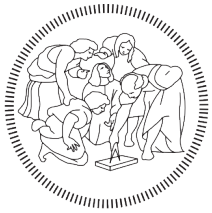


POLITECNICO
MILANO 1863

Integration Test Plan Document for PowerEnJoy

Daniele Riva* Marco Sartini†

January 15, 2017

version 1.0

*matr. 875154
†matr. 877979

Contents

1 Introduction

1.1 Purpose

This Integration Test Plan Document has the purpose to establish an acceptable program of tests to be performed on the developed modules and on the overall system.

The aim of the tests is to solicit the system and try to find incompleteness, bugs, modules compatibility problems, also minimizing the possibility to obtain critical and wrong states. Specifically, this document gathers a number of integration tests to attempt to ensure the rightness of the behaviors among components.

1.2 Scope

The project *Power EnJoy* is a platform based on mobile and web application thought to offer a car sharing service with electrical powered cars.

1.3 Definitions, acronyms, abbreviations

RASD requirements analysis and specification document;

DD design document;

DBMS data base management system;

Arquillian is a Java Integration Test framework. Details at http://www.arquillian.org;

Mockito is a mocking framework. Details at http://site.mockito.org;

JUnit is a simple framework to write repeatable tests. Based on xUnit architecture. Details at http://junit.org.

1.4 Reference documents

- RASD v1.0 available at https://github.com/marcosartini/PowerEnJoy/blob/master/releases/rasdPowerEnJoy.pdf
- DD v1.0 available at https://github.com/marcosartini/PowerEnJoy/blob/master/releases/dd.pdf

Revision history

Name	Date	Reason For Changes	Version
Marco e Daniele	10/01/2017	Starting	1.0

2 Integration strategy

2.1 Entry criteria

Before the integration testing phase begin, all the modules have to be properly unity tested. This means that every single module should work right independently by the others, that is, it should return the expected results related to the input provided. To speed up the starting of this step, it is not mandatory to have all the modules fully developed, but it is sufficient to dispose of the subparts that compose a thread. In some cases, although, a ordered schedule is requested. In particular, all the methods listed and described in the DD should be tested.

2.2 Elements to be integrated

The components to be integrated are:

1. MapManager, to be integrated with the CarsManager;
2. RentalManager, to be integrated with CarsManager, KeepAsideManager, OnBoardSystemManager, UserManager, PaymentManager and DataManager;
3. OnBoardSystemManager, to be integrated with the RentalManager;
4. ReservationManager, to be integrated with CarsManager, PaymentManager, UserManager RentalManager and DataManager;
5. KeepAsideManager, to be integrated with RentalManager, CarsManager and DataManager;
6. IssueManager, to be integrated with the UserManager, CarsManager and DataManager.

The other components are standalone, or at least they only serve the other components. To be precise, the ones which do not consult nor delegate to other components except the Data Manager are: UserManager, SettingsManager and PaymentManager.

2.3 Integration testing strategy

The modules are widely related each other. This means that the more traditional testing approaches are a bit overcharged by extra stubs and drivers. This consideration leads us to invest in a more efficient test process.

Then, we opt for the *thread approach*.

We adopt also a critical-module-first approach before actually start the thread testing, to proper check the rightness of the interaction between DBMS and DataManager, between all the components and the DataManager.

Benefits granted by the thread approach allows, as soon as the methods and classes related to a thread are ready developed, to be immediately tested. It is not necessary that a component is fully developed, but it is enough that are developed the involved subparts in the thread.

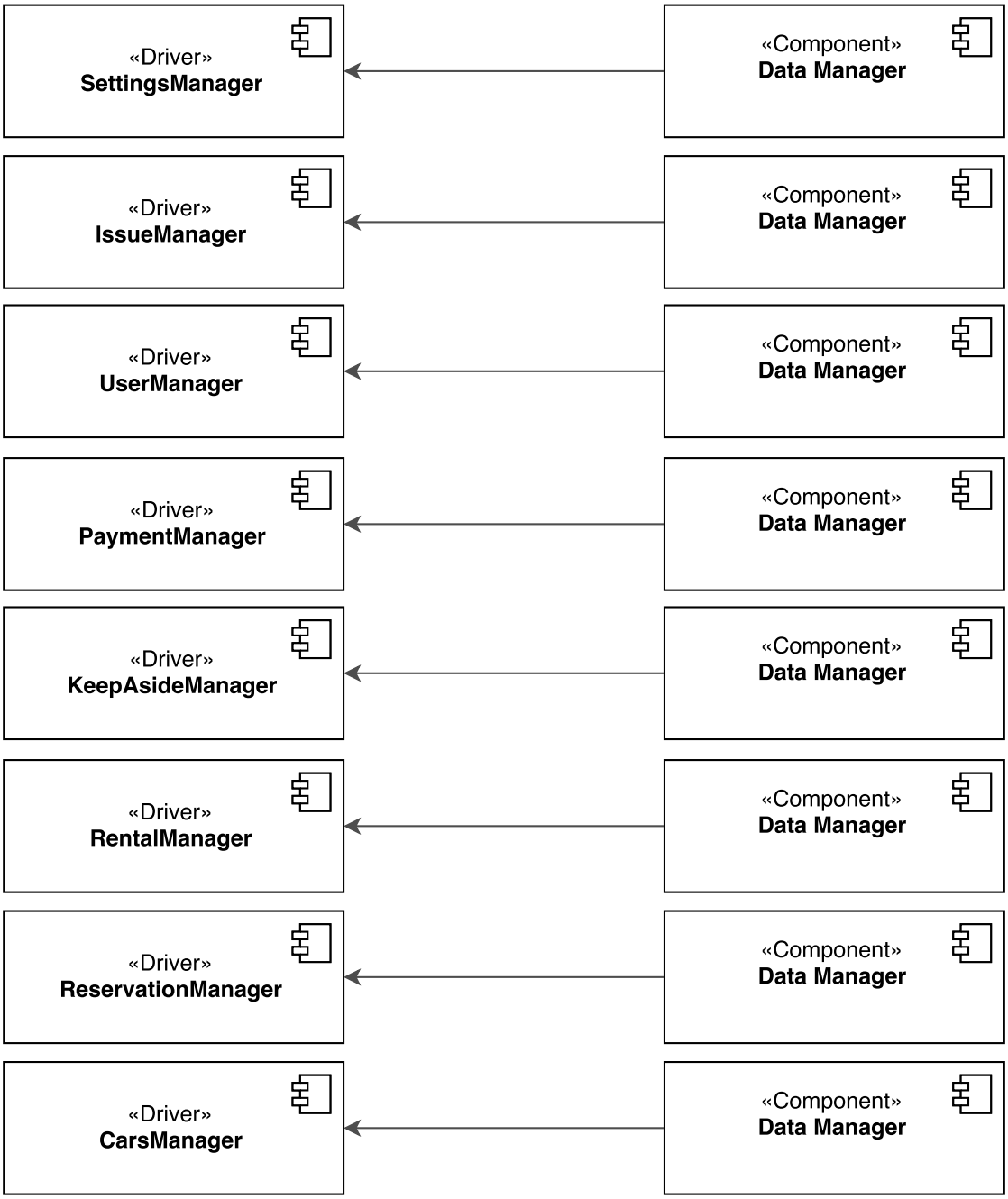
In addition, less if none stubs and drivers are required and if a component in a thread doesn't work, it is rational to exclude the subparts already tested in the other threads simplifying the resolution.

2.4 Sequence of function integration

2.4.1 DBMS and DataManager

This test should be the first to be performed because the entire system is basically based on the data manipulation. All the thread considered below write and read data from the database through the DataManager.

To test the DataManager, the component will be represented by drivers, one for each of them. The drivers will perform typical queries to check the correctness of the results.



2.4.2 Threads

Every identified thread is briefly described to clarify the its purpose and which components methods are involved. In the image, the colors stress the parts of the components which contribute to build up a thread (user-visible functionality). Some methods are shared by more than one thread, because many thread might need to employ the same

methods.

Colors caption:

yellow indicates the Rental functionality;

orange indicates the Reservation functionality;

dark blue indicates the End of the reservation functionality;

pink indicates the timeout triggered end of reservation functionality;

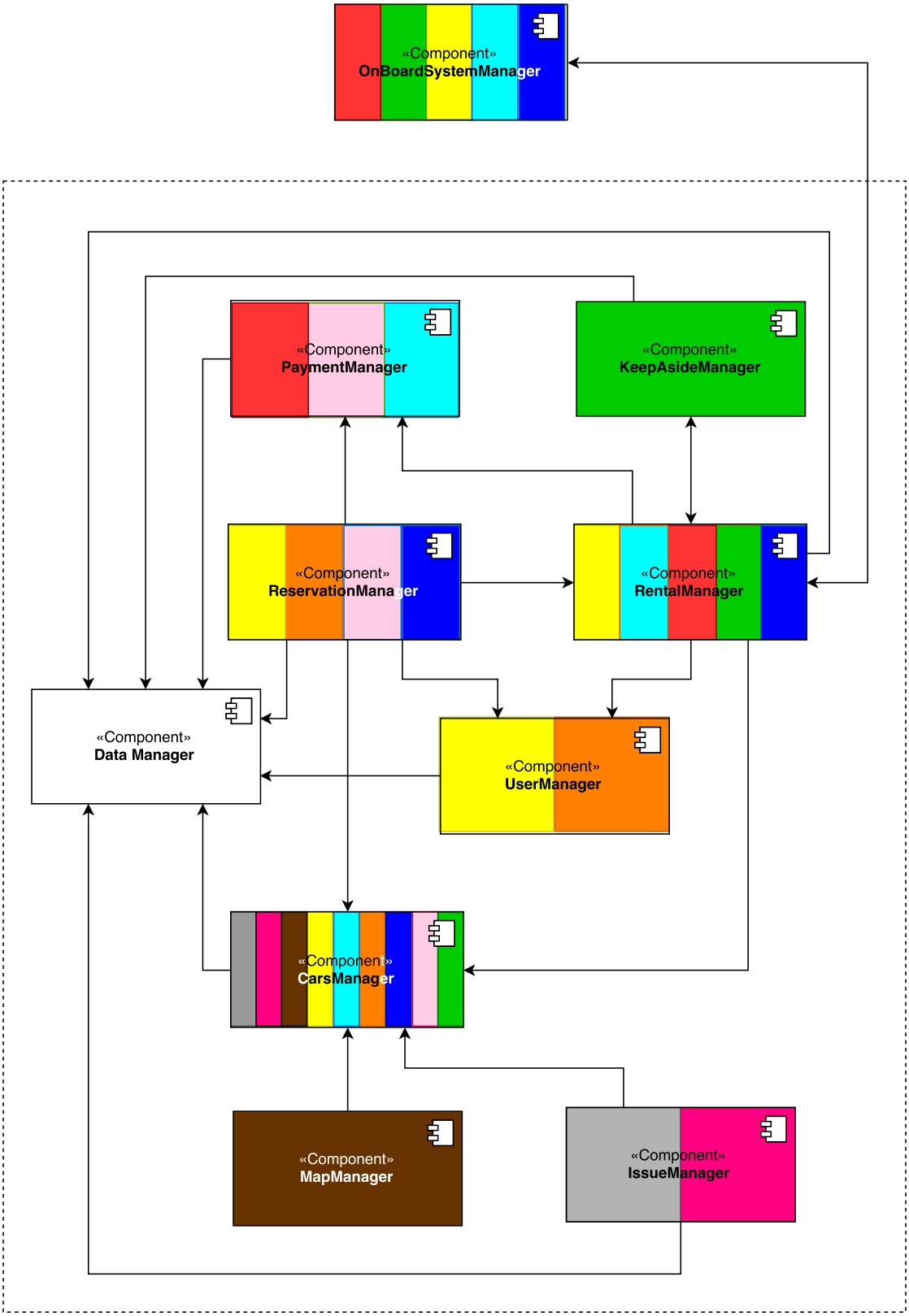
green indicates the Keep Aside functionality;

red indicates the start Rental functionality;

light blue indicates the end of the rental and related billing and payment functionality;

brown indicates the Map functionality;

gray and rose indicate the Issue functionality;


«Component»
OnBoardSystemManager
«Component»
PaymentManager
«Component»
KeepAsideManager
«Component»
ReservationMana ger
«Component»
RentalManager
«Component»
Data Manager
«Component»
UserManager
«Component»
CarsManager
«Component»
MapManager
«Component»
IssueManager

Threads do no need to be tested and verified in a particular order because of the inherent feature of independence, but we suggest to adopt the following:

1. Issues;
2. Map;
3. Rental;
4. Reservation;
5. Start rental;
6. End reservation;
7. Keep Aside;
8. End rental.

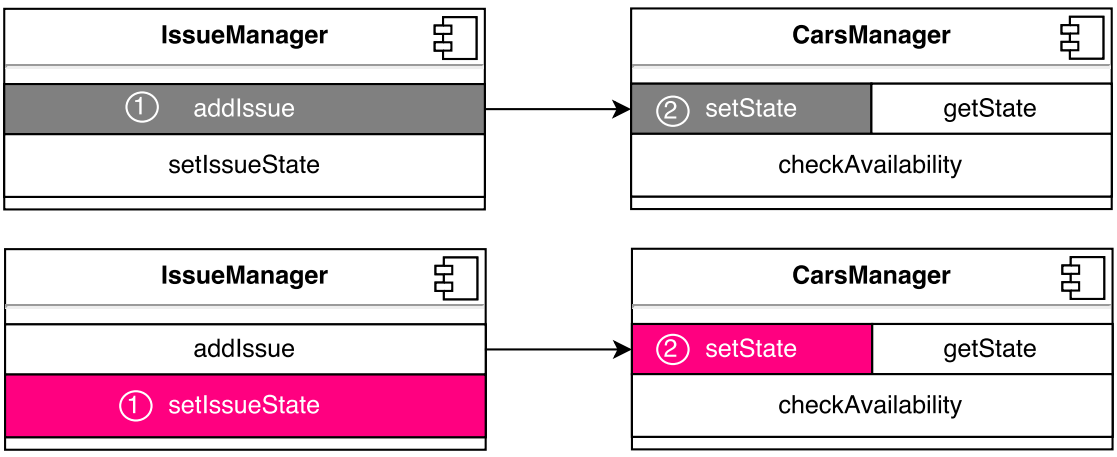
Note on the *OnBoardSystemManager*: this component, despite in the reality will be installed on the cars, is tested as it were local, by setting up the test environment at this purpose. This because test the real car means to have the car ready, and also the communication infrastructure working.

Issues

The two components to be integrated are the *IssueManager* and the *CarsManager*.

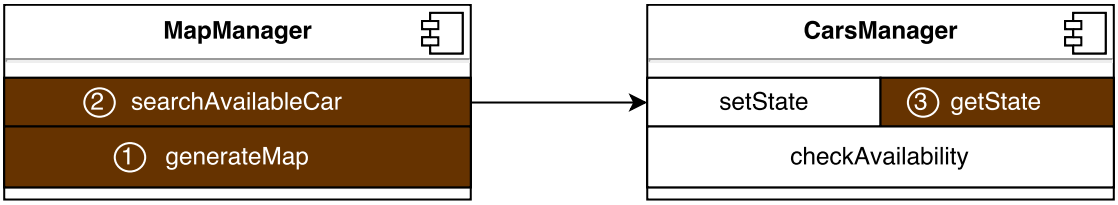
A request to add an issue is sent by the user: the *addIssue* method is activated, which is in charge to store an issue in the database, then it involves the *CarsManager* who is in charge to update the state of the car (which is received as parameter in the original request).

If no exceptions are raised, the *IssueManager* returns successfully to the caller.



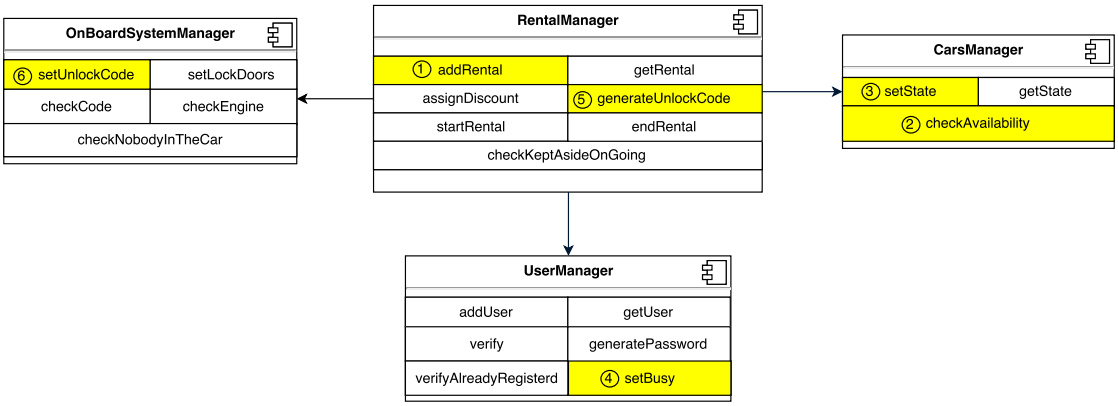
Map

It integrates with the *CarsManager* while building the map, to get the only available cars to put into that map.



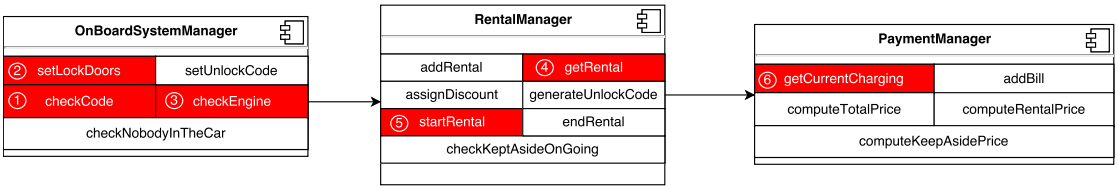
Rental

When a car has been chosen for a rental, and is confirmed, a request to apply for a rental is sent by the client. It is activated the *RentalManager* via the *addRental* method: the method is in charge to verify the availability of the car, thanks to the *CarsManager* respective query method; then in case of positive answer, the *CarsManager* is asked to update the car state; at the end, if no raised exception, the component generates an unlock code which is annotated in the *OnBoardSystemManager* via the *setUnlockCode*. Again if no raised exceptions, the unlock code is also returned to the client.



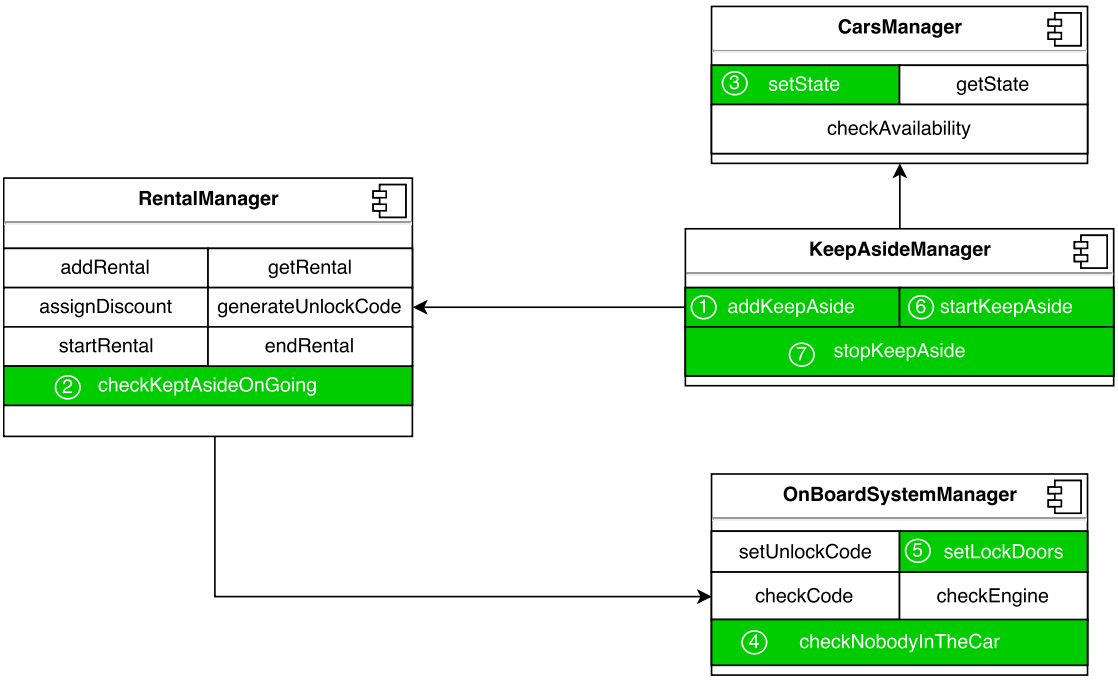
Unlock & Start

This thread is launched by the request of verifying the code. Once the code locally matches, the OnBoardSystem opens the door and listens for the engine to power on. In that moment, a request to start a rental is sent to the server, who refers to the *RentalManager*. The *RentalManager* stores on the database the data and it is ready to report the charging fees to the user.



Keep aside

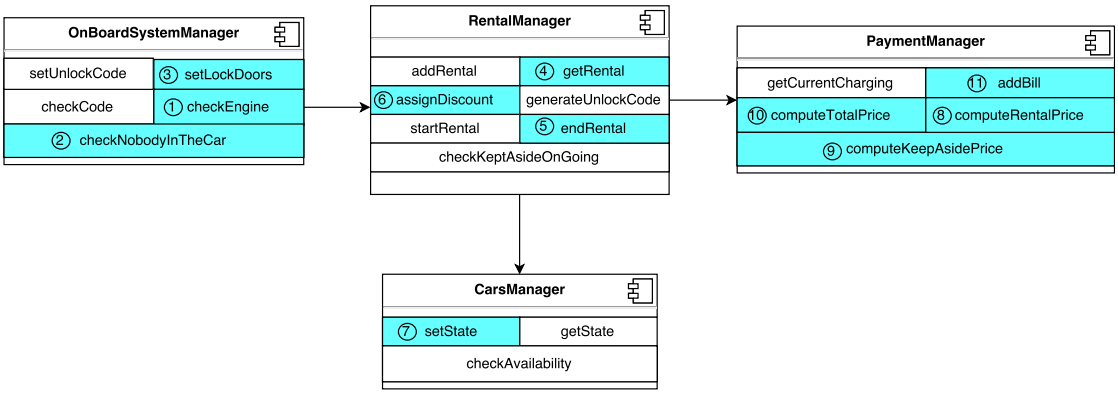
When a "keep aside" request is delivered to the server, the *KeepAsideManager* is activated, via the *addKeepAside* method. That method contacts the *CarsManager* to update the state of the car and, if no exceptions, starts the "keep aside" interval storing data in the database. If no exceptions, it returns to the *OnBoardSystemManager*, which cares to check the car to be empty and locks the doors. When a keep aside is going to be stopped, the user send a request to stop the "keep aside": the *KeepAsideManager* is involved via the method *stopKeepAside*which refers to the *CarsManager* to get the state updated and if no exceptions, the doors are unlocked by the *OnBoardSystemManager*.



End rental

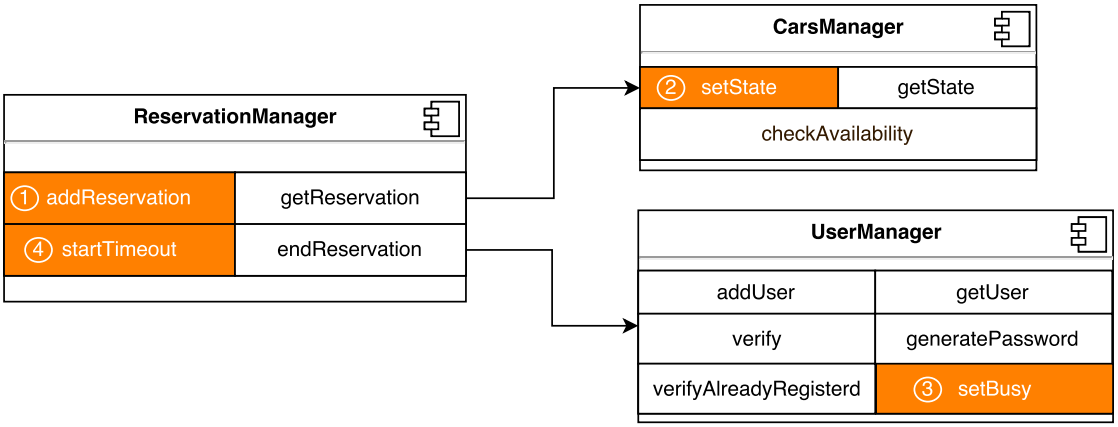
When a rental ends, the *OnBoardSystemManager* it is unawakened because the engine turn off, then it checks the passengers and is mandatory that the number is 0. So it locks the doors and invokes the *RentalManager* to end the rental via the *endRental* method. The *RentalManager* checks and eventually assigns discounts invoking its method *assignDiscount*, then the *PaymentManager* is involved to provide a bill and interact with the external Payment Handler.

Furthermore, the car state is updated by the *CarsManager*. There is a full return from the stack to reassure the *OnBoardSystemManager*.



Reservation

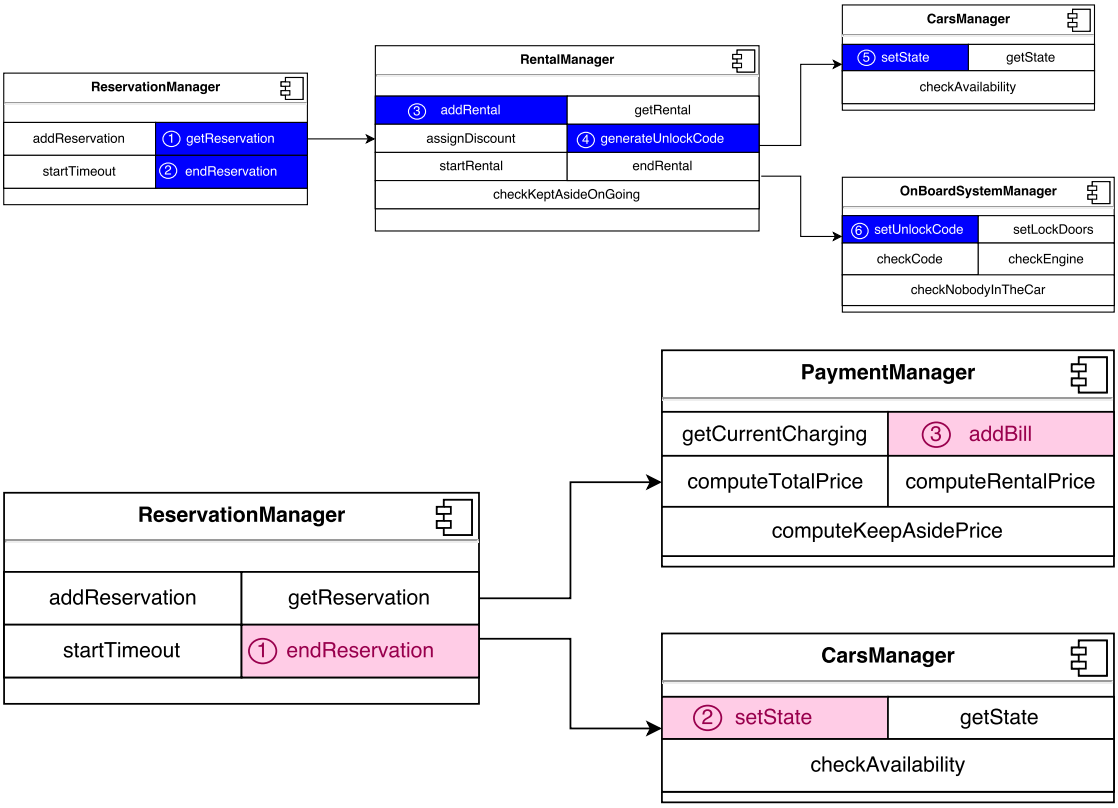
A car is chosen by the user, and is given to the *ReservationManager* who cares to add the reservation and to start the timeout. The *CarsManager* is at this point involved to have the car state updated; also the *UserManager* is involved to have the user status updated. At the end, the *ReservationManager* triggers the timeout.



End reservation & start rental

Case 1: A user decides to regularly begin a rental of his reserved car.

Case 2: If the reservation timeout expires, the reservation ends, so the *ReservationManager* will free the car from the reservation and will proceed to bill.


ReservationManager
addReservation
① getReservation
startTimeout
② endReservation
RentalManager
③ addRental
getRental
assignDiscount
④ generateUnlockCode
startRental
endRental
checkKeptAsideOnGoing
CarsManager
⑤ setState
getState
checkAvailability
OnBoardSystemManager
⑥ setUnlockCode
setLockDoors
checkCode
checkEngine
checkNobodyInTheCar
PaymentManager
getCurrentCharging
③ addBill
computeTotalPrice
computeRentalPrice
computeKeepAsidePrice
ReservationManager
addReservation
getReservation
startTimeout
① endReservation
CarsManager
② setState
getState
checkAvailability

3 Individual steps and test description

A schematic description of the test cases to be performed in order to meet the objectives of the integration test phase.

3.1 Data Manager

Test case identifier	ID0T1
Test items	DataManager $\longrightarrow$ DBMS
Input specification	Queries (insert, update, delete) on database tables
Output specification	Correct result of each query
Environmental needs	DBMS

Test case identifier	I0T1
Test items	Driver UserManager $\longrightarrow$ DataManager
Input specification	Calls method of UserManager to add new user or update data user
Output specification	DataManager generates queries on User table
Environmental needs	UserManager driver

Test case identifier	I0T2
Test items	Driver CarsManager $\longrightarrow$ DataManager
Input specification	Calls method of CarsManager to update cars State
Output specification	DataManager generates queries on Cars table
Environmental needs	CarsManager driver

Test case identifier	I0T3
Test items	Driver RentalManager $\longrightarrow$ DataManager
Input specification	Calls method of RentalManager to add new rental or update data of existing rental
Output specification	DataManager generates queries on Rental table
Environmental needs	RentalManager driver

Test case identifier	I0T4
Test items	Driver ReservationManager ⟶ DataManager
Input specification	Calls method of ReservationManager to add new reservation or update data of existing reservation
Output specification	DataManager generates queries on Rental table
Environmental needs	ReservationManager driver

Test case identifier	I0T5
Test items	Driver KeepAsideManager ⟶ DataManager
Input specification	Calls method of KeepAsideManager to add new keepAside of a rental or update data of existing keepAside
Output specification	DataManager generates queries onKeepAside table
Environmental needs	KeepAsideManager driver

Test case identifier	I0T6
Test items	Driver PaymentManager ⟶ DataManager
Input specification	Calls method of PaymentManager to add a new bill
Output specification	DataManager generates queries on Payment table
Environmental needs	PaymentManager driver

Test case identifier	I0T7
Test items	Driver IssueManager ⟶ DataManager
Input specification	Calls method of IssueManager to add a new issue or update an issue state
Output specification	DataManager generates queries on Payment table
Environmental needs	IssueManager driver

Test case identifier	I0T8
Test items	Driver SettingsManager ⟶ DataManager
Input specification	Calls methods of SettingsManager to manage prices
Output specification	DataManager generates queries on Settings table
Environmental needs	SettingsManager driver

3.2 Issue

Test case identifier	I1T1
Test items	Issue Manager ⟶ Car Manager
Input specification	Call the addIssue method
Output specification	The issue is added in the queue of the issue to solve; the car state is changed in ...

Below the description of the involved method calls.

IssueManager ⟶ addIssue(user, car, description, phoneNumber)	
One or more parameters are null	A NullArgumentException is raised
Description is a empty string	An InvalidArgumentException is raised, the problem must be specified
Phone number is an incorrect number	An InvalidArgumentException is raised
Set of valid parameter	Creates an instance of issue with IssueState as *ToSolve* and then writes it in the database

CarsManager ⟶ setState(car, state)	
One or more parameters are null	A NullArgumentException is raised
Incorrect parameter: a car or a state that not exists	An InvalidArgumentException is raised
Valid Parameters	Car state is changed into the new state and the car record in the database is updated

3.2.1 Maintenance solve issues

Test case identifier	I1T2
Test items	Issue Manager ⟶ Car Manager
Input specification	A call to setIssueState
Output specification	The car state is updated accordingly to the issue outcome

IssueManager ⟶ setIssueState(issue, man, state)	
One or more parameter are null	A NullArgumentException is raised
Invalid parameter: the issue is already solved or is being solved	An IllegalIssueException is raised
Invalid parameter: the man is already busy in another work	An IllegalIssueException is raised
Valid parameter	Assigns the state parameter (InResolution or Solved) to the issue state and updates the database

3.3 Map

Test case identifier	I2T1
Test items	MapManager ⟶ CarsManager
Input specification	Call to the generateMap method
Output specification	The generated map contains the available cars, if any

Below the description of the involved method calls.

MapManager ⟶ searchAvailableCar(listCar)	
Parameter is null	A NullArgumentException is raised
Empty List	Returns empty list, no cars are available
Valid parameter: list with no available cars	Returns a empty list
Valid parameter: list contains available cars	Returns a list with only available cars

MapManager ⟶ generateMap(listCar,position)	
One or more parameter are null	A NullArgumentException is raised
Empty List	An IllegalMapException is raised
A invalid position	An IllegalArgumentException is raised
Valid parameters: no empty List and valid position	Returns a list of the available cars into the default distance

3.4 Rental

Test case identifier	I3T1
Test items	RentalManager ⟶ CarsManager, UserManager, OnBoardSystemManager
Input specification	Call to the addRental method
Output specification	The availability of the car is checked and the setState is called, then the user is set to busy, an unlock code is generated and sent to the car.

Below the description of the involved method calls.

RentalManager ⟶ addRental(user, car)	
One or more parameters are null	A NullArgumentException is raised
Car is not available (checkAvailability returns false)	An InvalidRequestException is raised, the car is unavailable
User is already busy	An InvalidRequestException is raised, user is renting or reserving a car in that moment
Set of valid parameter	Creates an instance of new rental; the car state is changed (from available state to rental state), it generates unlock code and then the rental is written to the database

RentalManager ⟶ generateUnlockCode(rental)	
Valid Parameter	Generates a code to unlock the car and associates it to the rental
Parameter is null	A NullArgumentException is raised
Incorrect parameter: finished rental	An InvalidArgumentException is raised

CarsManager ⟶ checkAvailability(car)	
Parameter is null	A NullArgumentException is raised
Car state is not 'AVAILABLE'	Function returns False
Car state is 'AVAILABLE'	Function returns True

UserManager ⟶ setBusy(user, state)	
One or more parameters are null	A NullArgumentException is raised
Valid parameter	If state=true, it sets the user busy flag to *true* else if state = false, it sets the user busy flag to *false*

OnBoardSystemManager ⟶ setUnlockCode(code)	
Parameter is null	A NullArgumentException is raised
Incorrect parameter: empty String code	An InvalidArgumentException is raised
Valid parameter	Sets code to unlock the car

3.5 Unlock & Start rental

Test case identifier	I4T1
Test items	OnBoardSystemManager ⟶ RentalManager; RentalManager ⟶ PaymentManager;
Input specification	A code is injected in the OnBoardSystemManager, to have the code checked. Power on the engine.
Output specification	If the code is correct, the doors unlock and when the engine powers on the rental starts.

Below the description of the involved method calls.

OnBoardSystemManager ⟶ checkCode(code)	
Parameter is null	A NullArgumentException is raised
Code is incorrect (not equal to already saved code)	Returns false: unlock code is wrong
Valid parameter	Returns true: code saved and code parameter are equal

OnBoardSystemManager ⟶ setLockDoors(state)	
Parameter is null	A NullArgumentException is raised
State value: False	Sets the doors locked (false); car can't be opened
State value: True	Sets the doors unlocked (true); car is opened

OnBoardSystemManager ⟶ checkEngine()	
No parameter	Simulates the engine state: if the engine state is set to "run" it returns *true* else it returns *false*

RentalManager ⟶ getRental(car)	
Parameter is null	A NullArgumentException is raised
Invalid parameter: the car is not involved in any rental	An IllegalArgumentException is raised
Valid parameter	Gets the rental ongoing with that car

RentalManager ⟶ startRental(rental)	
Parameter is null	A NullArgumentException is raised
Invalid parameter: ended rental	An IllegalArgumentException is raised
Valid parameter	Sets the time when the rental starts and the value stored in the database is updated

PaymentManager ⟶ getCurrentCharging (rental)	
Parameter is null	A NullArgumentException is raised
Invalid parameter: rental is already finished	An IllegalArgumentException is raised
Valid parameter	Gets total charging of ongoing rental (included all keepAsides and discounts)

3.6 Keep aside

Test case identifier	I5T1
Test items	KeepAsideManager ⟶ RentalManager, CarsManager; RentalManager ⟶ OnBoardSystemManager;
Input specification	Call to the addKeepAside method
Output specification	A keep aside is applied to the car. and the doors are locked

Below the description of the involved method calls.

Test case identifier	I5T2
Test items	KeepAsideManager ⟶ RentalManager, CarsManager; RentalManager ⟶ OnBoardSystemManager;
Input specification	Call to the stopKeepAside method
Output specification	The doors are unlocked, and the car is ready to restart a rental.

KeepAsideManager ⟶ addKeepAside(rental)	
Parameter is null	A NullArgumentException is raised
Invalid parameter: keep aside is not allowed for this rental at this moment	An IllegalKeepAsideException is raised
Invalid parameter: checkKeepAsideOnGoing returns false	An IllegalKeepAsideException is raised, a keep aside is ongoing
Valid parameter	Creates a new instance of keepAside and adds it to the rental, then the keepAside is written to the database

KeepAsideManager ⟶ startKeepAside(rental)	
Parameter is null	A NullArgumentException is raised
Invalid Parameter : rental has no keepAside to start, rental has only finished keepaside	A IllegalKeepAsideException is raised
Valid Parameter	Sets time when keepAside started and the value in the database is updated

KeepAsideManager ⟶ stopKeepAside(rental)	
Parameter is null	A NullArgumentException is raised
Invalid Parameter : rental has no keepAside ongoing, rental has only finished keepaside	A IllegalKeepAsideException is raised
Valid parameter	Set time that keepAside ends, set keepaside as finished and database value is updated

RentalManager* ⟶ *checkKeptAsideOnGoing(rental)	
Parameter is null	A NullArgumentException is raised
Valid parameter	Return true if there is a keepaside on going, otherwise return false

OnBoardSystemManager* ⟶ *checkNobodyInTheCar()	
No parameter	If there is nobody in the car, calls the *lockDoors*, otherwise it does nothing

3.7 End rental

Test case identifier	I6T1
Test items	OnBoardSystemManager ⟶ RentalManager; RentalManager ⟶ PaymentManager, CarsManager;
Input specification	The engine of the car is powered off
Output specification	The doors are locked, the bill is stored
Environmental needs	The stub to simulate the Payment Handler

Below the description of the involved method calls.

RentalManager* ⟶ *endRental(rental)	
Parameter is null	A NullArgumentException is raised
Valid parameter	Calls the *assignDiscount* function and assigns the value to the rental, then sets the time when rental is ended and updates the database value. Then calls the method to update the car state and to compute the final price

RentalManager* ⟶ *assignDiscount(rental)	
Parameter is null	A NullArgumentException is raised
Invalid parameter : not finished rental	A IllegallArgumentException is raised
Valid parameter : finished rental	Assigns to the rental the list of the possible discount that user may receive. Now rental has a list (empty or with one or more value) containing discount values.

PaymentManager ⟶ computeKeepAsidePrice(rental)	
Parameter is null	A NullArgumentException is raised
Valid parameter: rental with empty List of keepAside	Returns 0 (zero)
Valid parameter: rental with no empty List of KeepAside	Returns the total cost of all keepAsides: this cost is computed as the price per minutes of keepAside times the duration in minutes

PaymentManager ⟶ computeRentalPrice(rental)	
Parameter is null	A NullArgumentException is raised
Invalid parameter: a rental is not ended yet	An IllegalArgumentException is raised
Valid parameter: ended rental	Returns only the cost of the rental: this cost is computed as the price per minutes of rental times the duration in minutes

PaymentManager ⟶ computeTotalPrice(rental)	
Parameter is null	A NullArgumentException is raised
Invalid parameter: a rental is not ended yet	An IllegalArgumentException is raised
Valid parameter: ended rental	Returns the total cost of the rental ⟶ sum rental cost and keepAside cost and applies discounts: it add the maximum value of discounts (overcharge) and the minimum value of discounts (best discount)

PaymentManager ⟶ addBill(rental)	
Parameter is null	A NullArgumentException is raised
Invalid parameter: a rental is not ended yet	An IllegalArgumentException is raised
Valid parameter: ended rental	Creates a bill of the rental and writes it to the database; then notifies the amount of the bill to the external Payment Handler

3.8 Reservation

Test case identifier	I7T1
Test items	ReservationManager ⟶ CarsManager, UserManager
Input specification	Call to the addReservation method
Output specification	The user is set to busy, the reservation is stored in the database, the timeout is triggered, the chosen car is correctly reserved

Below the description of the involved method calls.

ReservationManager ⟶ startTimeout(reservation)	
No parameter	Starts the hour-long countdown

ReservationManager ⟶ addReservation(user, car)	
One or more parameters are null	A NullArgumentException is raised
Car is not available (checkAvailability returns false)	An InvalidRequestException is raised, car is unavailable
User is already busy	An InvalidRequestException is raised, user is renting or reserving a car in this moment
Set of valid parameter	Creates an instance of reservation, calls setState of car to change it (from available state to reserve state), then the reservation is written to the database

3.9 End reservation

Test case identifier	I8T1
Test items	ReservationManager ⟶ RentalManager; RentalManager ⟶ OnBoardSystemManager, CarsManager;
Input specification	A call to getReservation
Output specification	The car is now in a rental state, with the associate unlock code, ready to be unlocked

Below the description of the involved method calls.

ReservationManager ⟶ getReservation(user)	
Parameter is null	A NullArgumentException is raised
Invalid parameter: the user is not involved in any reservation	An IllegalArgumentException is raised
Valid parameter	Gets the reservation related to the user

ReservationManager ⟶ endReservation(reservation)	
Parameter is null	A NullArgumentException is raised
Invalid parameter: reservation is already finished	An IllegalArgumentException is raised
Valid parameter	Set the time that reservation is finished and update database value.

3.10 Registration

This is a supplementary test description which refers to the unit tests to be carry out on the User Manager.

UserManager ⟶ verifyAlreadyRegistered(name, surname, mail, cf)	
One or more parameters are null	A NullArgumentException is raised
Mail already used	Return false, user already registered
CF already used	Return false, user already registered
An incorrect mail address or name or surname	An InvalidArgumentException is raised
An incorrect CF (no related person)	An InvalidArgumentException is raised
Set of valid parameters	Return true, registration can continue and verify driverLicense and code Account

UserManager ⟶ verify(driverLicense,codeAccount)	
One or more parameters are null	A NullArgumentException is raised
DriverLicense already used	Return false, user already registered
Incorrect driverLicense or code account	An InvalidArgumentException is raised
Set of valid parameters	Return true, registration is finished and user can be added to the system

UserManager ⟶ generatePassword()	
No input parameter	Create and return a password for new user

UserManager ⟶ addUser(name, surname, mail, cf, driverLicense, codeAccount)	
One or more parameters are null	A NullArgumentException is raised
One or more parameters are incorrect (verifyAlreadyRegistered or verify returns false)	An InvalidRegistrationException is raised, user is already registered
One or more parameters are incorrect (verify or verifyAlreadyRegistered raise an exception)	Raised exception is propagated, user inserts incorrect parameter
Set of valid parameters (verify and verifyAlreadyRegistered returns true)	Create a new user with parameters and password obtained from generatePassword and then user is added to the user's list and is written in the database

4 Tools and test equipment required

4.0.1 Tools

Tests are a significant key to produce stable and reliable systems, but to be effective they have to be carried out thoroughly. The best way to reduce the efforts, focusing on the goals, is to rely on tools optimized for that purpose.

Our advice is to adopt the *JUnit* framework flanked by the *Arquillian* integration testing framework, because of the Java environment that characterizes the system components.

Arquillian is an innovative and highly extensible testing platform for the JVM that enables developers to easily create automated integration, functional and acceptance tests for Java middleware. This tool will help to verify that components interact each other in the designed way, producing the expected behaviors. In particular, Because of our experience and based also on the guidelines of the tool, to add the framework to the system is better to work with *Apache Maven*.

JUnit is a proven framework which allows to deal with test cases gracefully, directly in the developing environment, to speed up and simplify developers' and testers' life. Then it is clear that this tool will be used to support tests by Arquillian.

To perform the unit tests, we suggest to adopt the *Mockito* framework to take advantage of its flexibility and its power in the simulation of objects and behaviors of classes and methods.

5 Program stubs and test data required

5.1 Program stubs and drivers

The integration strategy we chose does not require as much stubs or drivers as the bottom-up or top-down strategies. A stub needed is the one which simulates the external Payment Handler behavior. We also need drivers to test the *DataManager*, which simulates the behavior of the components in the writing and reading operations.

5.2 Test Data

Some test data are required to tackle the planned tests, and of course this data have to especially bring to light the plight of the test cases.

When stated "fair" is intended that there are at least one data which satisfies the condition required and at least a bunch of data that are considered safe by the test designers (that is, supposed to be regular, standard, correct). Here below a list to be complied:

- A fair amount of data related to the Map thread, including instances presenting:
  - null object;
  - null fields;
  - invalid positions;
  - invalid cars;
  - available cars;
  - unavailable cars;
  - PowerGrid stations;
  - SafeAreas;
- A fair amount of data related to the Issue thread, including instances presenting:
  - null object;
  - null fields;
  - invalid man;
  - invalid car states;
  - invalid issue states;
  - invalid phone numbers;

  - invalid description;

- A fair amount of data related to the Rental thread, including instances presenting:
  - null object;
  - null fields;
  - absurd plate number;
  - unavailable cars;
  - user involved in a rental yet;

- A fair amount of data related to the Unlock & start rental, including instances presenting:
  - null object;
  - null fields;
  - illegal code;
  - terminated rental;
  - available car;

- A fair amount of data related to the Keep aside, including instances presenting:
  - null object;
  - null fields;
  - terminated rental;
  - available car;

- A fair amount of data related to the End rental, including instances presenting:
  - null object;
  - null fields;
  - terminated rental;
  - a non-terminated rental;

- A fair amount of data related to the Reservation, including instances presenting:
  - null object;
  - null fields;
  - unavailable car;
  - busy user;
  - invalid reservation;

- A fair amount of data related to the End reservation, including instances presenting:

- null object;
- null fields;
- available car;
- invalid reservation;

6 Effort spent

Marco [h]	Daniele [h]
26	22

7 References

To draw up this document, we refer to the sample Intergation Test Plan Document provided in the lectures.