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

Students:
Andrea Facchini - 874891
Andrea Milanta - 878403
Antonio Gianola - 877235

PowerEnJoy

Electric Car-sharing Service

INTEGRATION TEST PLAN DOCUMENT

Andrea Facchini
Antonio Gianola
Andrea Milanta

Index

Andrea Facchini
Antonio Gianola
Andrea Milanta

1. Introduction

1.1 Purpose

The purpose of this document is to identify the best suited strategy for the testing of the software.

This document, written before the actual coding, is also meant to be a reference for the order of development of the units.

1.2 Scope

This document provides a valid solution for testing the integration of all the components and subcomponents of the final product.

In this respective it is important to note that this document does not intend to provide a testing solution for every software unit, but only for the integration of different units (components and subcomponents).

To achieve the desired result, software stubs and test equipment are also described.

1.3 Glossary

In this document the following acronyms and definitions have been used

- **PowerEnJoy Management System (PEMS):** It's the software system to be developed.
- **User:** The user of the software. He can register as a new user or access to the car sharing service via login and becoming an Authenticated User.
- **Suspended User:** User who has an account but cannot reserve a car because his driving license or payment information are found not be valid.
- **Money Saving Option (MSO):** Option for the AU to ask for a reduction. The system will suggest the user a SPA near the destination and suitable for a discount, according to the CT.
- **Integration Test Plan (ITP):** Plan for testing the integration of the modules of the PEMS.
- **Onroad Service (OS):** People, provided by an external system but at the disposal of the PEMS, which provide autonomous support for exceptional situations directly on site. The OS also provides an emergency call center where the user may directly communicate with an operator.
- **Payment System:** Any generic external system through which a payment to PowerEnJoy happens.
- **National Transport System (NTS):** AU's country state agency that stores all driving licenses information to be validated by external users.

1.4 Reference Documents

- Assignments AA 2016-2017.pdf
- PowerEnJoy Requirement Analysis and Specifications Document – Version 2
- PowerEnJoy Design Document – Version 2

1.5 Overview

The present document describes the testing process for the PEMS project.
It contains both a high level view of the testing approach and a more detailed focus on the type of tests required.

The document is organized in the following sections:

1. **Introduction**
   This section introduces the purpose, the scope, clarify the terminology and the acronyms and explain the general aspects of the project.
2. **Integration Strategy**
   This section describes the main strategy for the integration testing and the integration sequence of the units.
3. **Individual Steps and Test Description**
   This section describes in detail the integration process introduced at point 2. In particular, it provides the type of test that will be used to verify the proper integration of units.
4. **Tools and Test Equipment**
   This section describes the tools that will be used for testing.
5. **Program Stubs and Test Data**
   This section provides the required program stubs or special test data for testing.
6. **Effort Spent**
   This section contains the hour of work and the distribution of task between the group members.

2. Integration Strategy

2.1 Entry Criteria

This document takes as the main reference the PowerEnJoy Software Design Document.
It is hence required that it and, transitively, the RAS Document have been completed and approved.

The high-level components may not be fully working when the integration test begins. In particular, due to the Thread strategy chosen, when the test of a particular “feature” is started only the required subcomponents must have been implemented.

An exception goes for the following units, which must be completed and fully tested:

- Database and all the required features to access the data (DataAccess).
- External Interfaces, such as NTSInterface and PaymentInterface. This features are provided by the external services, thus they are already working and tested.

MSOCalcolator and EmergencyController are “independent” features that can be developed and added later on.
The graphic parts of the Client and Car applications may also be completed after the integration.

2.2 Elements to be integrated

The PEMS is built upon the interactions of many high-level components, each one possibly made up of many subcomponents, that implement specific functionalities and require other components to work.

With respect to the Design Document, we are going to focus more on the submodules. Since we are using the Thread strategy, testing will work on the subcomponents rather than on the components themselves.

The external interfaces will not be considered in the integration test. Since they are provided by external services, they are supposed to be already complete and fully tested. Furthermore, they only “talk” with one subsystem, hence the communication is tested within the test of the unit which directly interfaces with it.

All other subcomponents will be tested according to the use relationship of each functionality.

In general, the server side subcomponents will be implemented and tested before the client side ones.

To have a complete view of the components and subcomponents refer to the “Design Document – Version 2.pdf” at chapter 2.3 Components View.

2.3 Integration Testing Strategy

The integration test for the PEMS will follow the Thread Strategy.

The system has been designed in modules with different levels. In the highest level the three main modules are identified by the different target device to be implemented with.
At the level below the submodules are identified by more specific tasks that may be common to one or more functionality offered by the system (e.g. payment).

In the PEMS in fact many functionalities develop around different components, and at the same time different components interface themselves more than once and in different manners depending on the functionality.

Subsequently the integration test will work “per functionality” rather than “per module” effectively implementing a Thread strategy at the higher level.

Note that while the main modules will be tested in pieces, those pieces correspond to a full subsystem, thus maintaining a modular approach in the development process.

This strategy offers many advantages. It provides a direct correspondence between the development and testing and the requirements.
Moreover it allows to develop a completely tested system for just some functionalities, possibly allowing for a “reduced” version to be released sooner than the final product for further testing (beta and alpha) and validation.

2.4 Sequence of Components Integration

The thread strategy allows us to develop and test most of the functionalities of the system in parallel.

The sequence of integration thus is only related to the subcomponents required by each functionality, while the whole components will implicitly emerge with the functionalities.

All the functionalities except the MSO and the Emergencies, are considered equally important since they can be developed simultaneously and the end product requires all of them to work.

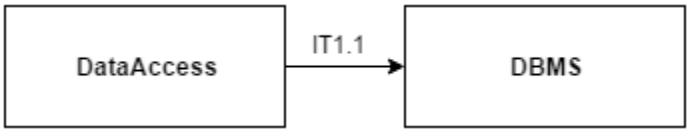
There are two exceptions to this. The DataAccess needs to be working with the DBMS before any other integration test can be done since almost all of the functionalities needs to write or read from the database. The same goes for ChargeUser, required by many functionalities.

Each functionality is tested “bottom up”. Each subcomponent required is developed and tested according to the relationship of need thus avoiding the need of stubs.

Notation: $[C_1] \rightarrow [C_2]$ implies that C_2 provides a function required for C_1 to work in the contest of the current functionality.
Each arrows corresponds to a test between two subcomponents (refer to Chapter 2.3)

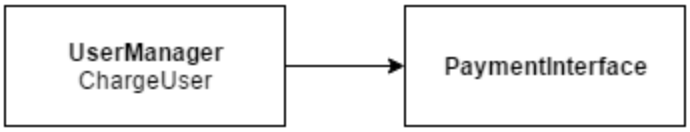
1. Data Access

As previously mentioned the first element to be integrated is the DataAccess submodule with the Database Management System (DBMS). This allows the rest of the system to perform query on the database.



2. Charge User

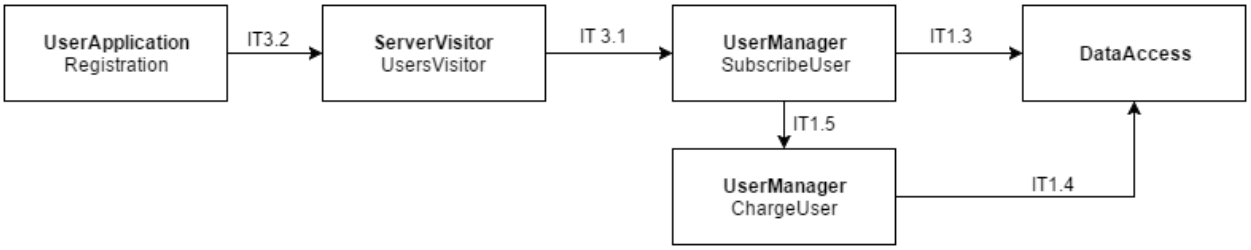
The charge user provides a standard access to the different Payment methods.
The testing, although represented as one, actually needs to be performed for all the agreed payment systems.



Note: ChargeUser is listed only because of its importance and repeated reference. Since it communicates with external systems using interfaces provided by the external services it is not tested as an integration but rather as a single unit.

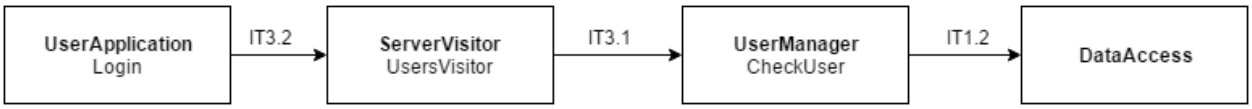
3. Registration

The registration is the functionality that allows a new user to subscribe to the service.
Note that ChargeUser is required to verify the payment information.



4. Login

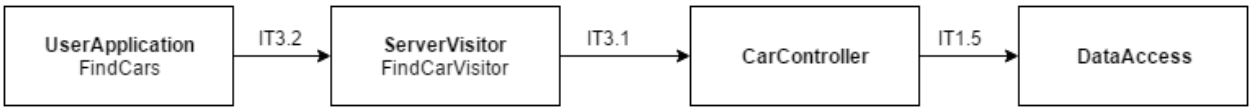
Login allows a registered user to access the server.



Andrea Facchini
Antonio Gianola
Andrea Milanta

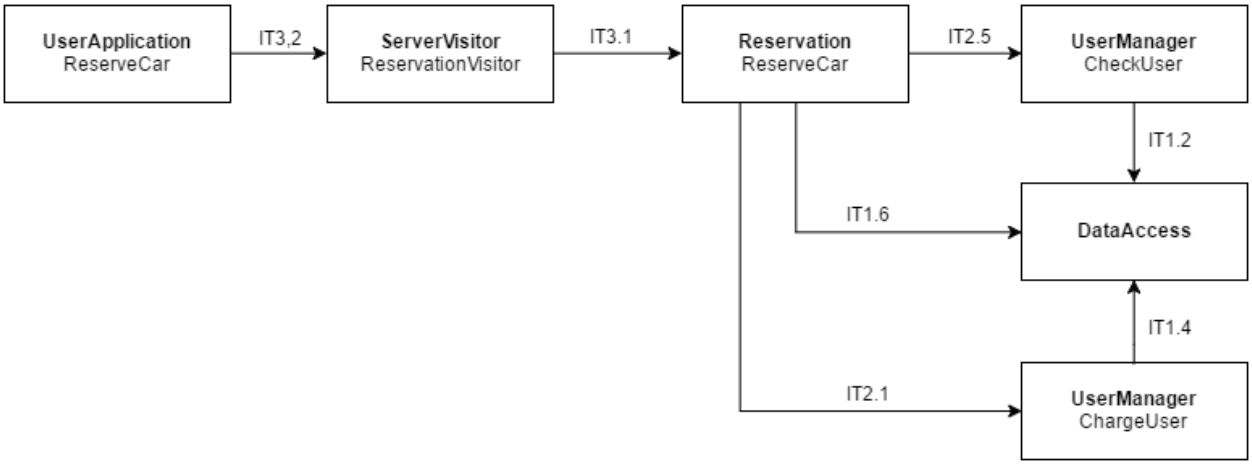
5. Find Cars

This functionality allows the user to find the available cars close to his position or an address of choice.



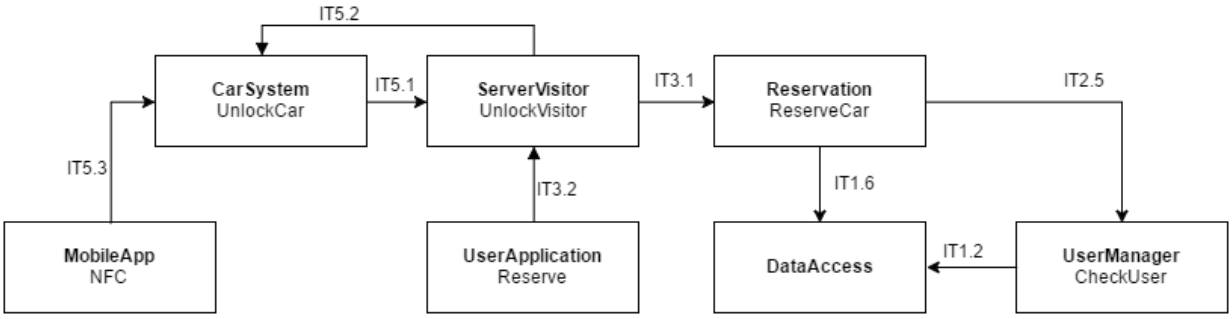
6. Reservation

This functionality allows the user to reserve a car. Charge user is required to pay the “fine” in case of reservation timeout.



7. Unlock

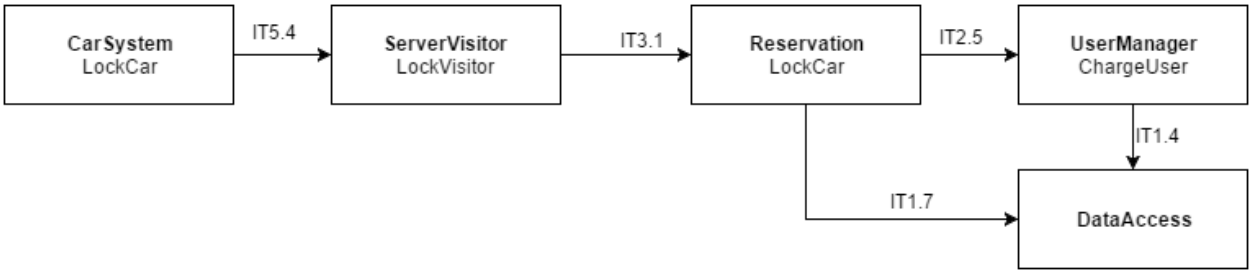
This functionality allows the user to unlock a reserved car. The CheckUser is required to verify the inserted pin.



Andrea Facchini
Antonio Gianola
Andrea Milanta

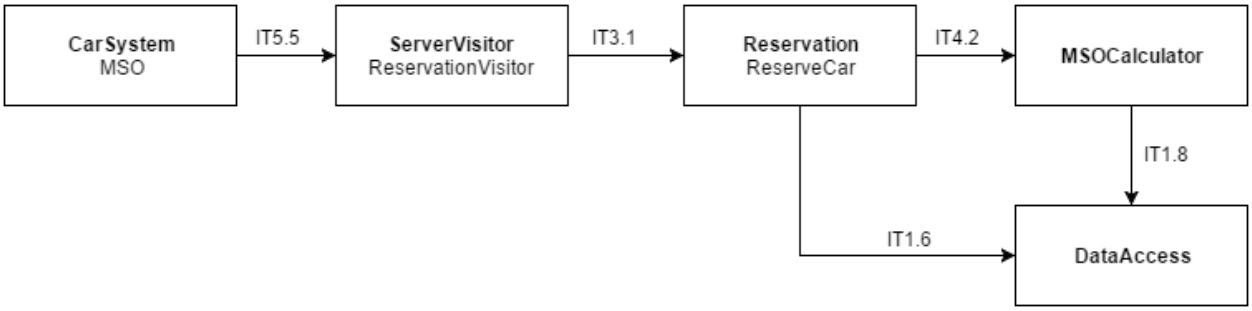
8. Lock

This functionality allows the user to lock a reserved car.



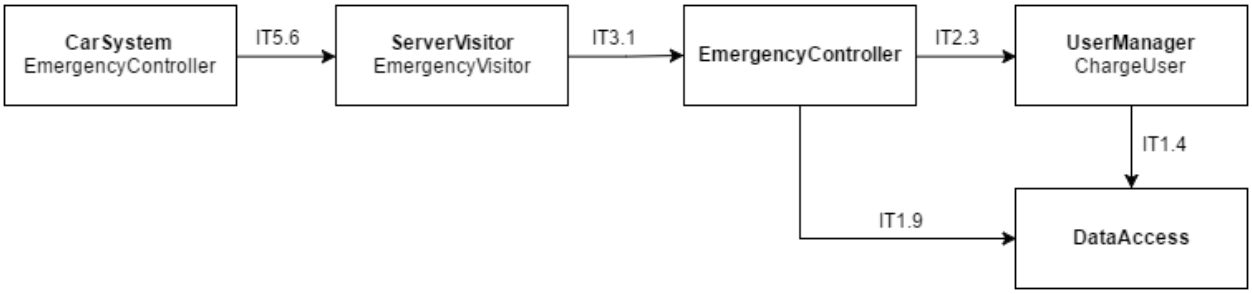
9. Money Saving Option

This functionality allows the user to set the Money Saving Option.



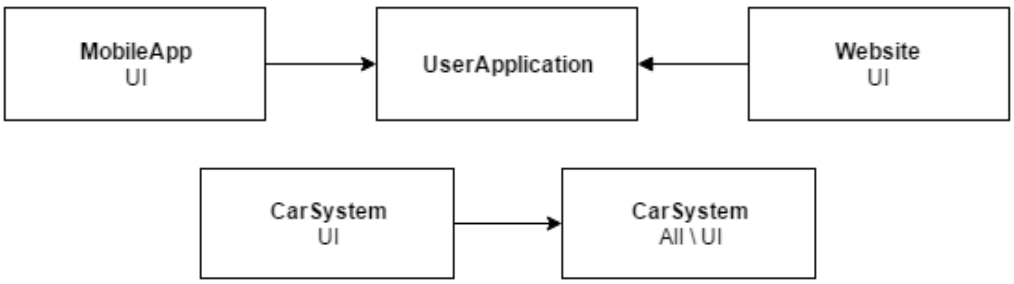
10. Emergency

This functionality activates in case of emergency.



11. User Interface

The user interfaces can be implemented at the end. It is in fact nothing more than a display of the UserApplication



3. Individual Steps Test Description

3.1 Tests Overview

These are the tests for the integration of each subcomponent. We will provide a detailed description of the tests to be performed on each subcomponents to be integrated.

In each pair of subcomponents we identified the *caller* that invokes methods of the *called*. We provide a description of the input value and the effect on the system.

- IT1. DataAccess
  - IT1.1 DataAccess, DBMS
  - IT1.2 CheckUser, DataAccess
  - IT1.3 SubscribeUser, DataAccess
  - IT1.4 ChargeUser, DataAccess
  - IT1.5 CarController, DataAccess
  - IT1.6 ReserveCar, DataAccess
  - IT1.7 LockCar, DataAccess
  - IT1.8 MSOCalculator, DataAccess
  - IT1.9 EmergencyController, DataAccess
- IT2. UserManager
  - IT2.1 ReserveCar, ChargeUser
  - IT2.2 LockCar, ChargeUser
  - IT2.3 EmergencyController, ChargeUser
  - IT2.4 SubscribeUser, ChargeUser
  - IT2.5 ReserveCar, CheckUser
- IT3. ServerVisitor
  - IT3.1 ServerVisitor, PEMS components
  - IT3.2 UserApplication, ServerVisitor
- IT4. Reservation
  - IT4.1 Reservation, UnlockVisitor
  - IT4.2 ReserveCar MSOCalculator
- IT5. CarSystem
  - IT5.1 UnlockCar, UnlockVisitor
  - IT5.2 ReserveCar, UnlockCar
  - IT5.3 NFC, UnlockCar
  - IT5.4 LockCar, LockVisitor
  - IT5.5 MSO, ReservationVisitor
  - IT5.6 EmergencyController, EmergencyVisitor

3.2 Test Description

IT1. DataAccess

IT1.1 Data Access, DBMS

DataAccess can read, create and modify some data on the database. In order to test DataAccess we have to check that the SQL queries are correctly formed and executed on the DBMS.

IT1.2 CheckUser, DataAccess

The component CheckUser asks DataAccess if the login data are valid and if the user is suspended or not.

checkUserAndPassword(String mail, String psw)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
Formally valid arguments.	True if mail and psw match with database, false otherwise.

isSuspended(User user)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A request with a user not existent in the database.	An InvalidArgumentException is raised.
Formally valid arguments.	True if user is suspended, false otherwise

IT1.3 SubscribeUser, DataAccess

The component SubscribeUser is used to create a new user.

insertPersonalInformation(User user)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A request with a mail and password already existent in the database.	An InvalidArgumentException is raised.
Formally valid arguments.	An entry containing the request data is inserted into the database.

insertPayment(Payment p)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A request with an incorrect payment information	An InvalidArgumentException is raised.
Formally valid arguments.	An entry containing the payment information is inserted into the database.

insertLicence(Licence l)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A request with a license already existent in the database.	An InvalidArgumentException is raised.
Formally valid arguments.	An entry containing the license number is inserted into the database.

Before testing the communication between these two component we make sure that the following methods works correctly.

checkPersonalInformation(User user)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A request with a user already existent in the database.	false.
a request with a user not existent in the database.	true.

Andrea Facchini
Antonio Gianola
Andrea Milanta

IT1.4 ChargeUser, DataAccess

The component ChargeUser is used to charge the user of the correct amount of money. it asks to DataAccess the payment information before the transaction and to store the transaction when it is done.

chargeUser(User user, float payment)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A request with a user not existent in the database.	An InvalidArgumentException is raised.
Formally valid arguments.	if payment is done it writes into a database the amount of money paid.

getPaymentInformation(User user)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A request with a user not existent in the database.	An InvalidArgumentException is raised.
Formally valid arguments.	return a payment information of the user.

IT1.5 CarController, DataAccess

CarController asks to DataAccess the list of available car.

getAvailableCars()	
Input	*Effects*
Nothing	It returns the list of available cars. with information like BatteryStatus and Position on the map

Before testing the communication between these two component we make sure that the following methods correctly works.

findCloseCar(GPS position)	
Input	*Effects*
A null parameter	A NullArgumentException is raised.
A position not in a GA.	An empty list of close car.
A position in a GA	A list of the close car to the position.

findCloseCar(Address position)	
Input	*Effects*
A null parameter	A NullArgumentException is raised.
A position not in a GA.	An empty list of close car.
A position in a GA	A list of the close car to the position.

IT1.6 ReserveCar, DataAccess

ReserveCar writes into a database the reservation information.

reserveTheCar(Car car, User user)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A car or user that does not exist	An InvalidArgumentException is raised.
An offline or already used car	An UnavableCarException is raised.
A user that has already reserved a car	An UnavableUserException is raised.
A suspended user	A UserSuspendedException is raised.
A correct car and user	it creates the reservation in database and return a code to be sent to the car.

checkReservationCode(User u, code C)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A car or user that does not exist	An InvalidArgumentException is raised.
A user that it has not reserved a car	An UnavableUserException is raised.
A suspended user	A UserSuspendedException is raised.
A correct car and code	true if the code match with user, false otherwise

Andrea Facchini
Antonio Gianola
Andrea Milanta

deleteReservation(User user)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A user that does not exist	An InvalidArgumentException is raised.
A correct user	it deletes a reservation into a database

unlockCar(User user, Car c)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A car or user that does not exist	An InvalidArgumentException is raised.
A user that it has not reserved a car	An UnavableUserException is raised.
A correct user and car	it set into a database that the car is unlocked.

IT1.7 LockCar, DataAccess

checkEndOfRideLocation(GPS position)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A position out of GA	An InvalidPositionException is raised and save the current car position into a database.
A position in a GA	Save the current position into a database.

Andrea Facchini
Antonio Gianola
Andrea Milanta

getRideInformation(User user)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A user that does not exist	An InvalidArgumentException is raised.
A user that it has not reserved a car	An UnavableUserException is raised.
A correct user	Read the ride information like time, position, number of People in a car and MSO from database. This informations are used to calculate ride costs

Before testing the communication between these two component we make sure that the following methods correctly works.

calculateRideCosts(Car car, User user)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A car or user that does not exist	An InvalidArgumentException is raised.
A user that it has not reserved a car	An UnavableUserException is raised.
A suspended user	A UserSuspendedException is raised.
A correct car and code	return how much the user has to pay.

IT1.8 MSOCalcolator, DataAccess

searchRA(Address a)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
An address that does not exist	An InvalidArgumentException is raised.
A correct address	it searches and return a list of RA where the user can leave the car.

Andrea Facchini
Antonio Gianola
Andrea Milanta

setMSO(User user, Address a)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A user that does not exist	An InvalidArgumentException is raised.
A user that it has not reserved a car	An UnavableUserException is raised.
A correct user and address	write on database that the user enabled the MSO option and where he is going.

IT1.9 EmergencyController, DataAccess

createEmergency(Emergency e)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A correct emergency	write the emergency on database.

setOffline(Car c)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A car that does not exist	An InvalidArgumentException is raised.
A correct car	the car is set to Offline into the database.

Before testing the communication between these two component we make sure that the following methods correctly works.

sendMessage(Emergency emergency)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A correct type of emergency	It sends an external message to OS in order to inform about the problem.

Andrea Facchini
Antonio Gianola
Andrea Milanta

IT2. UserManager

IT2.1 ReserveCar, ChargeUser
IT2.2 LockCar, ChargeUser
IT2.3 EmergencyController, ChargeUser
IT2.4 SubscribeUser, ChargeUser

Each of the previous four subcomponents calls the following method which has to work correctly.

chargeUser(User user, float payment)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A request with a user not existent in the database or negative payment.	An InvalidArgumentException is raised.
Formally valid arguments.	if payment is done returns true otherwise returns false.

IT2.5 ReserveCar, CheckUser

checkUser(User user)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A request with a user not existent in the database.	An InvalidArgumentException is raised.
Formally valid arguments.	if a User can reserve a car it returns true. It returns false otherwise.

Andrea Facchini
Antonio Gianola
Andrea Milanta

IT3. ServerVisitor

IT3.1 ServerVisitor, other PEMS components

With respect to sub-components dependencies the ServerVisitor communicates with a lot of components but its behavior is really simple. When a request is coming it understands the request's type and calls the correct submodule that can manage that request.

- UserVisitor
  - transferRequests(Request registration, User user)
  - transferRequests(Request registration, Payment p)
  - transferRequests(Request registration, License l)
- FindCarVisitor
  - transferRequests(Request findCar, GPS position)
  - transferRequests(Request findCar, Address a)
- ReservationVisitor
  - transferRequests(Request reserve, Car car, User user)
  - transferRequests(Request sensor, Sensor s)
  - transferRequests(Request mso, Address a)
- UnlockVisitor
  - transferRequests(Request unlock)
  - transferRequests(Request starting, Car c)
  - transferRequests(Request unlock, Code c)
- LockVisitor
  - transferRequests(Request lock, Car c)
  - transferRequests(Request charge, Car c)
- EmergencyVisitor
  - transferRequests(Request emergency, Car c)

The way to test this component is send some message and check if ServerVisitor call the correct function.

IT3.2 UserApplication, ServerVisitor

When the UserApplication sends commands to the PEMS the first component that manage the message is ServerVisitor.

A best way to test this communication is to do it manually for each function.

IT4. Reservation

IT4.1 Reservation, UnlockVisitor

These messages are forwarded from Reservation to CarSystem by UnlockVisitor.

UnlockDoors(Car car)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A car that is not reserved.	An InvalidArgumentException is raised.
A correct car.	the doors of the car are unlocked

unlockEngine(Car car)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A car that is not reserved.	An InvalidArgumentException is raised.
A correct car.	the ignition of the car is unlocked

Before testing the communication between these two component we make sure that the following methods correctly works.

startReservationTimer()	
Input	*Effects*
nothing	We have to be sure that the reservation timer starts immediately when the reservation process is finish.

startRideTimer()	
Input	*Effects*
nothing	We have to be sure that the ride timer starts immediately when the starting code is checked.

Andrea Facchini
Antonio Gianola
Andrea Milanta

IT5. CarSystem

Before testing the communication between CarSystem and Server we make sure that all the sensors in the car work correctly and can send or receive information from CarSystem.

These sensors are:

- Doors status (locked, unlocked)
- Ignition status (enabled, disabled)
- Number of people in the car
- Battery status
- Touch display

IT5.1 UnlockCar(CarSystem), UnlockVisitor

When a car is unlocked the CarSystem sends all needed information to the UnlockVisitor sub-component. We identify this kind of tests on this component.

setBatteryStatus(Car car, Sensor Battery)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A correct car	It sends to the reservation component the current battery status.

setNumberOfSeats(Car car, Sensor passenger)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A correct car	It sends to the reservation component the current number of passengers in the car.

checkStartingCode(Car car, Code code)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A correct car and code	It checks if the starting code is correct and if it is true unlock the ignition.

setUnlockDoors(Car car, Sensor doors)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A correct car	it sends to reservation that the doors are unlock and it start the ride timer.

IT5.2 UnlockVisitor, UnlockCar

UnlockDoors()	
Input	*Input*
Nothing	Car doors are unlocked.

IT5.3 NFC, UnlockCar

These two components interact with NFC. The best way to test the NFC communication between user application and car system is to do it manually.

IT5.4 LockCar, LockVisitor

sendPosition(GPS position)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A correct GPS position	the LockVisitor call Reservation component on the server with the correct parameter.

isRecharging(Car car)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A correct car	Send true if the car is recharging. False otherwise

IT5.5 MSO, ReservationVisitor

MSORequest(Car car, Address address)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A correct car	Send a message to server to request the MSO, it returns to a car the list of available RA.

enableMSO(Car car, Address address)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A correct car	Send a message to server to enable MSO, it enables the MSO for this reservation.

IT5.6 EmergencyController, EmergencyVisitor

sendEmergency(Emergency emergency, Car car)	
Input	*Effects*
A null parameter.	A NullArgumentException is raised.
A correct car and emergency	Send to the system a message with the information about the error and the car.

Andrea Facchini
Antonio Gianola
Andrea Milanta

4. Tools and Test Equipment

4.1 Testing Tools

In order to accomplish the integration test, the following tools are required.

Note that in this document takes in consideration only tools for integration testing; tools for unit testing are not discussed.

Arquillian

Arquillian is an integration testing framework for Java EE built on top of JUnit. It provides all the instruments needed for testing the Java Application Server and the interactions between the containers and the database.
The tool and the full documentation are accessible at: http://arquillian.org/

Manual Testing

In order to assure the correct behavior of those components of the system for which is complex to build an automatic test, like the user interface, a manual test is preferred.

JMeter

JMeter is an Apache testing tool for testing the performance of a web application. It will be used at the end of the integration test for evaluating the overall performance on the system. It allows also to simulate different loads for the service, and it will be useful to verify the correct behavior of the system in different scenarios.

4.2 Testing Equipment

For testing the correct behaviour of the system in a real world scenario, the following equipment is required:

- Server Application deployed on Amazon Elastic Compute Cloud (EC2).
- 2 Cars equipped with the on-board software and hardware.
- 3 Smartphones, one for each main OS: Android, iOS and Window Phone.
- 3 Tablet, one for each main OS: Android, iOS and Window Phone.
- A computer equipped with all the main browsers: Google Chrome, Mozilla Firefox, Safari and Microsoft Edge.

5. Program Stubs and Test Data

5.1 Program Stubs

As explained in chapter 2 (Integration Strategy) the integration testing follows the Thread Strategy. This means that the system is tested “by functionality”, and a functionality is tested only when it’s fully “reachable”, so it doesn’t need any particular testing over-structure.

A simplified user interface may be required though to simplify the checks of outputs.

5.2 Test data

In order to accomplish the test illustrated in this document, the following data are required, ordered by test of chapter 3.

Note: names starting with a capital letter are intended as class names).

- DataAccess:
  - a null object.
  - Users with valid mail and password.
  - Users with invalid mail and password (in all the combination: one valid and one invalid field, both invalid).
  - Users with correct mail and password.
  - Users with incorrect mail and password (in all the combination: one correct and one incorrect field, both incorrect).
  - Users suspended.
  - Users not suspended.
  - Users not existing in the database.
  - Users with correct payment information.
  - Users with incorrect payment information.
  - Users with valid driving license number.
  - Users with invalid driving license number.
  - Cars with valid parameters.
  - Cars in the same GA of a valid User
  - Cars not in the same GA of a valid User.
  - Cars offline.
  - Cars online.
  - Users that are currently reserving a Car.
  - Users that aren't currently reserving a Car.
  - Cars not existing in the database.
  - Cars locked.
  - Cars unlocked.
  - Cars with position outside a GA.
  - Cars with position inside a GA.
  - Address not existing in the database.
  - Address existing in the database.
  - Emergency with correct parameters.

- User Manager
  - a null object.
  - Users existing in the database.
  - Users not existing in the database.

- Server Visitor
  - a complete set of Requests.

- Reservation
  - a null object.
  - Cars reserved.
  - Cars not reserved.

- CarSystem
  - Cars with all the different Battery Status.
  - Cars with a different number of passengers.
  - correct Car Code.
  - incorrect Car Code.
  - Car in charging.
  - Car not in charging.
  - correct Address.
  - incorrect Address.
  - correct Emergency.

6. Effort Spent

Work Division

section	Paragraph	author	revisioned by
Introduction			
	Revision History	Milanta	Gianola
	Purpose	Milanta	Facchini
	Scope	Milanta	Facchini
	Glossary	Facchini	Milanta
	reference documents	Facchini	Milanta
	Overview	Facchini	Milanta
Integration Strategy			
	Entry Criteria	Milanta	Gianola
	Elements to be Integrated	Milanta	Gianola
	Integration Testing Strategy	Milanta Gianola	Everyone
	Sequence of Component Integration	Gianola	Milanta
Individual Steps Test Description			
		Gianola	Milanta
Tools and Test Equipment			
	Testing Tools	Facchini	Milanta Gianola
	Testing Equipment	Facchini	
Program Stubs and Test Data			
	Program Stubs	Facchini	Gianola Milanta
	Test Data	Facchini	
Effort Spent			
		Facchini	Milanta Gianola

Working Hours

	Week I							Week II							Week III							Week IV							Total
	19	20	21	22	23	24	25	26	27	28	29	30	31	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Antonio Gianola										1		1										3	4				2	2	13
Andrea Facchini			1							1					2				3				2				1	2	12
Andrea Milanta										1												3	4				2	2	12