carId	The car API are correctly called

lockCar(carId)	
INPUT	EFFECT
A null parameter	A null value exception is raised
An invalid parameter	An invalid parameter exception is raised
A valid carId	The car API are correctly called

An invalid parameter could be:

- An empty carId
- A carId that doesn't exist

3.2 Registration functionality subsystem (IT2)

3.2.1 Registration manager uses Payment manager

verifyValidity (paymentInformation)	
INPUT	EFFECT
A null parameter	A null value exception is raised
A not valid parameter	A non valid input exception is raised
A valid parameter, that doesn't correspond to a real account	False is returned
A valid parameter	True is returned

A not valid paymentInformation could have some field with the wrong format (number of character, or doesn't satisfies a regular expression), or an empty field.

3.2.2 External call to Registration Manager

doRegistration(username, password, paymentInformation, driveLicense)	
INPUT	EFFECT
One, some, or all null parameter	A null value exception is raised, and it is reported in the response
One, some, or all not valid parameter	An invalid parameter exception is raised, and it is reported in the response
A username that is already present in the database	The response reports that the username is already in use
All well formed, but the paymentInformation doesn't correspond to a real account	The payment manager methods are correctly called, and the response reports that the paymentInformation are not accepted
All valid, but the DriveLicense doesn't correspond to a real account	The response reports that the driveLicense is not accepted
Username that doesn't exist yet, valid password, paymentInformation and driveLicense	The response confirm the validity, the database is updated with the information

A not valid list of parameter could contain:

- An empty username
- An empty password
- A password that doesn't satisfies the general criteria of passwords (length, contains certain character...)
- A drive license that is not valid, like with an empty field or a field with wrong format (number of character, or doesn't satisfies a regular expression)
- Not well formed payment information

3.3 Ride functionality subsystem (IT3)

3.3.1 Ride manager uses Payment manger

doPayment(paymentInformation, price)	
INPUT	EFFECT
One or both null parameter	A null value exception is raised
One or both invalid parameter	A not valid input exception is raised
Valid paymentInformation and price	The payment is carried out

An invalid list of parameter could contain:

- Not well formed payment information (what does not well formed paymentInformation means is defined in 2.1.1 section)
- PaymentInformation that doesn't correspond to a real account
- Negative price

3.3.2 External call to Ride manager

unlockRequest(userId, carId, userPosition)	
INPUT	EFFECT
One, some, or all null parameter	A null value exception is raised, and it is reported in the response
One, some, or all invalid parameter	A not valid parameter exception is raised, and it is reported in the response
Valid userId, carId, but a position that is distant from the car position	The unlock method of car proxy is not called, the response reports that the car can't be unlock
Valid userId, carId, and a position that is near to the car position	The unlock method of car proxy is correctly called, and if it terminates correctly the response confirm, otherwise the response reports that it is not possible to unlock the car

An invalid list of parameter could contain

- An empty userId
- An empty carId
- A userId that doesn't exist
- A carId that doesn't exist

- UserId and carId that are not in the same ride
- Position in wrong format

lockRequest(userId, carId)	
INPUT	EFFECT
One, some, or all null parameter	A null value exception is raised, and it is reported in the response
One, or both invalid parameter	A not valid parameter exception is raised, and it is reported in the response
Valid userId and carId	The lock method of car proxy is correctly called, and if it terminates correctly the response confirm, otherwise the response reports that it is not possible to lock the car

An invalid list of parameter could contain:

- An empty userId
- An empty carId
- A userId that doesn't exist
- A carId that doesn't exist
- UserId and carId that are not in the same ride

pauseRide(ride)	
INPUT	EFFECT
A null parameter	A null value exception is raised, and it is reported in the response
An invalid parameter	A not valid parameter exception is raised, and it is reported in the response
A valid parameter	The ride state is set to pause, a new car listener is correctly initialized and the response confirm the success of the operation

An invalid parameter could be

- A ride that doesn't exist
- A ride with the engine that is running

- Like said in 1.3.1 section

terminateRide(ride)	
INPUT	EFFECT
A null parameter	A null value exception is raised, and it is reported in the response
An invalid parameter	A not valid parameter exception is raised, and it is reported in the response
A valid parameter	The procedure to compute the cost starts and the payment component methods are correctly called. The response contains the result of the payment.

An invalid parameter is the same as an invalid parameter for pauseRide

3.4 Reservation functionality subsystem (IT4)

3.4.1 Reservation manager uses Ride manager

rideInit(ride)	
INPUT	EFFECT
A null parameter	A null value exception is raised
An invalid parameter*	A not valid input exception is raised
A valid ride parameter	A CarListener instance is correctly created, and the car os is informed that the server is listening for events. The database is updated with a new ride instance

An invalid ride could contain:

- A user that doesn't exist
- A car that doesn't exist
- A car that hasn't been reserved by the user
- A car that has been reserved by another user
- A car that is under maintenance
- A date or time that is in the past

3.4.2 External call to Reservation manager

doReservation(userId, reservationInfo)	
INPUT	EFFECT
One or both null parameter	A null value exception is raised, and it is reported in the response
One or both invalid parameter	A not valid input exception is raised, and it is reported in the response
A valid userId, reservationInfo contains a car that have been already reserved	The response reports that the car is already reserved
A valid userId, reservationInfo contains a car that is under maintenance	The response reports that it is not possible to reserve that car
A valid userId, valid reservationInfo	rideInit is called and terminates correctly, the database is updated with a new reservation instance

An invalid list of parameter could contain:

- An empty userId
- A userId that doesn't exist
- Invalid reservationInfo (date or time in the past, car that doesn't exist)

3.5 Search functionality subsystem (IT5)

3.5.1 External call to car search engine

searchCars(position, radius)	
INPUT	EFFECT
A null parameter	A null value exception is raised, and it is reported in the response
An invalid parameter	A not valid parameter exception is raised, and it is reported in the response
A valid list of parameter	The car proxy is correctly called, and the correct list of car is returned

An invalid list of parameter could contain:

- Wrong format position
- Negative radius

3.6 Customer communication subsystem (IT6)

3.6.1 Call to NotificationManager from another subsystem

sendNotification(notificationInfo, notificationType, receiver)	
INPUT	EFFECT
A null parameter	A null value exception is raised, and it is reported in the response
An invalid parameter	A not valid parameter exception is raised, and it is reported in the response
A valid list of parameter	The notification is correctly sent to the client

This call should be tested with all type of client.
A list of invalid parameter could contain:

- Invalid notificationType
- Empty notificationInfo
- A receiver that does not exist

3.7 Car monitoring subsystem (IT7)

This subsystem doesn't expose methods, it works like a daemon and uses car proxy and notification manager components. So are defined some events that could happen, and the expected behavior of the car monitoring component.

A car needs an operator intervention	After an acceptable time interval, the car monitoring component asks to car proxy the car status, detects the malfunction, and correctly calls the notification manager

3.8 Car management subsystem (IT8)

3.8.1 External call to car manager

setCarStatus(carId, status)	
INPUT	EFFECT
One or both null parameter	A null value exception is raised, and it is reported in the response
One or both invalid parameter	A not valid parameter exception is raised, and it is reported in the response
CarId corresponds to a car that is reserved, status is set to under maintenance	The status is correctly modified in the database, the user that made the reservation is find and the notification manager is called
CarId corresponds to a car that is under maintenance, status is set to available	The status is correctly modified in the database

An invalid list of parameter could contain:

- An empty carId
- A carId that doesn't exist
- A not valid status (the valid status are listed in the RASD document)
- A non compatible car status changing

getCarStatus(carId)	
INPUT	EFFECT
A null parameter	A null value exception is raised, and it is reported in the response
An invalid parameter	A not valid parameter exception is raised, and it is reported in the response
A valid parameter	The car status is returned

unlockCar(carId)	
INPUT	EFFECT
A null parameter	A null value exception is raised, and it is reported in the response
An invalid parameter	A not valid parameter exception is raised, and it is reported in the response
A valid parameter	The car proxy is correctly called

lockCar(carId)	
INPUT	EFFECT
A null parameter	A null value exception is raised, and it is reported in the response
An invalid parameter	A not valid parameter exception is raised, and it is reported in the response
A valid parameter	The car proxy is correctly called

An invalid parameter could be:

- An empty carId
- A carId that doesn't exist

4 Tools and Test Equipment Required

In this section are specified all tools and test equipment required in order to perform integration testing of the previous sections.

4.1 Testing Tools

Testing tools are required in order to perform in an effective way tests described before. Since for the server part JEE technology has been used, there is the need of testing framework compatible with that technology.

The first testing tool needed is **Arquillian**. Arquillian makes easy to create integration testing.

In the integration testing phase Arquillian is used for:

- Managing the lifecycle of the container (or containers)
- Bundling the test case, dependent classes and resources into a ShrinkWrap archive (or archives)
- Deploying the archive (or archives) to the container (or containers)
- Enriching the test case by providing dependency injection and other declarative services
- Executing the tests inside (or against) the container
- Capturing the results and returning them to the test runner for reporting

With Arquillian integration phase is accelerated.

Although the integration strategy is bottom up, sometimes there is the need of stub. In order to create stubs we need something like **Mockito**.

Mockito allows to:

- Abstract dependencies
- Have predictable results
- Check the interaction between the caller and the mocked object.

In order to do be more effective **JUnit** framework will be used. JUnit is a framework used mainly for Unit Testing but it can be also used for integration testing (for example to test if the interaction between components is happing in the correct way). There is the possibility to test if the correct parameter are passed to certain methods, if the method returns the expected result or raises the excepted exceptions.

Another testing tool that could be useful is **JUnitEE** . This framework extends the standard JUnit so that it can execute unit tests in an application server container. It is configured in the J2EE Web module of a unit test application, and it uses a TestRunner to output HTML or XML test results. It also includes a TestServlet for an entry point to JUnit test cases. Building your test harness as a standard J2EE Web offers several benefits:

- Tests are packaged in a J2EE Web module (in a WAR file), which is easy to deploy and execute.
- Test cases look just like production code, and they can use the same Java beans that you use as a facade for your EJBs.
- Tests can be automated by using an **Ant** script.

In order to perform some performance test **Apache JMeter** will be used. It can be used to simulate a heavy load on a server, network or object to test its strength or to analyze overall performance under different load types. In our software JMeter can be useful also for parametric and automated test on Authentication Manager and for testing the performance of the Car Proxy component.

Since the software to be will be running on different platforms, in order to test the client part are necessary all debuggers and testing tool related to the target OS (Windows Phone, Android, iOS).

Notice that to perform the tests described in the previous sections some **manual testing** will be necessary. This is intended in term of preparing correct test data for every test case.

As a summary, the following testing tools are required:

- Arquillian
- Mockito
- JUnit
- JUnitEE
- JMeter

4.2 Test Equipment

Some test equipment are required to test the software to be. Test equipment is divided among the different part of the system they are needed.

Mobile Side

Since the application will be running on different platforms there is the necessity to ensure the correct behavior on them.

For testing the client side of the software the following devices are required:

- Some Android smartphones and tablets with several screen sizes and resolution.
- Some WP smartphones with several screen sizes and resolutions.
- One iOS smartphone for each generation.
- One iOS tablet for each generation.

Web Side

The web application will be tested using different notebooks and desktop computers. Like what concern the mobile side, the computers need to have different screen size and different resolutions.

The web application need to be tested with different browsers (e.g. Safari, Google Chrome, Firefox, Opera, Microsoft Edge). It's important also to test the web side on the mobile devices listed above.

Car Side

The most critical part of the system regards the interaction with the car.

It could be useful (but maybe not feasible) for testing purposes to have a board that simulates the car with carOS installed on it. This could be used to test the system interaction with the carOS in an easier way with respect to test it on the car itself. This must be discussed with the company of the carOS.

Also a software that simulates the car and his OS is needed to make testing simpler.

Notice that the carOS won't be tested because there is the assumption that is a commercial system already present on the market. What will be tested will be the interaction between the software to be and that system.

Server Side

The testing environment needs to be very similar to the production environment to minimize the possibility of production error. The DBMS and the Application Server will be the same as the production.

Also the OS under the Application Server needs to be the same as the OS of the production environment.

For the sake of limit testing cost the testing version of the environment could be a scaled down version of the production environment.

5 Program Stubs And Test Data Required

5.1 Program stubs and driver

Here are listed for each subsystem the stubs and the drivers required in order to perform the tests as soon as possible during the development. The drivers and the stubs are only listed because the method that have to be called by each driver, or exposed by each stub, can be simply deducted from the section 3.

Data model drivers

The only driver required in this integration is DataModelCaller driver, that simulates the call from other subsystems to the DataModel component.

Car communication

This is the exception of the bottom-up method, infact the car uses the CarListener component, but it is a part of the RideComponent that have to be developed and tested later than the car communication subsystem. So there will be the need of a CarListener stub.
The needed driver are the following:

- CarProxy driver
- CarProxyCaller driver that simulates the calls to the car proxy from a generic other subsystem

Ride functionality

In section 2 is specified that the NotificationManager component at this level could be a stub, so there is the need of a NotificationManager stub.
The needed driver are the following:

- RideManager driver
- RideManagerCaller driver that simulates the calls to the RideManager from a generic client.

Registration functionality

The needed driver are the following:

- RegistrationManager driver
- RegistrationManagerCaller driver that simulates the calls to the RegistrationManager from a generic client.

Reservation functionality

The needed driver are the following:

- ReservationManager driver
- ReservationManagerCaller driver that simulates the calls to the ReservationManager from a generic client.

Search functionality

The needed driver are the following:

- SearchEngine driver
- SearchEngineCaller driver that simulates the calls to the SearchEngine from a generic client.

Notification Functionality

In this integration the client application could be a stub, so there is the need of a ClientApplication stub that expose the methods that the NotificationReceiver could call.
The needed driver are the following:

- NotificationManager driver
- NotificationManagerCaller driver that simulates the calls to the NotificationManager from a generic other subsystem.

Monitoring Functionality

The only needed driver in this subsystem integration is the CarMonitoring component driver.

Customer Communication

The needed driver are the following:

- CustomerMessageHandler driver
- CustomerMessageHandlerCaller driver that simulates the calls to the CustomerMessageHandler from a generic client.

Car Management Functionality

The needed driver are the following:

- CarManager driver
- CarManagerCaller driver that simulates the calls to the CarManager from a generic client.

5.2 Test Data Required

The definition of the test data have been already done in the section 3.

6 Appendix

6.1 Hours of Work

Emanuele Ghelfi:

- 24/11/16: 4 h, Overview of the problem
- 29/11/16 5 h, Focus on architecture
- 30/11/16: 5 h, Diagrams
- 1/12/16: 1 h, introduction
- 2/12/16: 3 h, Architecture Overview, diagrams, algorithms, data model
- 3/12/16: 3 h, Comments on architecture, requirements traceability
- 6/12/16: 4 h, Components and interfaces
- 7/12/16: 3 h, Algorithms and refactoring

Total hours: 28 h

Emiliano Gagliardi:

- 24/11/16: 4 h, Overview of the problem
- 29/11/16 5 h, Focus on architecture
- 30/11/16: 5 h, Diagrams
- 1/12/16: 1 h, introduction
- 2/12/16: 3 h, Architecture Overview, diagrams, algorithms, data model
- 3/12/16: 3 h, Comments on architecture, requirements traceability
- 6/12/16: 4 h, Components and interfaces
- 7/12/16: 3 h, Algorithms and refactoring

Total hours: 28 h

6.2 Used Tools

The tools used to create this RASD document are:

- Github: for version control.
- Lyx: to redact and organize this document.
- StarUML: to create UML diagrams (component diagram, data model diagram, deployment diagram, sequence diagram, state-chart diagram, use case diagrams).

References

[1] http://jmeter.apache.org/

[2] http://arquillian.org/

[3] http://site.mockito.org/

[4] http://junit.org/junit4/

[5] http://www.softwaretestinghelp.com/what-is-integration-testing/

[6] http://softwaretestingfundamentals.com/integration-testing/

[7] http://www.stackoverflow.com