Integration Test Plan Document

Document Version 1.0

Matteo Frosi (mat. 875393)
Luca Costa (mat. 808109)

Contents

1 Introduction

1.1 Scope and purpose

The purpose of the Integration Test Plan Document (ITPD) is to present to the testing team the sequence of tests to be applied to different components and interfaces forming the application. The goal is to test whether the components and interfaces designed and proposed in the Design Document behave in the expected manner and interact with each other correctly or not.
Each component (interface) is overlooked by a test, that analyzes its correctness and coherence.

1.2 Definitions and abbreviations

- RASD: Requirements Analysis and Specifications Document.
- DD: Design Document.
- ITPD: Integration Test Plan Document.
- Stub: some codes emulating other functionality or data, eventually using fake data.
- Drivers: drivers are like stubs with the difference that they are not used to be called by the component actually tested, but they are used to call themselves specific functions of the component actually tested. In this document the word driver is also used to design the "driver component" of our application. The distinction between the two should easily be done by the reader thanks to the context.
- Mocks: stubs with the possibility of verifying whether or not a specific method of this mock has called a specific number of times. Mocks are therefore slightly more complex stubs.
- Unit test: the most famous way to perform tests via assertions.
- Bottom-up: Bottom-up is a strategy of information processing. It is used in many different fields such as software or scientific theories. Regarding integration testing, the bottom-up strategy consists in the integration of low level modules first and the integration of higher level modules after.
- Top-down: Top-down is a strategy of information processing. Regarding integration testing, the top-down strategy consists in the integration of high level modules first and the integration of lower level modules after. It is the opposite of bottom-up.
- Big-bang: Big-bang is a non-incremental integration strategy where all the components are integrated at once, right after they are all unit-tested.
- jMeter: Java GUI program to measure the performance of a web server, it is developed by Apache.
- Apache: open source software company.
- SMS: short message service; it is the most famous way to send text messages to a mobile phone.
- Push notification: the modern way to send complex messages to a smartphone.
- API: application programming interface; it is a common way to communicate with another system.

2 Integration strategy

2.1 Entry criteria

In this subsection there are specified the criteria that must be met before integration testing of specific elements can begin (e.g., functions and modules must have been unit tested).
Some classes, the ones representing the software model, must be unit-tested immediately, to perform then the rest of tests, such the integration ones. These classes includes:

- Reservation
- Monitoring
- Ride
- SafeArea
- Car
- User

Moreover, all the classes related to the controllers should be unit-tested, independently from the other model classes. As example, the ServiceCore component includes all the algorithm classes, two of them discussed in section 3 of the Design Document.
All the methods, with the exception of getters, should be tested. Referring once again to the algorithms presented in the DD as example, we have (with the notation # representing inside the class and -> inside the component):

- Car#changePosition: elaborates the new position of the car
- ServiceCore->LocationModule#isInSafeArea: checks whether a car is in a safe area or not
- ServiceCore->LocationModule#isMoreThan3kmDistant: checks whether a car is more than 3km distant from the nearest safe area
- ServiceCore->RideCostAlgorithm#calculateRideCost: invoked at the end of the service, it calculates the cost of a ride, given its parameters (duration, bonuses, and so on)
- ModelSkeleton#getMonitoringByCar: returns a Monitoring if, given the research car, there is one, nothing otherwise
- SafeAreas#addSafeArea: adds a safe area to the list of parking zones of PowerEnJoy

Obviously, the just presented methods are nothing but a fragment of all the parts of our software to be tested.

2.2 Elements to be integrated

All the components described in the component view section of the Design Document, 2.3.2, have to be integrated.

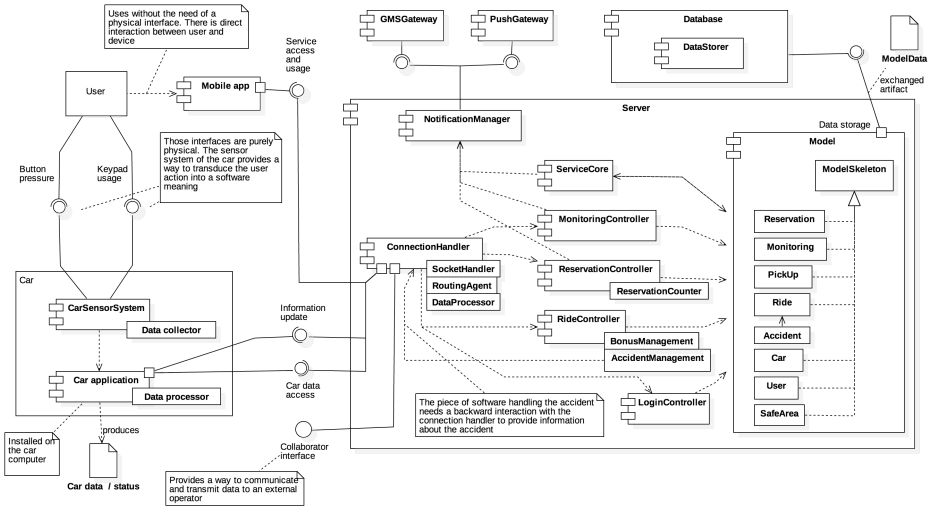

Figure 1: Components to be integrated

An interesting fact to point out is that mid level components, such as the controllers, don't need the lower components, or classed, to be fully implemented when tested. Each class forming a component needs in fact only some aspects and functionality provided by the lower level modules. As example, the RideCostAlgorithm in the ServiceCore component doesn't need a Model that includes methods to research active Reservation and Monitoring objects, but it does need information about the Ride on which it is needed to calculate the cost.

2.3 Integration Testing strategy

As previously stated, more complex components, such as the controllers, do not require a complete version of the depending modules/classes to be tested. This lead to a possible approach, very similar to the bottom-up method but with a significant difference. While the latter starts from the "leaves" of the software, that in this case are the model classes, the possible approach allows to start the testing from any depth of complexity, creating fragment of modules and only then integrating them, completing the lower level module. There are advantages but also drawbacks. Such approach is very fast in the early phases, because it allows to "take care" of the trickier components of the software and simultaneously makes the lower components be partially built. However, the tests are independent one from the other and the lower modules that are constructed to accomplish higher component tests may be incompatible or, in the worst case, conflicting. A good idea to solve this is to group each module realization given by the lower dependencies, but it requires foreseeing skills and it's difficult to accomplish. Moreover such approach is an easy trap, because if the starting depth is the highest, we will fall in the top-down approach.
A bottom-up approach suits better the sequence of integrations that will have to be applied on the components of the software, for the easy location of errors and the not need of stubs. Not always lower level means simplicity, and because higher level components, following the bottom-up approach, would be tested lastly, a variant of the bottom-up method could be followed. Instead of growing in depth, from lower to higher, it could be followed the dimension growth of modules, referring to the first approach presented. Moving down to top, so still using a bottom-up approach, some modules can be partially tested or even skipped, to leave space for more important modules. The skipped/partially tested classes/modules can be completed after that. So, we are not moving just upwards, but in a "snakelike" way. Such approach also gives the chance to catch a glimpse of the higher level components (and tests) while testing the lower

level ones.

2.4 Sequence of integrations

Because we are following a bottom-up like approach (with elements of top-down), both drivers and stubs are requested to be created, respectively that call and are called by the functions to be tested.

2.4.1 Server

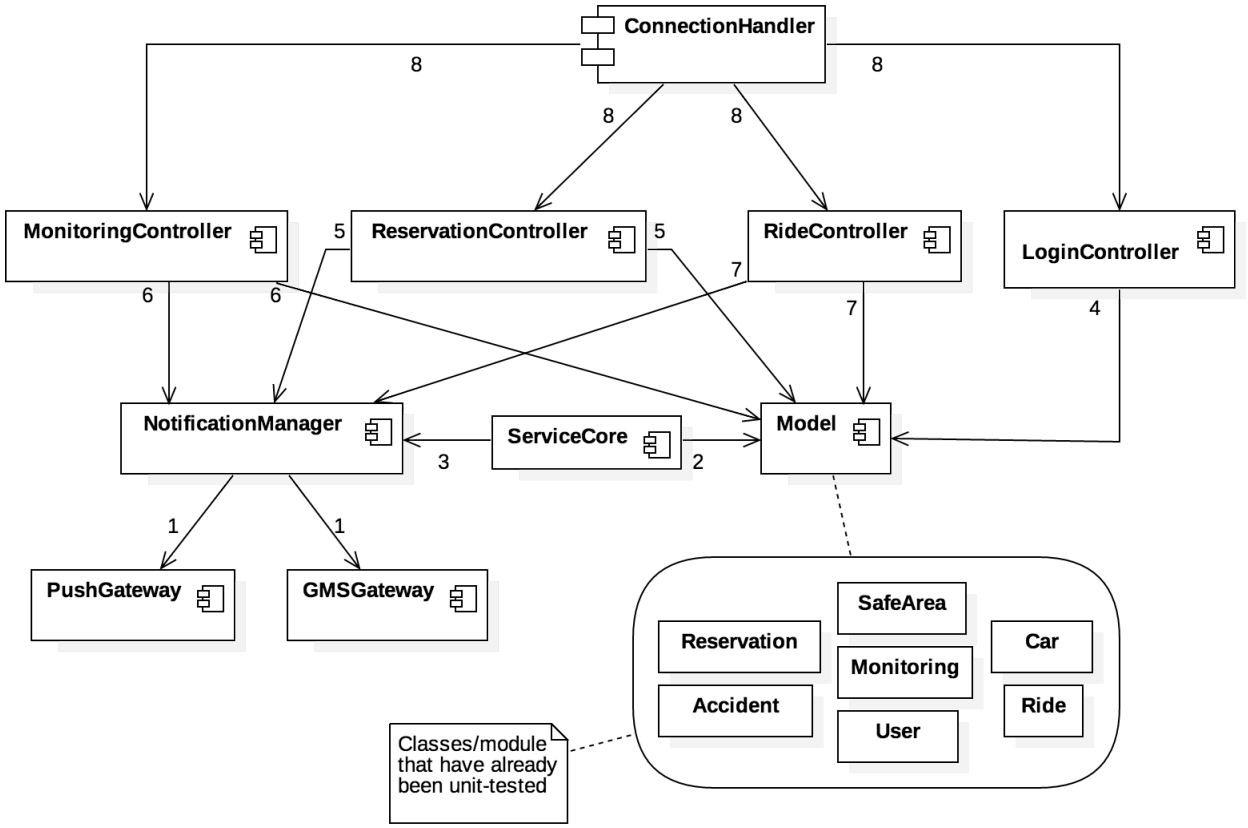

Figure 2: Integration sequence on the server

ID	Integration Test	Paragraph
IS1	NotificationManager -> GMSGateway, PushGateway	3.1.1
IS2	ServiceCore -> Model	3.1.2
IS3	ServiceCore -> NotificationManager	3.1.3
IS4	LoginController -> Model	3.1.4
IS5	ReservationController -> Model, NotificationManger	3.1.5
IS6	MonitoringController -> Model, NotificationManager	3.1.6
IS7	RideController -> Model, NotificationManager	3.1.7
IS8	ConnectionHandler -> ReservationController, MonitoringController, RideController, LoginController	3.1.8

A note must be made on the integration tests with ID 4, 5, 6 and 7, referring to the controllers of the server. Even though they requires an integration with the Model component, they interact just with a small chunk of its modules. As example, the ReservationController will interact only with Car, SafeArea, Reservation and User, while Ride will interact with all except for the Monitoring and Reservation objects. Such consideration makes the integration test easier and less verbose, due to the fact that each integration requires not all the modules of a component.
Last important thing to mention is that there could also be an integration test between the ConnectionHandler and the Model, because of the interaction between the Car and the Server, but the exchanged messages are very simple, just a cluster of data, and such handled directly by the ConnectionHandler directly into the Model.

2.4.2 Car

The car needs only one test for integration, assuming that both components, the software part of the CarSensorSystem and the CarApplication have already been unit tested. Without any need of figures describing the situation, we have:

ID	Integration Test	Paragraph
IC1	CarApplication -> CarSensorSystem (software)	3.2.1

2.4.3 Subsystem integration sequence

The PowerEnJoy software can be divided into different high level subsystems, As can be seen in section 2.3.1 of the Design Document, the high level components are four:

- Server
- Database
- User
- Car

As it can be seen from Figure 2, in subsection 2.4.1, describing the server sequence of integrations to be done, the Server is composed by three big chunks of components:

- Communication components: NotificationManager, PushGateway, GMSGateway, ConnectionHandler
- Processing components: ServiceCore, ReservationController, MonitoringController, RideController
- Data storing components: Model and its subclasses

These three clusters can be considered as different subsystems. The Model needs no further test since all of its modules have already been unit-tested. There is not a complete integration step between two or more components, but there are multiple interactions. As example we can just consider the Server integration sequence described in 2.4.1, having the Communication, Processing and Data storing components interact multiple times in the integration.
Once the integration between the Server components is done, the Database will be integrated, followed by the Car and lastly the User. The User must be integrated lastly because this way it can fully make use of all the functionality provided by the other subsystems. However the integration of the Car subsystem requires a version of the Client, that acts as a driver, to work and to be tested.

Why the Car subsystem first and then the User and not viceversa? The Car subsystem moves in a stand-alone way (the periodic updates do not requires the intervention of the user, hence the User components) or user driven. The last includes a single event, from the user, that leads, in a waterfall way, to the Car sub-system usage. It is then very easy to "stimulate", and then test, the Car before the User just by emulating such events (as a simple Reservation or PickUp).
In the following figure, representing what has been just said, the arrows direction mean "What has to be integrated with what".

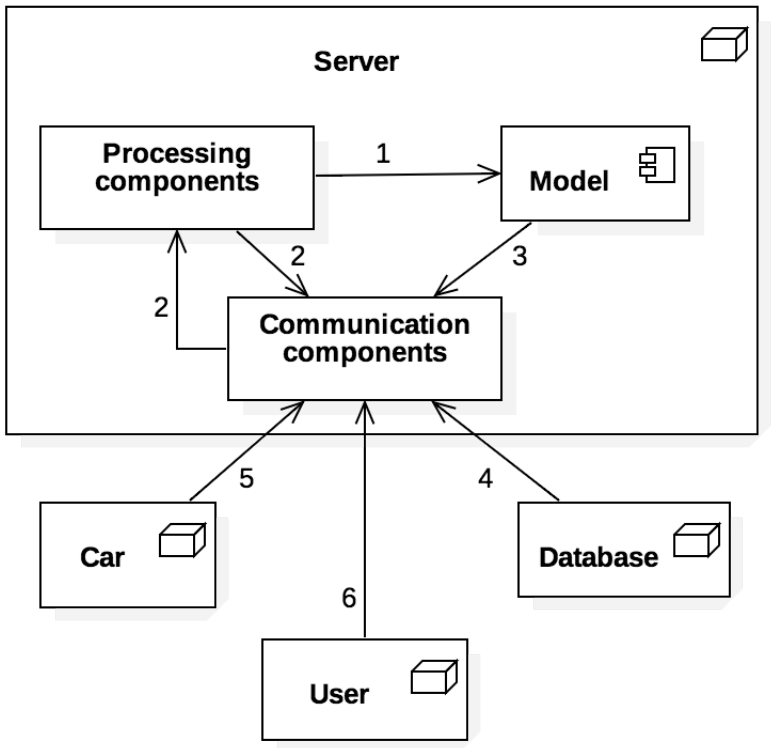

Figure 3: Subsystem integration

ID	Integration Test
SS1	Processing components -> Model
SS2	Processing components <-> Communication components
SS3	Model -> Communication components
SS123	*At this point the Server is fully integrated internally*
SS4	Database -> Server
SS5	Car -> Server
SS6	User -> Server

3 Individual steps and Test description

3.1 Server integration test cases

3.1.1 Test case IS1

- **ID:** IS1T1.
- **Test Item(s):** NotificationManager -> GMSGateway.
- **Input specification:** User (from the Model).
- **Output specification:** a notification is sent as a mail or an SMS to the GMS gateway.
- **Description/purpose:** when the user creates an account or requests the access code for the car he/she has to be correctly notified via SMS or mail. This is possible verifying the interaction between the NotificationManager and GMSGateway components.
- **Dependencies:** GMSGateway stub.

- **ID:** IS1T2.
- **Test Item(s):** NotificationManager -> PushGateway.
- **Input specification:** User (from the Model).
- **Output specification:** a push notification to the GMS gateway.
- **Description/purpose:** when the user creates succeeds in creating a reservation or a monitoring or when it receives the ride end data he/she has to be correctly notified. This is possible verifying the interaction between the NotificationManager and PushGateway components.
- **Dependencies:** PushGateway stub.

3.1.2 Test case IS2

- **ID:** IS2T1.
- **Test Item(s):** ServiceCore -> Model.
- **Input specification:** ServiceCore modules (algorithm classes), Monitoring, Reservation, Car, User, Ride, SafeArea.
- **Output specification:** all the algorithms are executed in the correct way.
- **Description/purpose:** while the Model contains only a skeleton, a copy of the software data, the ServiceCore components contain all the necessary modules to compute and process the data. All the connections between the ServiceCore and the Model have to be tested.
- **Dependencies:** NONE.

3.1.3 Test case IS3

- **ID:** IS3T1.
- **Test Item(s):** ServiceCore -> NotificationManager
- **Input specification:** Model modules.
- **Output specification:** a notification (push or message) is sent to the User.
- **Description/purpose:** whenever an event occurs, such as a successful end of the service or just a registration, the User has to be notified. It may seem strange that this happens at the ServiceCore level and not for each respective controller, but the controllers only has the task to elaborate the incoming information and instruct the Model (and ServiceCore) on what to do. The real logic can be found in the ServiceCore, hence the end of an event chain triggers the notification for the User.
- **Dependencies:** GMSGateway and PushGateway stub.

3.1.4 Test case IS4

- **ID:** IS4T1.
- **Test Item(s):** LoginController -> Model.
- **Input specification:** User (from Model).
- **Output specification:** an existing User in the Model changes status in successfully logged in or logged out as a consequence of a trigger message or an expiration of a timer.
- **Description/purpose:** the correct interaction between the LoginController and the model has to be verified, since it represents one of the most important phase. An incorrect interaction may lead to wrong usage of the service and it can even be exploited to damage it. Hence, this is a critical test, that must be done before the other controllers are integrated (as example a not logged user mustn't make a reservation or a monitoring request.
- **Dependencies:** Database stub.

3.1.5 Test case IS5

- **ID:** IS5T1.
- **Test Item(s):** ReservationController -> Model.
- **Input specification:** ReservationController modules, Reservation, Car, User (from Model).
- **Output specification:** a request of a reservation, from the user, is handled correctly.
- **Description/purpose:** the user must be able to make a reservation and to take it back. Each action includes the elaboration of the request (handled by the ReservationController modules) and the creation/usage of the appropriate objects corresponding to the Model. This integration test verifies the correctness of each sequence of operations when all the events concerning a reservation occurs.
- **Dependencies:** NONE.

- **ID:** IS5T2.
- **Test Item(s):** ReservationController -> NotificationManager.
- **Input specification:** ReservationController modules, Reservation, User (from Model).
- **Output specification:** a push notification is sent to the User, about the handled Reservation.
- **Description/purpose:** when the sequence of operations about a Reservation (make, take back, and so on) ends, the user has to be notified and so the ReservationController and NotificationManager must communicate and interact correctly.
- **Dependencies:** PushGateway stub.

3.1.6 Test case IS6

- **ID:** IS6T1.
- **Test Item(s):** MonitoringController -> Model.
- **Input specification:** MonitoringController modules, Monitoring, Car, User (from Model).
- **Output specification:** a request of a monitoring, from the user, is handled correctly.
- **Description/purpose:** the user must be able to make a monitoring request (or more) and to take it back. Each action includes the elaboration of the request (handled by the MonitoringController modules) and the creation/usage of the appropriate objects corresponding to the Model. This integration test verifies the correctness of each sequence of operations when all the events concerning a reservation occurs.
- **Dependencies:** NONE.

- **ID:** IS6T2.
- **Test Item(s):** MonitoringController -> NotificationManager.
- **Input specification:** MonitoringController modules, Monitoring, User (from Model).
- **Output specification:** a push notification is sent to the User, about the handled Monitoring.
- **Description/purpose:** when the sequence of operations about a Monitoring (make, take back, check and so on) ends, the user has to be notified and so the MonitoringController and NotificationManager must communicate and interact correctly.
- **Dependencies:** PushGateway stub.

3.1.7 Test case IS7

- **ID:** IS7T1.
- **Test Item(s):** RideController -> Model.
- **Input specification:** RideController modules, Ride, User (from Model).
- **Output specification:** a ride is handled correctly.
- **Description/purpose:** when the user accesses the car and turns on the engine, the ride starts. The server is constantly informed about the ride data, such as location, cost, duration, fee, and so on. The ReservationController has to elaborate them and integrate all of this information with the Model objects.
- **Dependencies:** NONE.

- **ID:** IS7T2.
- **Test Item(s):** RideController -> NotificationManager.
- **Input specification:** RideController modules, Ride, User (from Model).
- **Output specification:** a push notification is sent to the User, about the handled Reservation.
- **Description/purpose:** when the sequence of operations about a Ride (begin, check state, end) ends, the user has to be notified and so the RideController and NotificationManager must communicate and interact correctly.
- **Dependencies:** PushGateway stub.

3.1.8 Test case IS8

- **ID:** IS8T1.
- **Test Item(s):** ConnectionHandler -> ReservationController.
- **Input specification:** Method triggers (reservation concerning messages).
- **Output specification:** the incoming message is correctly sorted into the ReservationController.
- **Description/purpose:** Among all the incoming messages (both from the car and from the user) concerning a reservation have to be handled by the respective controller, that is the ReservationController.
- **Dependencies:** NONE.

- **ID:** IS8T2.
- **Test Item(s):** ConnectionHandler -> MonitoringController.
- **Input specification:** Method triggers (monitoring concerning messages).

- **Output specification:** the incoming message is correctly sorted into the MonitoringController.
- **Description/purpose:** Among all the incoming messages (both from the car and from the user) concerning a monitoring have to be handled by the respective controller, that is the Monitoring.
- **Dependencies:** NONE.

- **ID:** IS8T3.
- **Test Item(s):** ConnectionHandler -> RideController.
- **Input specification:** Method triggers (ride concerning messages).
- **Output specification:** the incoming message is correctly sorted into the RideController.
- **Description/purpose:** Among all the incoming messages (both from the car and from the user) concerning a ride have to be handled by the respective controller, that is the RideController.
- **Dependencies:** NONE.

- **ID:** IS8T4.
- **Test Item(s):** ConnectionHandler -> LoginController.
- **Input specification:** Method triggers (login and registration concerning messages).
- **Output specification:** the incoming message is correctly sorted into the LoginController.
- **Description/purpose:** Among all the incoming messages (both from the car and from the user) concerning a login or a registration have to be handled by the respective controller, that is the LoginController.
- **Dependencies:** NONE.

3.2 Car integration test cases

3.2.1 Test case IC1

- **ID:** IC1T1.
- **Test Item(s):** CarApplication -> CarSensorSystem (software)
- **Input specification:** CarSensorSystem data.
- **Output specification:** all the car data and signals are correctly elaborated.
- **Description/purpose:** before the integration with the rest of the subsystems, it is necessary to ensure that the car application works correctly, grasping the car electronics data and elaborating it.
- **Dependencies:** NONE.

4 Tools and Test Equipment Required

As described in the Design Document, the PowerEnJoy application is purely Java-based, so it is appropriate to use the numerous tools offered for testing java enterprise applications, including jUnit (for unit-testing) or Arquillan (for integration and functional testing).
When testing, especially in the early phases (and so mostly during the unit-testing), dummy objects will be created to represent real situations. However, as stated in section 2.3, those fakes should be created with a certain "logic" so that each fragment can be combined to form the real abstraction. As example, consider the class Reservation: in different unit tests it will be needed, but only partially, providing just some methods; combining all of its "versions" we may obtain a good result of the class, even almost complete.

4.1 Automatic tests

Different tools can be used to achieve automatic, for both integration and unit, tests.

- **EvoSuite**: tool that automatically generates test cases with assertions for classes written in Java code. EvoSuite applies a novel hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion. For the produced test suites, EvoSuite suggests possible oracles by adding small and effective sets of assertions that concisely summarize the current behavior.
- **JUnitEE**: is an extension of JUnit, which runs directly in the same application server as the project. JUnitEE provides three Servlets, which call the regular JUnit test cases.
- **Mock objects**: are mostly used for Unit Testing. They help in testing the interactions between the objects in an application. The one that will be used are JMock and Mockito.
- **Arquillian**: very powerful tool for Integration and Functional testing of Java middleware. It is most frequently used with the build tool Maven and on top of Unit Testing framework such as JUnit and TestNG. It frees the tester from creating Mock objects. Arquillian can be used to test JSF, EJB, Servlets and other Java classes.
- **SOASTA TouchTest™**: it delivers complete functional test automation for continuous multi-touch, gesture-based mobile applications. Mobile devices are controlled through a lightweight software agent, SOASTA TouchTest Agent.

Following two JUnit4 tests are presented, related to the possible implementation of algorithms presented in the Design Document, section 3.

```
private MonitoringAlgorithm algorithm;
private Model mo0;
private User u0, u1;
private Car c0,c1;

@Before
public void setup() {
    mo0 = new Model();
    algorithm = new MonitoringAlgorithm(mo0);

    u0 = new User("Lilith","gravure", "1152");
    u1 = new User("Mitch", "wife", "2333");
    c0 = new Car("LD111", new Position());
    c1 = new Car("LS122", new Position());
}

@Test public void correctRequest() throws ReservationAlreadyDoneException, AlreadyRequestedException {
    algorithm.monitoringRequest(u0,c0);
    assertTrue(mo0.getMonitoringByCar(c0).isMonitoring(u0));
    algorithm.monitoringRequest(u1,c0);
    assertTrue(mo0.getMonitoringByCar(c0).isMonitoring(u1));
    algorithm.monitoringRequest(u1,c1);
    assertTrue(mo0.getMonitoringByCar(c1).isMonitoring(u1));
}


@Test (expected = ReservationAlreadyDoneException.class)
public void userWithReservation() throws ReservationAlreadyDoneException, AlreadyRequestedException {
    mo0.addReservation(new Reservation(c0,u0));
    algorithm.monitoringRequest(u1,c1);
    algorithm.monitoringRequest(u0,c1);
}

@Test (expected = AlreadyRequestedException.class)
public void alreadyMonitoring() throws AlreadyRequestedException, ReservationAlreadyDoneException {
    algorithm.monitoringRequest(u0,c0);
    algorithm.monitoringRequest(u0,c0);
}
```

Figure 4: Monitoring algorithm class test

```
private RideCostAlgorithm algorithm;
private Model m0;
private ServiceCore sc0;
private Ride r0;
private Car c0;
private SafeAreas s0;
private FixedPosition p0,p1,p2;
private int normalTimeStep, stopByStep, passengerStep;
private int carX, carY, carZ, carBattery;
private boolean carPlugged;
private float expectedCost;

@Parameterized.Parameters
public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] {
            {20, 3, 0, 1, 2, 3, 40, false, (float) 55.0},       //No bonuses or overcharge
            {25, 0, 13, 1, 2, 3, 33, false, (float) 67.275},    //Passenger bonus
            {10, 5, 2, 2, 2, 2, 55, false, (float) 17.52},      //High battery bonus
            {10, 5, 4, 2, 2, 2, 15, false, (float) 28.47},      //Low battery overcharge
            {10, 5, 0, 10, 10, 10, 40, false, (float) 28.47},   //Distance overcharge
            {20, 0, 9, 1, 2, 1, 35, true, (float) 41.86},       //Plugged car bonus
            {40, 0, 35, 1, 1, 1, 80, true, (float) 47.84},      //Virtuous user, all bonuses
            {40, 0, 0, 10, 10, 10, 15, false, (float) 191.36},  //Not virtuous user, all overcharges
            {40, 0, 22, 9, 4, 7, 15, true, (float) 107.64}      //Mixed ride
    });
}

public RideCostTest(int normalTimeStep, int stopByStep, int passengerStep,
                    int carX, int carY, int carZ, int carBattery, boolean carPlugged,
                    float expectedCost) {...}

@Before
public void setup() {...}

@Test public void calculateRideCost() {
    r0.increaseDurationOfRide(normalTimeStep);
    r0.increaseDurationOfStopBy(stopByStep);
    r0.increaseDurationOfPassengers(passengerStep);
    c0.changeCurrentCharge(carBattery);
    c0.changePosition(new Position(carX, carY, carZ));
    if(carPlugged) c0.plug();
    else c0.unplug();

    assertEquals(expectedCost, algorithm.calculateRideCost(r0), 0.001);
}
```

Figure 5: Ride cost algorithm class test

4.2 Manual tests

The whole application will be tested manually, to find possible way of improvement or faults. One way to accomplish that is relying on the user experiences, which serves both to find design faults or dissatisfaction with the requirements of the software (as shown in the sample of figure 22, subsection 4.2 of the Design Document). Moreover technical aspects have to be tested, both in the user's device and in the cars (such as the piece of software that retrieves the data of the car or the position of a device).

4.3 Performance tests

To have a performance test we will use Apache JMeter. This tool may be used to test performance both on static and dynamic resources, to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyze overall performance under different load types.
We will perform a huge amount of requests of the same type and uniformly distributed (heavy load on reservation requests but also high amount of data exchange from the devices).

Moreover, some objects will be forced to endure extreme work, such as the logic components (great number of simultaneous algorithms invoked and such).
Important factors that will be analyzed will be response time for each request, maximum load and response behavior to peaks of work surrounded by periods of stasis (this represents a more real-world representation, because the service will be likely to be used more during certain periods of the day, like during the midday or evening work pause).

5 Program stubs and Test data required

5.1 Stubs

Since a bottom-up like approach has been followed, there are just a few stubs required when having the integration tests.

5.1.1 GMSGateway

Usages

- IS1T1
- IS3T1

Description The GMSGateway stub allows to test the SMS and mail sending functionality, emulating the external gateway. One of the main advantages is that there is no real usage of the external service (that is usually offered by an external provider, hence it has a money cost) and there is no need to test network features (it just needs to emulate the sending of stuff, not to do it for real).

5.1.2 PushGateway

Usages

- IS1T2
- IS3T1
- IS5T2
- IS6T2
- IS7T2

Description The PushGateway stub allows to test the push notification functionality, emulating the external gateway. As for the GMSGateway stub, it provides a way to use the push service without paying and no network features are needed.

5.1.3 Database

Usages

- IS4T1

Description Even before the subsystem integration and tests, during the Server components tests there is a need to simulate a real container for data, concerning the registration of a user. However integrating immediately the DB would be a waste of time and a complex procedure because it is not necessary to use all of its functions. Hence, a fictional version will be created to just receive registration requests and store small amount of data, mostly user concerning.

5.2 Data needed for the tests

As it can be seen in the samples provided in subsection 4.1, a large number of fake objects (and consequently fake data) has to be inserted in the database and recreated in the model, to accomplish effective tests.
Various java libraries fills objects with dummy data (such as DummyCreator) and Arquillian provides a way to easily populates the database with such objects, used for testing.

5.3 Critical data tests

As it was described in the section dedicated to the performance tests, some situation may require particular fake data clusters, such as:

- High number of cars sharing one or more features (position almost equal, same charge, same conditions).
- High number of users making the same reservation or monitoring request.
- Missing data when doing an operation (what if somehow, a user can make a reservation over an already occupied car?).
- Incomplete data.
- No categories (no users, no reservations, no monitoring, no rides, no accidents and so on).

6 Document further information

6.1 References

- RASD_4.7, before version 4.7
- DD_3.2, before version 3.2
- Assignments AA 2016-2017.pdf
- testing.pdf
- TP1.1.pdf
- Integration Test Plan Example.pdf
- `http://www.softwaretestinghelp.com/java-testing-tools/`, describing some of the most well known automated testing tools for java based applications
- `http://jmeter.apache.org`, describing the tool for performance testing
- `http://arquillian.org`, describing the most used tool for Integration and Functional testing of Java middleware

6.2 Used tools

In creating the ITPD document, the following tools have been used:

- Github, for version controller
- LyX, to write the document and converting in .pdf format
- Hunspell, for the spell check of the document
- Atom, for the project markdown

6.3 ChangeLog

- [21/12/2016] [**Version 0.1**] :: Given a structure to the document.
- [24-28/12/2016] [**Version 1.0**] :: Added all the sections of the document.

6.4 Hours of work

Matteo Frosi

[21-28/12/2016]: 14.00 hours (spent writing the whole document)
TOTAL: 14.00 hours

Luca Costa

TOTAL: 0.00 hours