POLITECNICO
MILANO 1863

AA 2016-2017

Software Engineering 2 project: PowerEnJoy
Integration Test Plan Document (ITPD)

Matteo Franchi - 807046
Luca Guida - 878001
Alessandro Lavelli - 875788

Document version 1.0

Contents

1 Introduction

1.1 Revision History

- Version 1.0 (13 Jan 2017)

1.2 Purpose and Scope

This document is the Integration Testing Plan Document for *PowerEnJoy Platform.* Integration testing is essential in order to guarantee that all the different subsystems function according to the requirements laid out in the RASD and without exhibiting unwanted behaviors. The purpose of this document is to describe the preparation of the integration testing activity for all the components, which as a whole build up the product. In the following sections we're going to provide:

- A list of all the components and their subsystems involved in the integration testing activity
- The entry criteria that must be met by the project status in order to properly begin the integration testing
- A description of the chosen integration testing strategy and the reasons that led to its adoption
- The sequence with which the components and their subsystems will be integrated
- A description of the planned testing activities for each integration step, including all the input data and the expected outcome
- A list of all the tools required during the testing activities and a description of the starting environment in which the tests will be executed

1.3 List of Definitions and Abbreviations

- **DD**: Design Document
- **RASD**: Requirements Analysis and Specifications Document

1.4 List of Reference Documents

- *Project goal, schedule, and rules (Assignments AA 2016-2017.pdf)*
- *Requirements Analysis and Specification Document*, version 1.2
- *Design Document*, version 1.1
- *JUnit Wiki* (junit.org/junit4)

- *Mockito documentation* (site.mockito.org)
- *Arquillian Guides* (arquillian.org/guides)
- *Apache JMeter User's Manual* (jmeter.apache.org/usermanual)
- *Android developers - Manage Your App's Memory* (developer.android.com/topic/performance/memory.html)
- *Android developers - Optimizing for Battery Life* (developer.android.com/topic/performance/power/index.html)
- *Apple developers - Performance Tools* (developer.apple.com/library/content/documentation/Performance/Conceptual/PerformanceOverview/PerformanceTools/PerformanceTools.html)
- *Apple developers - Measure Energy Impact* (developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/MeasuringEnergyImpact.html)
- *Android developers - Run Apps on the Android Emulator* (developer.android.com/studio/run/emulator.html)
- *Apple developers - Simulator User Guide* (developer.apple.com/library/content/documentation/IDEs/Conceptual/iOS_Simulator_Guide/Introduction/Introduction.html)
- *Microsoft developer resources - Test with the Microsoft Emulator for Windows 10 Mobile* (msdn.microsoft.com/en-us/windows/uwp/debug-test-perf/test-with-the-emulator)

2 Integration Strategy

2.1 Entry Criteria

Integration testing should start as soon as the the main components of the system are released, but some preconditions should be satisfied before beginning the verification activity.

- First of all, the *Requirements Analysis and Specification Document* and the *Design Document* should be complete and available to all the people involved in the project in order to inform them about requirements and adopted design choices.

- In order to test specific features of the system components, some low level modules and external APIs should be available (i.e., access to APIs should be granted by the respective API providers after subscribing proper usage plans). In particular,
  - the DBMS should be configured and operative in order to allow to test all the components which need access to the databases,
  - for integration tests involving the *Rental Module*, and in particular rental fees charging features, the external *Payment Gateway* should be already configured and fully operative,
  - for integration tests involving the *Account Manager* component, and in particular the user registration and activation features, the external *SMS Gateway* should be already available and ready to be used,
  - for integration tests involving the *SOS Call Module*, the *Emergency Service Manager* component (an external VoIP API) should be fully implemented,
  - for integration tests involving maps and navigation components, like the ones of the customer mobile application, car on-board tablet app or employee web app, the *Maps API* should be available and fully usable.

- In order to test the integration of two components, the main features of both of them should have been developed and the related unit tests should have been performed. To be more specific, minimum percentages of completion of each component with respect to its functionalities should have been reached:
  - 60% of the *Feedback Module*,
  - 60% of the *Rental Module* (at least car reservation and booking cancellation features),
  - 60% of the *Account Manager* (at least user creation and login features),
  - 60% of the *Navigation Controller* (at least start/end rental features),

  - 80% of the *Car Controller*,
  - 80% of the *SOS Call Module*,
  - 60% of the *Fleet Manager* (at least change car state features),
  - 70% of the *Field Service Module*.

  The different percentage values reflect the minimum number of features which need to be implemented in order to perform meaningful integration tests.

2.2 Elements to be Integrated

Referring to the *design document,* the system is composed of several components. They can be divided into three categories:

- Front-end components: mobile application, web application, telematic control unit and on-board tablet application
- Back-end components: Employee web services, Customer mobile services and Car services
- External components: all the components which refer to functionalities provided by external systems and the DBMS

All the components belonging to Front-end and External categories are independent one from each other. In the Back-end category, instead, there are few components which are not independent with the others of the same category. Due to this architecture, the integration tests could be performed without a subsystem being entirely tested.

There are two types of partial integration: back-end with external components and front-end with back-end. After the partial integration is completed it will be possible to integrate together the components of the three categories.

The main integrations of back-end components with external components are:

- *Customer mobile services subsystem:*
  - **Feedback module, DMBS**
  - **Rental module, DBMS, Maps API, Payment Gateway, Car Controller**
  - **Account Manager, DBMS, Payment Gateway, SMS Gateway**
- *Car services subsystem:*
  - **Navigation controller, Maps API, DBMS, Rental Module**
  - **Car controller, DBMS**
  - **SOS call module, DBMS, Emergency Service Manager**

    - **Car controller, Navigation Controller, SOS Call Module**

- *Employee web services subsystem:*
    - **Fleet Manager**, **Maps API**, **DBMS, Emergency Service Manager**
    - **Field service module**, **Maps API**, **DBMS**

The main integrations of front-end components with back-end components are:

- **Web application, Employee web services**
- **Customer mobile application, Customer mobile services**
- **Telematic control unit, Car services**
- **On-board tablet application, Car services**

2.3 Integration Testing Strategy

Given that the entire development effort mainly follows a bottom-up strategy, we opted to use a similar approach with respect to the testing phase: we will start by integrating the components that do not depend on other ones in order to function properly, or that only depend on already developed components (which also include third-party software and hardware), then, once all the single components are fully tested, we will finish by testing the subsystems which the previous components take part of. This allows us to begin to test a component or a subsystem as soon as it's finished (or sometimes near completion), which in turn gives us immediate feedback and allows us to (partially) parallelize the development phase and the testing phase.
Moreover, following the critical-module-first approach, we will start by focusing on the *Car Controller* component since it's the most critical one, so that we will be able to fix any bug, which could lead to dangerous consequences (especially for the customer), as soon as they appear.

2.4 Sequence of Component/Function Integration

In this section of the document the order of integration of the components and subsystems of *PowerEnJoy Platform* will be described.
As a notation, an arrow going from component A to component B means that A is necessary for B to function, so it must have already been implemented before performing the integration.

2.4.1 Software Integration sequence

Customer mobile services The three main components of the Customer mobile services subsystem all require the *DBMS* in order to function properly. Moreover, *Account Manager* requires *Payment Gateway* and *SMS Gateway*, finally *Rental Module* requires *Maps API*, *Payment Gateway* and *Car Controller* (the latter can be found in the *Car services* subsystem).

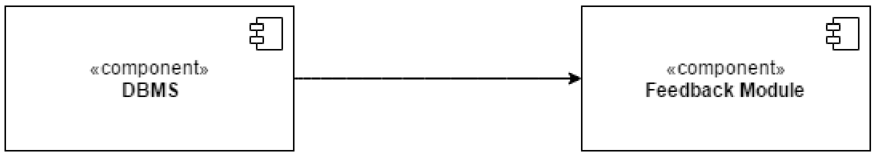

Figure 1: I1.1 - Feedback Module

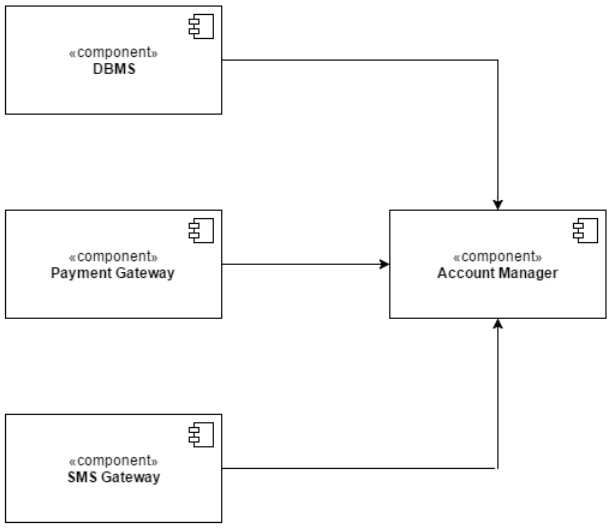

Figure 2: I1.2 - Account Manager

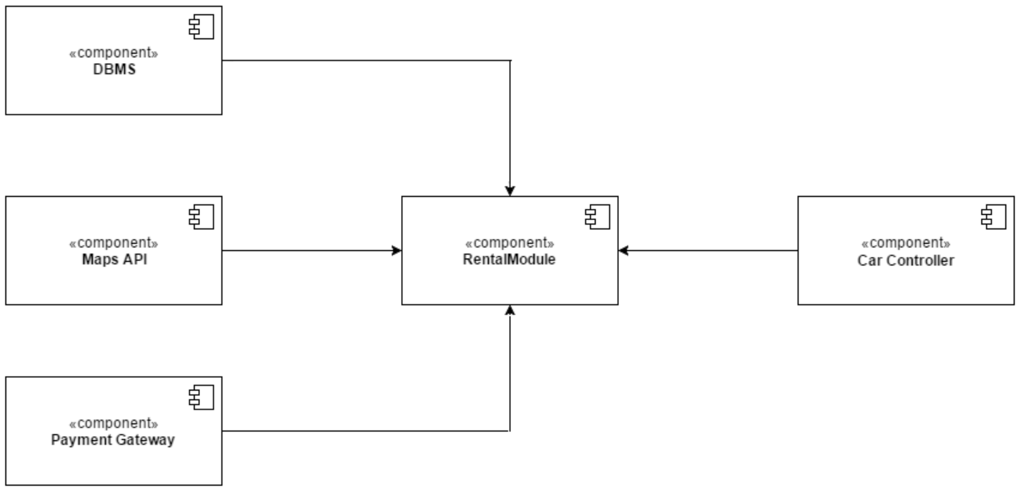

Figure 3: I1.3 - Rental Module

Car services All the three components require the *DBMS* module to work properly; in addition, *Navigation Controller* requires *Car Controller, Rental Module* and *Maps API*, while *SOS Call Module* requires *Emergency Service Manager.*

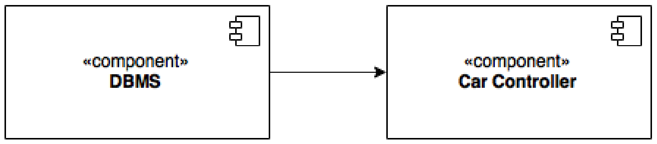

Figure 4: I2.2 - Car Controller

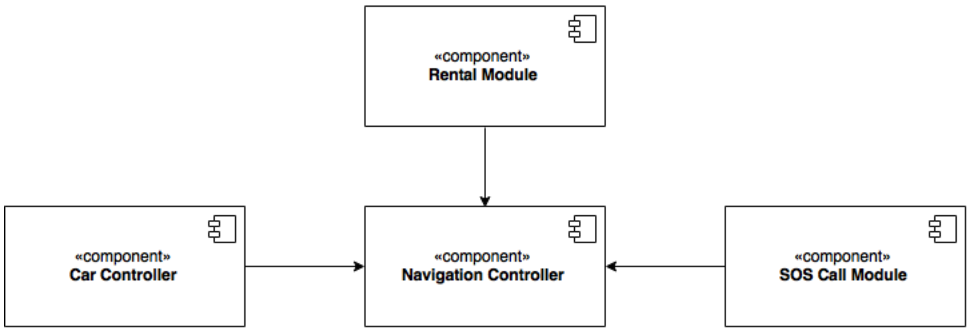

Figure 5: I2.1 - Navigation Controller

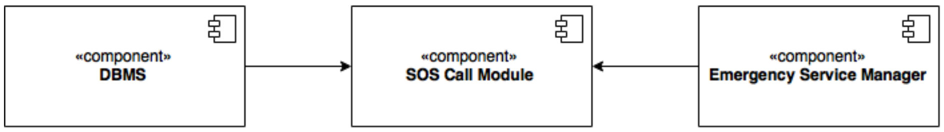

Figure 6: I2.3 - SOS Call Module

Car Controller should make the *ManageCar* interface available to *Navigation Controller*, while *Rental Module* should provide the *ManageRental* interface to *Navigation Controller*, so *Car Controller* and at least some features of *Rental Module* should be implemented before *Navigation Controller*; similarly, *SOS Call Module* should be implemented before *Navigation Controller.*

Employee web services This subsystem has two main subcomponents and, as previously stated, their implementation can be performed without a specific order. Both *Fleet Manager* and *Field Service Module* need *DBMS* and *Maps API* components in order to function correctly, so first we need to test those integrations. In addition, *Fleet Manager* requires the integration with *Car Controller* and *Emergency Service manager* components.

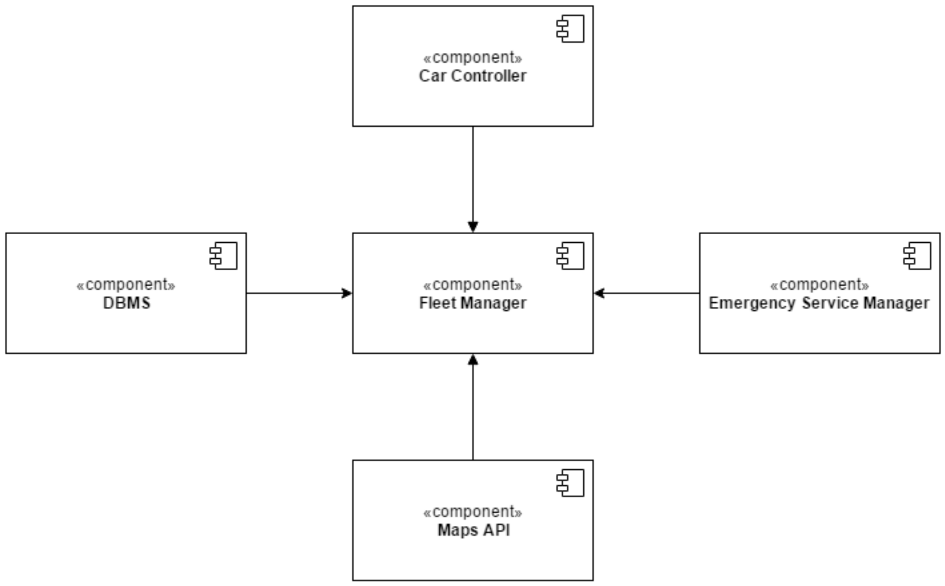

Figure 7: I3.1 - Fleet Manager

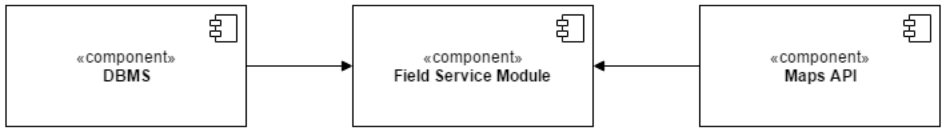

Figure 8: I3.2 - Field Service Module

2.4.2 Subsystem Integration Sequence

The following diagrams presents the integration sequence of the front-end components of the platform subsystems:

- *Customer mobile services:*
  - *Customer mobile application*
- *Car services:*
  - *On-board tablet Application*
  - *Telematic Control Unit*
- *Employee web services:*
  - *Web Application*

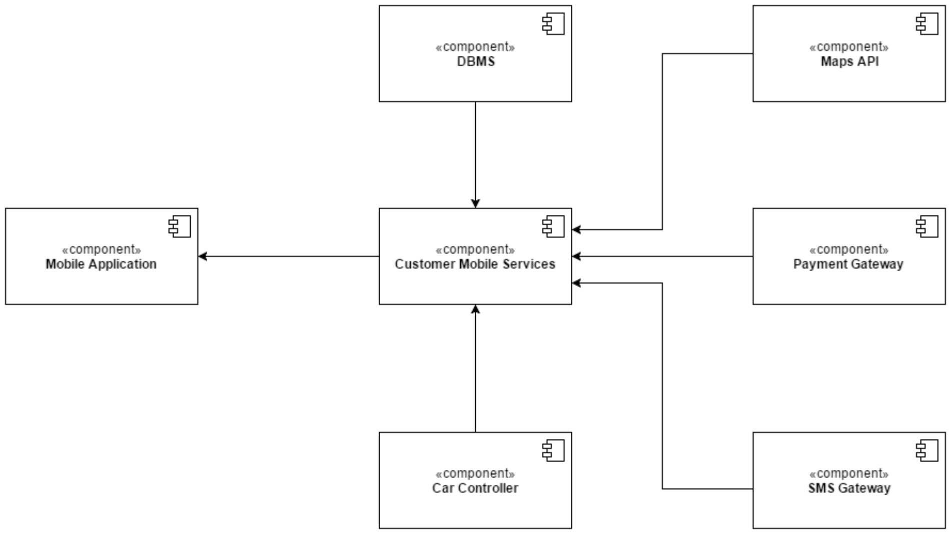
«component» DBMS
«component» Maps API
«component» Mobile Application
«component» Customer Mobile Services
«component» Payment Gateway
«component» Car Controller
«component» SMS Gateway

Figure 9: I1 - Customer mobile services

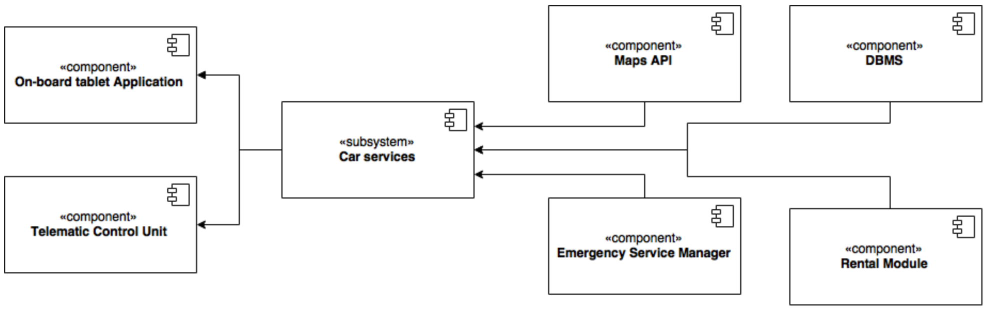

Figure 10: I2 - Car services

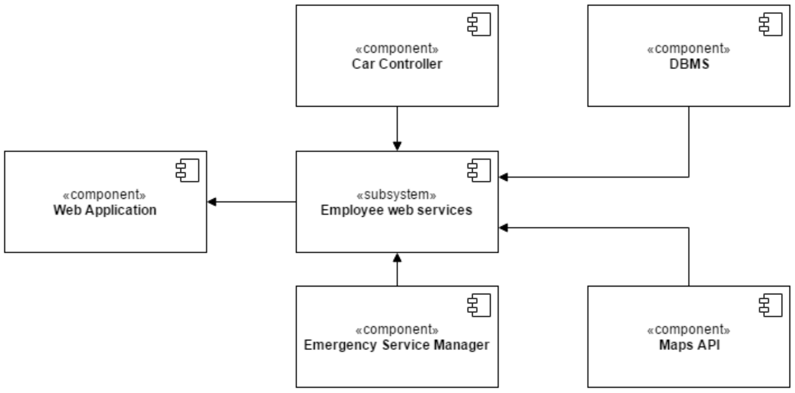

Figure 11: I3 - Employee web services

Once all the subcomponents of each subsystem are fully integrated, high-level subsystems should be integrated together in order to deploy the *PowerEnJoy Platform*.

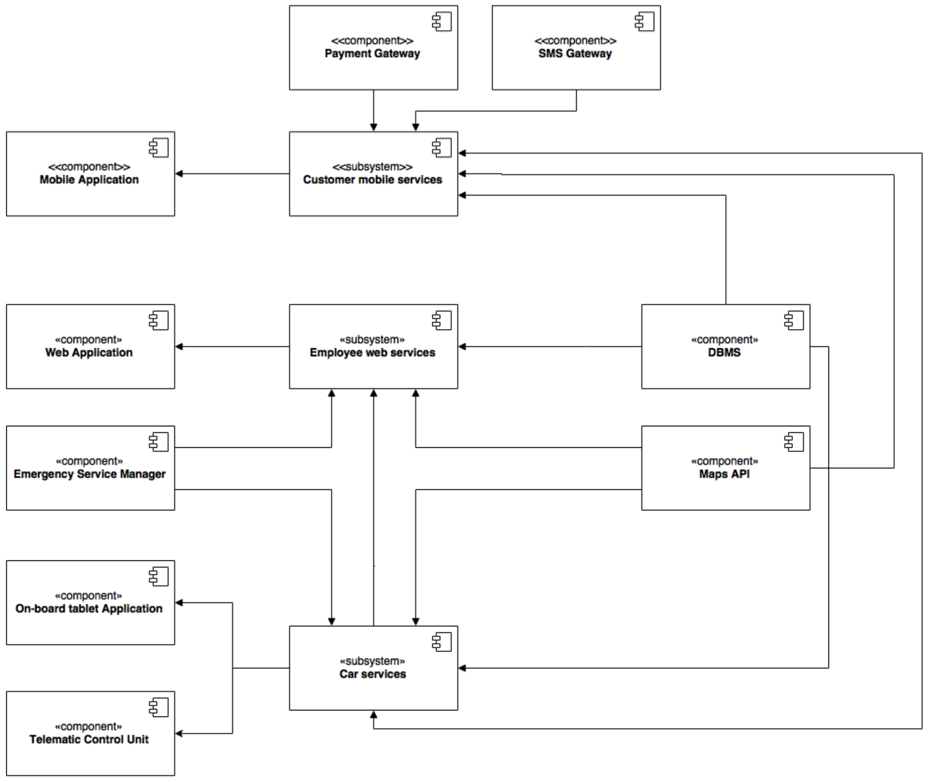

Figure 12: System integration

3 Individual Steps and Test Description

Note that some of the test cases related to system components contain an *Exception* inside the *Output specification* field: this indicates that in case of invalid arguments or invalid method invocation an exceptional output is produced. The same exception implicitly applies to all the subsystem test cases which involve system components containing exceptions in *Output specification.*
Also note that when the *DBMS* component is mentioned, we assume that a proper database has been configured and activated.

Feedback Module - Integration test case I1.1

Test ID	I1.1 T1
Components	Feedback Module, DBMS
Input specification	Send feedback message
Output specification	Check if the report was stored in the database correctly
Description	Feedback Module sends an update query to the DBMS in order to append the message to the list of existing messages
Environmental needs	-

Account Manager - Integration test case I1.2

Test ID	I1.2 T1
Components	Account Manager, DBMS, SMS Gateway
Input specification	Register new account
Output specification	Check if the new account is available from the database and if the SMS was correctly sent • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the account should not be created
Description	Account Manager sends an update query to the DBMS in order to append the message to the list of existing accounts and uses the SMS Gateway for sending the confirmation message to the user
Environmental needs	-

Test ID	I1.2 T2
Components	Account Manager, DBMS
Input specification	Login with existing account
Output specification	Check if the login was successful • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the login request should be denied
Description	Account Manager sends a select query to the DBMS in order to verify the account credentials
Environmental needs	-

Test ID	I1.2 T3
Components	Account Manager, DBMS, Payment Gateway
Input specification	Add payment method to the account
Output specification	Check if the payment method was successfully registered through the Payment Gateway by completing a "dummy" rent session
Description	The data of the payment method is transferred to the 3rd party Payment Gateway (without storing it into the DBMS)
Environmental needs	I1.3 (T1), I2.2 (T2)

Rental Module - Integration test case I1.3

Test ID	I1.3 T1
Components	Rental Module, DBMS
Input specification	Reserve a car
Output specification	Check if the reservation was stored in the database correctly and if the car state is set to *Reserved* • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the reservation should be denied
Description	Rental Module sends a select query to the DBMS in order to verify that the car is available, then sends an update query so that the car state is changed
Environmental needs	-

Test ID	I1.3 T2
Components	Rental Module, DBMS
Input specification	Cancel reservation
Output specification	Check in the database if the reservation state was changed to *Canceled* • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the reservation shouldn't be canceled
Description	Rental Module sends and update query to the DBMS in order to change the reservation state to "cancelled"
Environmental needs	-

Test ID	I1.3 T3
Components	Rental Module, DBMS, Maps API
Input specification	Search cars
Output specification	Check that all available cars in the desired range are properly displayed in the user map • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), an error should be notified via pop-up message
Description	Rental Module sends a select query to the DBMS in order to fetch the data required to display the cars position and status
Environmental needs	-

Test ID	I1.3 T4
Components	Rental Module, DBMS, Car Controller
Input specification	Begin rent session
Output specification	Check in the DBMS that the car state was properly changed and that the car was actually unlocked and manned by someone • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the rent session shouldn't be allowed to start
Description	Rental Module is asked by Car Controller (via the *ManageRental* interface) to send an update query to the DBMS in order to change the car state to "rented" and start charging the user with the proper tariff
Environmental needs	-

Test ID	I1.3 T5
Components	Rental Module, DBMS, Car Controller, Payment Gateway
Input specification	End rent session
Output specification	Check in the database that the car state was properly changed and that the car engine was turned off • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the rent session shouldn't be allowed to end
Description	Rental Module is asked by Car Controller (via the *ManageRental* interface) to send an update query to the DBMS in order to change the car state to "available", stop charging the user and complete the payment transaction through the Payment Gateway
Environmental needs	-

Test ID	I1.3 T6
Components	Rental Module, DBMS, Car Controller
Input specification	Start TPM
Output specification	Check in the database that the car state was properly changed and that the car engine was turned off • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the TPM shouldn't be allowed to start
Description	Rental Module is asked by Car Controller (via the *ManageRental* interface) to send an update query to the DBMS in order to change the car state to "temporarily parked " and switch from the normal tariff to the discounted one
Environmental needs	-

Test ID	I1.3 T7
Components	Rental Module, DBMS, Car Controller
Input specification	End TPM
Output specification	Check in the database that the car state was properly changed • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the rent session shouldn't be allowed to end
Description	Rental Module is asked by Car Controller (via the *ManageRental* interface) to send an update query to the DBMS in order to change the car state to "in use" and switch form the discounted tariff to the normal one
Environmental needs	-

Test ID	I1.3 T8
Components	Rental Module, DBMS, Car Controller
Input specification	Scan QR Code
Output specification	Check if the lock state of the car is changed to *Unlocked* • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the lock state of the car should not change
Description	After validating the QR code through a select query to the DBMS, Rental Module sends an unlock request via *ManageCar* interface to the Car Controller
Environmental needs	I2.1 T1

Subsystem: Customer mobile services - Integration test case I1

Test ID	I1 T1
Components	Customer mobile services, Mobile Application, DBMS, Payment Gateway, SMS Gateway
Input specification	Account creation and setup
Output specification	Check the output of each functionality involved
Description	Every functionality of the Customer mobile services concerning the creation of the account, the log-in process and the payment method setup is tried in a single test case to verify the integrity of the whole system
Environmental needs	I1.2 (T1, T2, T3)

Test ID	I1 T2
Components	Customer mobile services, Mobile Application, DBMS, Payment Gateway, Car Controller
Input specification	Rent session creation and completion (with TPM usage)
Output specification	Check the output of each functionality involved
Description	Every functionality of the Customer mobile services concerning the reservation of the car, its unlock and usage (with at least one TPM switch) and the end of the rental is tested in a single test case to verify the integrity of the whole system
Environmental needs	I1.3 (T1, T2, T3, T4, T5, T6, T7), I2.1

Car Controller - Integration test case I2.1

Test ID	I2.1 T1
Components	Car Controller, DBMS
Input specification	Remote unlock of a specific car upon request by other system component
Output specification	Check if the lock state of the car is changed to *Unlocked.* • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the lock state of the car (if an existing one has been specified) should not change
Description	Car Controller receives an unlock request via *ManageCar* interface from another component of the system (e.g. Fleet Manager or Rental Module) for a specific vehicle, so it sends an update query to the DBMS in order to change the car lock state
Environmental needs	-

Test ID	I2.1 T2
Components	Car Controller, DBMS
Input specification	Remote lock of a specific car upon request by other system component
Output specification	Check if the lock state of the car is changed to *Locked* • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the lock state of the car (if an existing one has been specified) should not change
Description	Car Controller receives a lock request from another component of the system (e.g. Fleet Manager) for a specific vehicle, so it sends an update query to the DBMS in order to change the car lock state
Environmental needs	-

Test ID	I2.1 T3
Components	Car Controller, DBMS
Input specification	Update of car status values (GPS position, speed, battery level, engine status, number of passengers) of a vehicle in the DB
Output specification	Check if the car status values had been properly updated in the tuple related to the vehicle • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the car status values of the vehicle (if an existing one has been specified) should not change
Description	Car Controller sends an update query to the DBMS in order to change the car status values of a specific car in the DB
Environmental needs	-

Navigation Controller - Integration test case I2.2

Test ID	I2.2 T1
Components	Navigation Controller, DBMS, Rental Module
Input specification	Start rental on a specific vehicle
Output specification	Check if the car state of the vehicle is changed to *inUse* • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the car state values of the vehicle (if an existing one has been specified) should not change
Description	Navigation Controller sends an update query to the DBMS in order to change the car state. Furthermore, it requires the Rental Module (via *ManageRental* interface) to start the rental session of the selected vehicle
Environmental needs	I1.3 (T4)

Test ID	I2.2 T2
Components	Navigation Controller, DBMS, Rental Module
Input specification	End rental on a specific vehicle
Output specification	Check if the car state of the vehicle is changed to *Available* • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the car state values of the vehicle (if an existing one has been specified) should not change
Description	Navigation Controller sends an update query to the DBMS in order to change the car state. Furthermore, it requires the Rental Module (via *ManageRental* interface) to end the rental session of the selected vehicle
Environmental needs	I1.3 (T5)

Test ID	I2.2 T3
Components	Navigation Controller, DBMS, Rental Module
Input specification	Activate Temporary parking mode on a specific vehicle
Output specification	Check if the car state of the vehicle is changed to *TemporaryParked* • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the car state values of the vehicle (if an existing one has been specified) should not change
Description	Navigation Controller sends an update query to the DBMS in order to change the car state. Furthermore, it requires the Rental Module (via *ManageRental* interface) to start the Temporary parking mode on the selected vehicle
Environmental needs	I1.3 (T6)

Test ID	I2.2 T4
Components	Navigation Controller, DBMS
Input specification	Save *initial report* for a specific rental
Output specification	Check if the *initial report* has been appended to the list of reports of the vehicle object of the rental • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), the new initial report (if one has been specified) should not be stored
Description	Navigation Controller sends an update query to the DBMS in order to append the new *initial report* to the list of already existing reports
Environmental needs	-

Test ID	I2.2 T5
Components	Navigation Controller, DBMS, Maps API
Input specification	Determine Money Saving Option destination location having as input starting location A and desired destination B
Output specification	Check if the *Destination choice with Money Saving Option algorithm* has not raised *NoSafeAreasAvailableException* • Exception: if the method is invoked on a non-valid object (see *5.2 Test Data* section), *NoSafeAreasAvailableException* should be raised
Description	Navigation Controller computes a destination using the *Destination choice with Money Saving Option algorithm* and Maps API
Environmental needs	-

SOS Call Module - Integration test case I2.3

Test ID	I2.3 T1
Components	SOS Call Module, DBMS, Emergency Service Manager
Input specification	Start a SOS VoIP call
Output specification	Check if the VoIP call request has received a positive response by the Emergency Service Manager
Description	SOS Call Module sends a VoIP call request to the Emergency Service Manager; a positive response and an active VoIP stream should be returned to the SOS Call Module.
Environmental needs	-

Subsystem: Car services - Integration test case I2

Test ID	I2 T1
Components	Car on-board tablet Application, Car services, DBMS, Customer Mobile Services (Rental Module)
Input specification	Fill in *initial report* and start rental
Output specification	Check that the *initial report* has been stored in the DB and that the car state has been changed to *inUse*
Description	Navigation Controller receives an *initialReport* method invocation via *Navigate* interface from the Car on-board tablet Application, so it save the *initial report* in the DB and invokes method *updateCarState* in order to change the state of the car from *Rented* to *inUse*
Environmental needs	I2.1, I2.2, I1.3

Test ID	I2 T2
Components	Car on-board tablet Application, Car services, DBMS, Maps API
Input specification	Show map of safe areas
Output specification	Check if safe area locations are properly rendered on a map via Maps API
Description	Navigation Controller receives a *viewMapSafeArea* method invocation via *Navigate* interface from the Car on-board tablet Application, so it fetches the safe area locations from the DB and renders them on a map using the Maps API
Environmental needs	I2.2

Test ID	I2 T3
Components	Car on-board tablet Application, Car services, DBMS, Maps API
Input specification	Start navigation to destination
Output specification	Check if turn-by-turn navigation is correctly activated after calling the Maps API
Description	Navigation Controller receives a *setNavigation* method invocation via *Navigate* interface from the Car on-board tablet Application (*Get directions* or *Get directions with Money Saving Option* features), so it starts turn-by-turn navigation app on the Car on-board tablet via an *intent*
Environmental needs	I2.1, I2.2

Test ID	I2 T4
Components	Car on-board tablet Application, Car services, Customer Mobile Services (Rental Module), DBMS, Maps API
Input specification	Enable Temporary Parking mode
Output specification	Check if turn-by-turn navigation has been paused and the vehicles car state has been changed to *TemporaryParked*
Description	Navigation Controller receives a *enableTemporaryParkMode* method invocation via *Navigate* interface from the Car on-board tablet Application, so it temporarily stops turn-by-turn navigation on the Car on-board tablet and changes the car state to *TemporaryParked*
Environmental needs	I2.1, I2.2, I1.3

Test ID	I2 T5
Components	Car on-board tablet Application, Car services, Customer Mobile Services (Rental Module), DBMS, Maps API
Input specification	Disable Temporary Parking mode
Output specification	Check if turn-by-turn navigation has been reactivated, vehicles car state has been changed to *inUse* and temporary parking information has been saved in the rental record.
Description	Navigation Controller receives a *disableTemporaryParkMode* method invocation via *Navigate* interface from the Car on-board tablet Application, so it reactivates turn-by-turn navigation on the Car on-board tablet, it changes the car state to *inUse* and stores temporary parking information in the rental record.
Environmental needs	I2.1, I2.2, I1.3

Test ID	I2 T6
Components	Car on-board tablet Application, Car services, DBMS, Maps API
Input specification	Show discount opportunities
Output specification	Check if discount opportunities are properly rendered on the on-board tablet Application
Description	Navigation Controller receives a *viewDiscountOpportunities* method invocation via *Navigate* interface from the Car on-board tablet Application, so it hides turn-by-turn navigation and display discount opportunities screen to the user
Environmental needs	I2.1, I2.2

Test ID	I2 T7
Components	Car on-board tablet Application, Car services, Customer Mobile Services (Rental Module), DBMS, Maps API
Input specification	End rental
Output specification	Check if the turn-by-turn navigation has been ended and car state has been changed to *Available*
Description	Navigation Controller receives a *endRental* method invocation via *Navigate* interface from the Car on-board tablet Application, so it stops turn-by-turn navigation and changes the car state to *Available.*
Environmental needs	I2.1, I2.2, I1.3

Test ID	I2 T8
Components	Car on-board tablet Application, Car services, DBMS, Emergency Service Manager
Input specification	Start a SOS Call between vehicle and Fleet Manager
Output specification	Check if the SOS report creation process has been started and if the VoIP call has been correctly activated
Description	Navigation Controller receives an SOS request via *Navigate* interface from the Car on-board tablet Application (*launchSOSCall* method invocation), so it sends a call request to Emergency Service Manager and a SOS report creation request (*insertSOSCall* method invocation) to Employee web services
Environmental needs	I2.3, I3.1

Test ID	I2 T9
Components	Telematic Control Unit, Car services, DBMS
Input specification	Transmit car status values (GPS position, speed, battery level, engine status, number of passengers) of a vehicle to the backend of the system
Output specification	Check if the car status values have been properly updated
Description	Telematic Control Unit periodically sends an update request to the Car services subsystem invoking the *updateCarState* method via *ManageCar* interface of the Car Controller
Environmental needs	I2.1

Test ID	I2 T10
Components	Telematic Control Unit, Car services, DBMS
Input specification	Actuation of a change in the lock state of a vehicle
Output specification	Check if the vehicle has been locked/unlocked accordingly to the lock state change
Description	The Telematic Control Unit of each vehicle constantly observes the car lock state and physically locks/unlocks the vehicle accordingly
Environmental needs	I2.1

Fleet Manager - Integration test case I3.1

Test ID	I3.1 T1
Components	Fleet Manager, DBMS, Car services (Car Controller)
Input specification	Remote unlock of a specific car
Output specification	Check if the car lock state has been changed to *Unlocked*
Description	Fleet Manager invokes the *Unlock* method via ManageCar interface, then it checks if the car lock state has been changed to *Unlocked* • Exception: If the car is already unlocked it returns the exception *CarAlreadyUnlokedException* and if the is an invalid object as parameter (see the 5.2 *Test Data* section) it raises an *InvalidParameterException*
Environmental needs	I.2.1

Test ID	I3.1 T2
Components	Fleet Manager, DBMS, Emergency Service Manager
Input specification	Creation of the well formed SOS call report
Output specification	Check if the record has been saved correctly on the database
Description	After checking the input parameter and retrieving the call audio registration from the Emergency Service Manager, the SOS call report is created and saved in the DB • Exception: if the is an invalid object as parameter (see the 5.2 *Test Data* section) it raises an *InvalidParameterException* and the report is not created
Environmental needs	-

Field Service module - Integration test case I3.2

Test ID	I3.2 T1
Components	Field service module, DBMS
Input specification	Tag as completed an assigned task
Output specification	Check if in the database the task state is changed to *completed*
Description	Field service module calls an update query to the DBMS in order to change the task state • Exception: if the is an invalid object as parameter (see the 5.2 *Test Data* section) it raises an *InvalidParameterException*
Environmental needs	-

Subsystem: Employee web services - Integration test case I3

Test ID	I3 T1
Components	Web application, Employee web services, DBMS, Maps API
Input specification	Set a car state to maintenance required then assign its recovery to a field agent
Output specification	Check if the car has changed state to *maintenanceRequired* and if the field agent has a new task with it
Description	Initially, *addMaintenanceActivity* method is invoked and the Fleet Manager calls the *updateCarState* method in order to change the car state. Then, calling *assignCarToFieldAgent* method the task is assigned to a field agent. Finally, *showTask* method is invoked and it is checked if the field agent has been assigned the new task. • Exception: if the is an invalid object as parameter (see the 5.2 *Test Data* section) it raises an *InvalidParameterException*
Environmental needs	I2.1, I3.1, I3.2

Test ID	I3 T2
Components	Web application, Employee web services, DBMS, Maps API
Input specification	Display the fleet in the web application
Output specification	Check if properly rendered via Maps API
Description	Fleet Manager component forwards the request to the DBMS through a specific query and with the Maps API provides as output the list of the car displayed on the map
Environmental needs	I3.1, I3.2

4 Tools and Test Equipment Required

4.1 Tools

The testing process may be supported by the usage of a multiplicity of software verification tools which allow to run large number of tests during the development and verification of the system.

Assuming that the majority of the components will be implemented using the Java Enterprise Edition platform, a collection of tools may be used to speed up and partially automate testing activities while developing and integrating different modules.

- The first tool which we are going to use in this phase is **JUnit**, a testing framework mainly designed for unit testing activities, which may also be useful for testing the interfaces of specific components. More specifically, considering that JUnit allows to verify assertions on return values after method invocations, this tool can be used to check if a method, for example one the methods defined in the *Component interfaces diagram* (*Design Document*, section 2.6), correctly returns the expected objects. The same approach may also be used to check if exceptions are properly raised when invalid parameters are passed, when an error occurs or when an object is not found. Interaction testing will also be supported by **Mockito**, a mocking framework for unit tests which may be used for the scaffolding activity, helping the developer defining stubs when a component cannot be tested in isolation.

- The second tool which we are going to leverage for a proper integration of the components of our system is **Arquillian**, an integration testing framework specifically designed for Jave EE. Arquillian tests are written in a JUnit-like syntax and allow to execute test cases agains containers unburdening the developer from aspects surrounding test execution (containers lifecycle management, dependencies bundling for test classes, test results capture, ...). To be more specific, Arquillian will be used to verify that the interactions with the databases work properly, but also to check whether components are correctly injected when dependency injection is specified.

- When the most meaningful features will be almost fully implemented, manual testing will also be used: testers will play the role of end users performing sequence of actions on the mobile app, tablet app and web application in order to verify that all the integrated components work properly. Use cases (*Requirements Analysis and Specification Document*) and test cases will provide useful information about the main functionalities to be tested.

- Non-functional requirements testing will be addressed with specific performance testing tools.

  - The open source software **Apache JMeter** will be used to simulate a heavy load on servers, but also to analyze overall performance under different load types, without actual browser sessions, but working directly at protocol level with a multi-threading framework for simultaneous sampling of different functions.
  - Platform-specific tools will be used to test performance on mobile device in terms of CPU and memory usage, but also battery consumption. More specifically:
    * *Memory Monitor* in *Android Studio* shows how the app allocates memory over the course of a single session through a graph of available and allocated Java memory over time, including garbage collection events; *Allocation Tracker* tool in *Android Studio* provides further details about how the app allocates memory; *Profile GPU Rendering* and *Battery Historian* tools in *Android Studio* help to identify areas that may be optimized for better battery life;
    * *Graphics Tools* in *Xcode* will be used to gather GPU-related performance data, *Activity Monitor* in *Xcode* provides usage statistics relating to memory and CPU usage for the currently running processes, *malloc_history* command-line tool shows the malloc allocations performed by a specified process, *Energy Impact gauge* in *Xcode* provides an overview of energy usage of apps;
    * *Windows Performance Analyzer*, included in *Windows Assessment and Deployment Kit*, is a tool that provides useful insight about performance creating graphs and data tables while testing Windows Phone apps.

4.2 Test Equipment

Testing activities on the *PowerEnJoy Platform* will require the usage of both physical devices, simulators and software tools, as already mentioned in the previous paragraph.

- For what concerns the client side of the platform, the following devices and simulators will be required:
  - Customer mobile app:
    * *Android Emulator* in *Android studio*, *Simulator* in *Xcode* and *Microsoft Emulator* will be used to simulate the usage of the mobile application on a variety of different devices in terms of OS versions, screen sizes and resolutions.
    * Physical devices will also be employed in order to test specific features which need access to physical components or specific smartphone features, such as GPS, camera and SMS. In particular, smartphone will be selected among different categories in

    order to test the app on cheap, mid-range and high-end devices with screen sizes from 3" to 6", taking into account that some display sizes or resolutions may not be available in iOS products. The choice of devices will be driven by the purpose of covering the majority of operating scenarios.

  - Car on-board tablet app:
    * the app will run on a customized Android tablet with a 7-inch IPS screen (1920 x 1200 Full-HD resolution), so test will be executed both on the physical device and on *Android Emulator* in *Android studio.*
  - Employee web-app:
    * the app will be tested on computer, laptops and tablets with a variety of configurations in terms of OS, browsers, display sizes and resolution. The testing environment choices will always take into account the *Operating Environment* requirements stated in the *Requirements Analysis and Specification Document.*

- The back-end of the system will be tested on a specific testing environment which faithfully simulates the real operating environment in terms of hardware and software. We assume to adopt the implementation suggested in the *Deployment view - Recommended implementation* (*Design Document*, section 2.4):
  - *Oracle SPARC T7* server with *Oracle Solaris 11.3* operating system,
  - *Java Enterprise Edition runtime,*
  - *Apache HTTP Server* as a web server,
  - *Apache Tomcat* script engine and *JBoss* application server,
  - *Oracle database 12c database management system.*

5 Program Stubs and Test Data Required

5.1 Program Stubs and Driver

As already presented in the Integration Strategy section, a bottom-up approach will be adopted while integrating and testing components of *PowerEnJoy Platform*. This is the reason why we will need some *drivers*, pieces of software that "drive" the modules to be tested creating the inputs required by the component being tested.

Here are the main drivers:

Customer mobile services

- **Feedback Module driver**: it will invoke methods of the *Feedback* interface in order to perform the test cases described in section 3, subsection I1.1
- **Rental Module driver**: it will invoke methods of the *RentalHistory* and *BookSession* interface in order to perform the test cases described in section 3, subsection I1.3
- **Account Manager driver**: it will invoke methods of the *ProfileManagement* interface in order to perform the test cases describe in section 3, subsection I1.2
- **Mobile Application driver**: it will simulate the features of the Mobile Application (utilized by customers) in order to perform the test cases described in section 3, subsection I1

Car services

- **Car Controller driver**: it will invoke methods of the *ManageCar* interface provided by Car Controller in order to perform the test cases described in section 3, subsection I2.1
- **Navigation Controller driver**: it will invoke methods of the *Navigation* interface provided by Navigation Controller in order to perform the test cases described in section 3, subsection I2.2
- **SOS Call module driver**: it will invoke methods of the *SOScall* interface provided by SOS Call Module in order to perform the test cases described in section 3, subsection I2.3
- **Car on-board tablet application driver**: it will mimic the features of the car on-board tablet application in order to perform the test cases described in section 3, subsection I.2
- **Telematic Control Unit driver**: it will simulate the actions of the Telematic Control Unit of a vehicle in order to perform the test cases described in section 3, subsection I.2

Employee web services

- **Fleet Manager driver**: it will invoke the methods of the two interfaces provided by *Fleet Manager* component, such as *ManageFleet* and *SOScall* which are needed to fulfill the test cases of section 3, subsection I3.1
- **Field service module driver**: it will invoke the methods of the interface *FieldService*, in particular on test **I3.1 T1** *showTask* and *confirmTaskCompletion* will be invoked*:* the first is called to display an existing task and the second to perform the test goal.
- **Web application driver**: it will work as the web application and it will invoke the methods provided by the interfaces of *Employee web services* in order to test its functionalities. It is used during the tests about the integration I3

5.2 Test Data

The following paragraph presents a set of test data which will be used to perform the aforementioned test cases and simulate both valid and invalid inputs.
Please note that some test cases may not require specific test input data: in the case, the test will not be specified in the following list.

Customer mobile services

- Test I1.1 T1 on the **Feedback Module** component requires a *Feedback report* object and the corresponding *message* object. The test will be performed with instances of :
  - Null object as *Feedback Report* object
  - Null object as *message* object
  - *Feedback Report* object and mismatching *message* object
  - Correct instances of *Feedback Report* object and relative *message* object
- Test I1.2 T1 on the **Account Manager** component requires a *Profile* object. The test will be performed with instances of :
  - Null object as *Profile* object
  - Null object as any of the compulsory fields of the profile creation section (e.g. *email, phone number....*)
  - *Profile* object and incorrect *email* object, *drivingLicenceNumber* object and *drivingLicenceExpirationDate*
  - Correct instances of *Profile* object and its values
- Test I1.2 T2 on the **Account Manager** component requires a *Profile* object. The test will be performed with instances of :

  - Null object as *Profile* object
  - Null object as *name* object and *password* object
  - *Profile* object and invalid *name* object and *password* object
  - Correct instances of *Profile* object, *name* object and *password* object

- Test I1.2 T3 of the **Account Manager** component requires a *Profile* object and the corresponding *Payment method* object. The test will be performed with instances of :
  - Null object as *Profile* object
  - Null object as *Payment method* object
  - *Profile* object and invalid *Payment method* (invalid field for each field of the Payment method)
  - Correct instances of *Profile* object and *Payment method* object

- Test I1.3 T1 on the **Rental Module** component requires a *Car* object. The test will be performed with instances of :
  - Null object as *Car* object
  - Null object as *carState* object
  - *Car* object and incorrect *carState* object (e.g. Rented, Unavailable ...)
  - Correct instances of *Car* object and *carState* object (Available)

- Test I1.3 T2 on the **Rental Module** component requires a *Car* object and one corresponding *Rental* object. The test will be performed with instances of :
  - Null object as Car object
  - Null object as Rental object
  - Car object and mismatching Rental object.
  - Correct instances of Car object and Rental object

- Test I1.3 T3 on the **Rental Module** component requires multiple objects of type Car. The test will be performed with instances based on the ones you can find in test I1.3 T2

- Test I1.3 T4 on the **Rental Module** component may be performed with the same test data as test I1.3 T2

- Test I1.3 T5 on the **Rental Module** component requires a Car object, one corresponding Rental object and a Payment Method object. The test will be performed with instances of :
  - Null object as Car object

  - Null object as Rental object
  - Car object and mismatching Rental object.
  - Car object, Rental object and null object as Payment Method object
  - Correct instances of Car object, Rental object and Payment Method object

- Test I1.3 T6 on the **Rental Module** component may be performed with the same test data as test I1.3 T2
- Test I1.3 T7 on the **Rental Module** component may be performed with the same test data as test I1.3 T2
- Test I1.3 T8 on the **Rental Module** component requires a Car object, one corresponding Rental object and one QR Code object. The test will be performed with instances of :
  - Null object as Car object
  - Null object as Rental object
  - Car object and mismatching Rental object.
  - Car object, Rental object and null object as QR Code object
  - Correct instances of Car object, Rental object and QR Code object
- Test I1 T1 and I2 T2 on the **Customer Mobile Services Subsystem** may be performed with the same test data of their respective Component Test Cases

Car services

- Test I2.1 T1 on the **Car Controller** component requires a *Car* object. The test will be performed with instances of:
  - Null object as *Car* object
  - Correct instances of *Car* object
  - Car which is already *unlocked*
- Test I2.1 T2 on the **Car Controller** component requires a *Car* object. The test will be performed with instances of:
  - Null object as *Car* object
  - Correct instances of *Car* object
  - Car which is already locked
- Test I2.1 T3 on the **Car Controller** component requires a *Car* object and a *CarStatus* object The test will be performed with instances of:
  - Null object as *Car* object

  - Null object as *CarStatus* object
  - *CarStatus* objects containing invalid values (eg. negative speed, out of bound GPS coordinates, ...)
  - Correct instances of *Car* object and *CarStatus* object

- Test I2.2 T1 on the **Navigation Controller** component requires a *Car* object. The test will be performed with instances of:
  - Null object as *Car* object
  - Correct instances of *Car* object (car with car state equal to *Rented)*
  - *Car* object with a car state different from *Rented*

- Tests I2.2 T2 and I2.2 T3 on the **Navigation Controller** component require a *Car* object. The test will be performed with instances of:
  - Null object as *Car* object
  - Correct instances of *Car* object (car with car state equal to *inUse)*
  - *Car* object with a car state different from *inUse*

- Test I2.2 T4 on the **Navigation Controller** component requires a *Car* object and an *initialReport* object. The test will be performed with instances of:
  - Null object as *Car* object
  - Null object as *initialReport* object
  - Correct instances of *Car* object (car with car state equal to *Rented)* and *initialReport* object
  - *Car* object with a car state different from *Rented*

- Test I2.2 T5 on the **Navigation Controller** component requires two *location* objects (GPS coordinates or address in textual format). The test will be performed with instances of:
  - Null objects as *location* objects
  - invalid *location* objects (out of bound GPS coordinates, invalid or not existing address, coincident *location* objects, ...)
  - Correct instances of distinct *location* objects

- Test I2 T1 on the **Car services subsystem** may be performed with the same test data of test I2.2 T4

- Test I2 T4 on the **Car services subsystem** requires a *Car* object. The test will be performed with instances of:
  - Null object as *Car* object
  - Correct instances of *Car* object (car with car state equal to *inUse)*

  - *Car* object with a car state different from *inUse*

- Test I2 T5 on the **Car services subsystem** requires a *Car* object. The test will be performed with instances of:
  - Null object as *Car* object
  - Correct instances of *Car* object (car with car state equal to *TemporaryParked)*
  - *Car* object with a car state different from *TemporaryParked*

- Test I2 T7 on the **Car services subsystem** may be performed with the same test data of tests I2.2 T2 and I2.2 T3

- Test I2 T9 on the **Car services subsystem** may be performed with the same test data of tests I2.1 T3

- Test I2 T10 on the **Car services subsystem** requires a *CarLockState* object. The test will be performed with instances of:
  - Null object as *CarLockState* object
  - Correct instances of *CarLockState* object (*CarLockState* equal to *Locked* or *Unlocked)*
  - *CarLockState* objects with a value different from *Locked* or *Unlocked*

Employee web services

- Test I3.1 T1 on the **Fleet Manager** component requires a *Car* object, tested instances of it:
  - Null object
  - Correct Instance
  - Car which is already unlocked

- For test I3.1 T2 on the **Fleet Manager** component are needed several instances of *SOScall record*:
  - Null object
  - Correct instance
  - Invalid parameter (audioRecording, involvedCar, description)

- For test I3.2 T1 on the **Field service module** component is needed the *Task object*, for a completed test it will have two instances:
  - Null object
  - Correct instance

- Test I3 T1 on the **Employee web services subsystem** requires *Car* and *Field Agent* objects, the instances to be tested are:

- Null object
- Correct instances
- Car which is already under maintenance
- Car which is in use

Appendix

Effort Spent

Hours of work spent by each member of the team redacting the document:

- Matteo Franchi: 11,5 h
- Luca Guida: 15 h
- Alessandro Lavelli: 8 h
- *Additional teamwork*: 3 h