Combining Mockito with JUnit for Software Quality Control

João Brito, M9984
Department of Computer Science
University of Beira Interior
Covilhã, Portugal
joao.pedro.brito@ubi.pt

Tomás Jerónimo, M9988
Department of Computer Science
University of Beira Interior
Covilhã, Portugal
tomas.jeronimo@ubi.pt

***Abstract*—The present paper has the primary purpose of highlighting the characteristics of both JUnit and Mockito, their natural mutualism and how to implement them in a real world application. Moreover, theoretical aspects will be described in, hopefully, simple terms, to serve as the foundation for some practical realisations.**

I. Introduction

Within the scope of this work, both Mockito and JUnit were studied. As it will become increasingly clear, Mockito shall be used as an enabler for Unit Testing, rather than performing the tests itself. That task will be reserved for JUnit.

With this thought in mind, the next sections will present introductions to both JUnit and Mockito, how they interact with each other and a case study, which aims at translating theory into practice.

II. Background and Foundational Concepts

In this section, we provide some insight on the concept of Unit Testing and some relevant definitions, useful for the full comprehension of the present document.

A. What is Unit Testing?

Being a key concept in the present document, Unit Testing is a form of software testing where individual components are tested. A system/piece of software can be thought of as a combination of small, indivisible pieces (units). Therefore, each and every unit has to perform according to the accepted specifications. Frameworks and techniques that apply such principals were, ultimately, what the authors of this document sought after.

In the context of Object Oriented Programming (OOP), a method is the unitary block, the smallest fundamental piece. Given this, testing will target such entities (although they appear inside classes, these shouldn't be considered the units).

The remainder of this paper will be dedicated to exploring tools and methodologies that incorporate the important concept of Unit Testing.

B. Android Studio IDE

Android Studio is the official Integrated Development Environment (IDE) for the conception, testing and development of Android applications. It is based on IntelliJ and offers the following features, among others:

- Flexible compilation system, based on Gradle.
- Fast emulator with inumerous resources.
- Unified environment that enables the deployment of applications on a wide range of devices.
- Linting tools that empower the programmer when it comes to debugging.

Naturally, this tool was heavily used, both to automate the tests and to refactor the original code for the case study we'll go over in section VII.

C. JSON

An acronym for *JavaScript Object Notation*, a JSON is a compact, open-source data exchange format. It uses a key-value approach where an attribute (called a "key") has one and only one value in a structure resembling a dictionary. Among the supported data types are Integers, Strings and Booleans.

In this project, a specific JSON will prove to be an important part of the chosen case study.

D. HTTP request

An HTTP client can send an HTTP request to a server the format Method SPACE Request-URI SPACE HTTP-Version CRLF. Breaking this format down: a Method can be of several types, among which, GET (to retireve data) or POST (to send data); a Request-URI identifies unequivocally the resource upon which to apply the request and the HTTP-Version is self-explanatory, stating the version of this protocol that is being used.

III. JUnit

A. What is JUnit?

JUnit is an open source testing framework, created by Erich Gamma and Kent Black, that allows the creation of automated tests in the Java programming language. It is part of a family of unit testing frameworks, collectively known as xUnit.

A JUnit test is a method contained in a class which is only used for testing. This method executes the code under test and checks if the expected result is the same as the actual result. A formal written unit test case is characterised by a known input and expected output. The known input should

test a precondition and the expected output should test a post-condition. These tests cases should also provide a meaningful message, allowing the tester to identify and fix any problems. Generally, there must be created, at least, two unit cases for each requirement, one being an expected positive test and the other an expected negative test.

This testing framework promotes the idea of setting up the test data for a piece of code first and implement it after. This approach increases the stability of the program code, which reduces the time spent debugging.

B. JUnit features

As mentioned in the previous section, JUnit allows us to mark methods, in Java, as tests. In this section it is going to be discussed how JUnit is able to find and execute these tests.

To find the methods marked as tests, JUnit provides a tool called *test runner* that searches for tests in the code file and executes those blocks of code. As each test is executed, it is necessary to validate them, which is why JUnit also provides an *assertion library*. This library is what allows us to compare the expected outputs with the actual ones and determine if the test fails of passes.

JUnit also provides some basic reporting tools to mark the result of each test. If an assertion fails, JUnit will report the test as failed. If the code causes an exception it will also get marked as failed. If everything runs smoothly and all assertions are valid, it will report a pass.

C. Code examples

In this section there are going to be presented a couple of code examples for writing JUnit tests where each one will represent a different annotation. It is important to note that the examples used are from JUnit 5.

The first annotation that is going to be presented is the *@Test* annotation. This simply marks a method as a unit test.

```
public class AppTest {
    @Test
    public void testOne() {
        System.out.println("Running a test");
    }
}
```

Two other important annotations are the *@BeforeEach* and *@AfterEach*. These annotations mark methods to be run before or after, respectively, each test method in the test class.

```
public class AppTest {
    @BeforeEach
    public void setup() {
        System.out.println("@BeforeEach
            executed");
    }
}
```

The final example is going to be an assertion example. Assertions are how the automation of the scenario that is being tested begins and how it is decided if what the application is returning is acceptable. There are many examples if assertions in the JUnit documentation[CITE] but only a few will be presented.

```
public class AppTest{
    @Test
    public void testOne() {
        int x = 2 * 2;
        Assert.assertEquals(4, x);
        Assert.assertThat(x, is(equalTo(4)));
    }

    @Test
    public void testTwo() {
        int x = 2;
        Assert.assertTrue(x==2);
    }
}
```

IV. Mockito

A. Mockito and the need for isolation

The second technique we will go over in this paper is, according to the official website, ”a mocking framework that tastes really good”. Translating this definition into more technical terms, Mockito is a mocking framework in the form of a JAVA based library.

This intuitive API is used to mock interfaces so that a dummy functionality can be executed and validated in a controlled environment.

One of the main driving forces behind Mockito is its focus on simplicity. Thus, the goal seems clear: simple test code that encourages the developer to write clean and simple application code.

Over the years, a relatively small team of dedicated of developers (among others, Szczepan Faber, Brice Dutheil, Rafael Winterhalter, Tim van der Lippe, Marcin Grzejszczak and Marcin Zajaczkowski) has updated the tool with various capabilities.

Now in its third major version, Mockito has several benefits. Just to name a few:

- Exception and return value support: Mockito is flexible enough to channel return values and/or deal with exceptions;
- Easiness: the programmer does not have to implement mock objects, it is taken care of by Mockito;
- Annotation support: pretty self explanatory, Mockito allows the use of Java annotaions.

Two key concepts appear regularly in this context: mock and stub. To clarify, a mock is a dummy instance of a possibly complex class with granular control over it. A stub focuses on rigging the dummy instance with pre-determined results.

Setting aside the theoretical substance, in practice, most of the classes we implement have dependencies (i.e. other ”foreign” methods are needed to perform the main task). Recalling the previous definition, it hints at a certain need for independence. But how exactly can we mock the behaviour of said dependencies?

B. Code examples

Simply put, by incorporating Mockito (or other similar tools) into the development process. Let's consider a simple, yet enlightening, example. Assume that we have implemented the following class in Java:

```
// CustomerService.java
public class CustomerService{

    @Inject
    private CustomerAux aux;

    public boolean addCustomer(Customer
        customer){

       if(aux.exists(customer.getID()))
           return false;

        return aux.save(customer);
    }

    public CustomerAux getCustomerAux(){
        return aux;
    }

    public void setCustomerAux(CustomerAux
        aux){
        this.aux = aux;
    }
}
```

In the context of Unit Testing and the Java programming language, a class is the basic unit, the building block of any application. So, it is also assumed that we have a dedicated test class for the previous class:

```
// CustomerServiceTest.java
public class CustomerServiceTest{

    @Mock
    private CustomerAux aux;

    @InjectMocks
    private CustomerService service;

    @Before
    public void setUp() throws Exception{

        MockitoAnnotations.initMocks(this);
    }

    // (...)
}
```

Given the code above, we would like to highlight some aspects:

- The annotation `@Mock` is used to create a mock implementation of the `CustomerAux` class.
- `@InjectMocks` will, as the name suggests, inject the mocks marked with `@Mock` into an instance of the class it is associated with (in this case, the `CustomerService` class).
- The `setUp()` method is responsible for instantiating the mocks we have declared.
- The `@Before` annotation will make sure that the `setUp()` method is called before every other method in this test class, therefore ensuring that our mocks are initialized and ready to be used.

The next step is to tell the mock what it should do when given a certain call:

```
// CustomerServiceTest.java
public class CustomerServiceTest{

    // (...)

    @Test
    public void testAddCustomer_True(){

        when(aux.save(any(Customer.class))).
        thenReturn(true);
    }

    @Test
    public void testAddCustomer_False(){

        when(aux.save(any(Customer.class))).
        thenReturn(false);
    }

    // (...)
}
```

Of note:

- The `@Test` annotation indicates that the following method is a test.
- To simulate (i.e. mock) the call of `ConsumerAux`'s `save()` method, we can use the `when(...).then(...)` pattern. Effectively, this operation is quite readable: when `save()` is called (with any instance of the `Customer` class), we will simulate it's response by returning the boolean value `true`. This process is often called "stubbing".

These code snippets are barely scraping the surface of what Mockito can achieve. In the following sections, besides entwining JUnit and Mockito, a case study will showcase their strengths, as well as, the Software Quality principles in play.

V. Environment Setup

This section aims at showing how Mockito (version 1) and JUnit 4 were setup in the test environment. Because the case study presented, in this document, is a mobile application this frameworks were installed in the Android Studio IDE.

To setup JUnit and Mockito the following dependencies were specified in the app's top level *build.gradle* file:

```
dependencies {
    testImplementation 'junit:junit:4.12'
    testImplementation
        'org.mockito:mockito-core:1.10.19'
}
```

After this, the test files can be created. Generally unit test are created in a separate folder. This way the test code is separated from the real code. The standart conversion for Gradle is to use *src/main/java* for Java classes and *src/test/java* for test classes.

VI. Theoretical Combination

So far this document has reflected on the definition and objective of each presented framework. In this section, a point will be made as to when should JUnit and Mockito be combined, highlighting the use cases of each one along the way.

As it was described in previous sections, a unit test should only test one functionality in isolation and side effects from other classes must be eliminated for the unit test. This can be archived by replacing the real dependencies for test doubles. Test doubles can classified according to their use:

- Dummy object - a object usually used to fill some parameters of a method. This object is never used and its methods are never called.
- Fake object - objects with working implementations that are usually simplified.
- Stub class - this are partial implementations for an interface or a class with the objective of using an instance of this stub class during a test.
- Mock object - these objects are fake implementations for an interface or a class in which the output of a certain method call is defined by the tester.

This is how the Mockito framework complements JUnit. It creates fake dependencies for external methods called inside the unit under test and allows the tester to define the output of those methods. This ensures that the class under test is not affected by any side effects and that the class is only tested when the testing is performed.

VII. Case Study

This section will present the application that is going to be studied and tested using both frameworks. First an introduction to the application will be made. After, each test is described and it's outcome is presented. Lastly, some main takeaways are explained in the result discussion.

A. Application Overview

The application chosen to be tested is called TitchersPET [5]. It is a mobile application and it was developed by this document's authors as a final project for the Programming of Mobile Devices subject.

The scope of the application is to provide a solution for teachers so that they can manage their classes in a more efficient manner. The application allows teachers to create personalised classes, create a profile for each student, make daily reports about the development and behaviour of each student, as well as register class attendance. It also provides a weather report, allowing the teacher to plan outdoor activities, and a functionality to email the daily reports to the student's parents.

The reason why this application was chosen was the fact that it was developed by the document's authors, ensuring they have a solid insight on the overall design. This is important because even though JUnit by itself does not belong, exclusively, in the White-Box testing methods, the addition of the Mockito framework makes it a White-Box testing technique.

The application starts with an login menu, as it can be seen in the figure 1, where the user, if already registered, can insert his information and start using the application. If this is not the case, the user must go through the registration process. The registration is an application form and an administrator must approve the request before the user is allowed to login (figure 1).

Fig. 1: Case study - Login Menu

After inserting his valid login data, the user is taken to the main menu where the following elements are available: a weather forecast for the following eight hours, four main buttons that display different menus, a settings button and an exit button. Each one of the menus available in the main activity contains different information about the user's students and various features that allow him to manage his class more efficiently. Each menu will be described bellow (figure 2).

The first menu (top left) presents a list of the user's students and by clicking in each one, information about them is displayed. The information can be edited and saved if needed. The user can also add a new student to his class by hitting the top right button and filling in the required information.

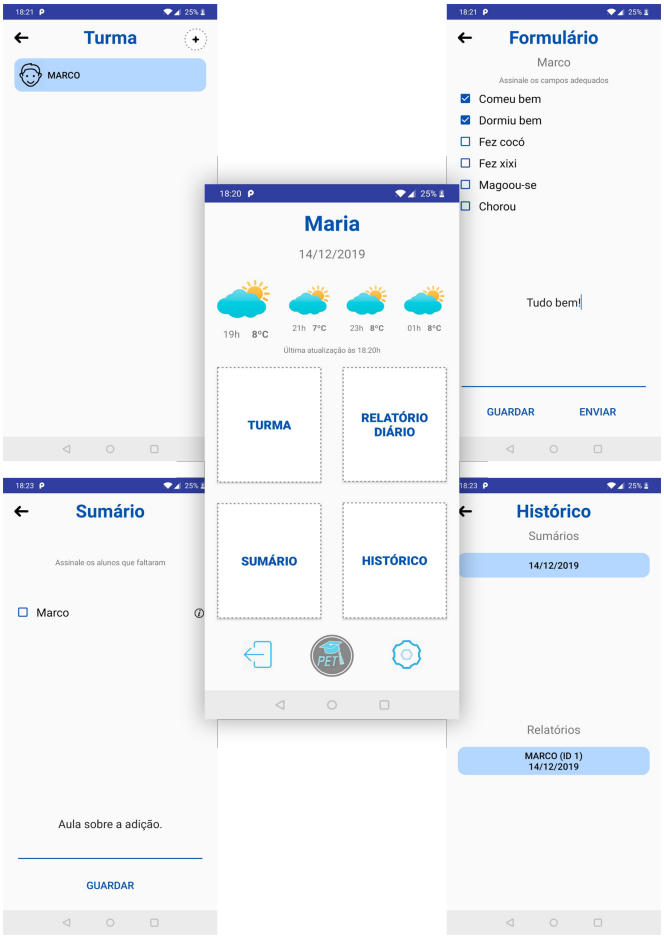

Fig. 2: Case study - Main screen and sub-menus

The second button (top right) allows the user to write a daily report about each student. It presents a list of students and by clicking each one the user can fill out a form about their daily activities. This report is then stored in the system and it can also be sent to the student's parents via email.

The third menu (bottom left) brings up a window where the user can fill in the attendance record and compose a brief summary for that day.

The last menu (bottom right) allows the user to visualise previous summaries and daily reports.

Given the presented layout, a list of tests was devised to test the following features: external dependencies and the various input fields. The test methodology (i.e. structure, objectives and results) is documented in the following sections.

B. Test External Dependencies (Weather Forecast)

In this test there are two main objectives, testing the external weather dependency, used to get the weather forecast, and testing how the application processes the received data. The weather forecast must present weather and outside temperature for the following twelve hours, with two hour intervals.

To get the weather information, an API was used. More specifically, the AccuWeather API, which retrieves a JSON package containing 12 weather predictions for the next 12 hours with each one including, but not limited to: the weather, the temperature, the likelihood of raining and the time of day.

The proposed tests aim to verify that, given a certain JSON package, the information retrieved from it's processing is the intended one. To do so, the weather class was mocked and instantiated using the created mock. The *when(...).thenReturn(...)* method chain was used to specify that the returned value of the HTTP request class must be the pre-defined JSON. The weather class is then called, with the pre-defined JSON being passed as input to the *parseJSON()* method, which will retrieve the following information: Hour, Temperature and IconPhrase (a small description):

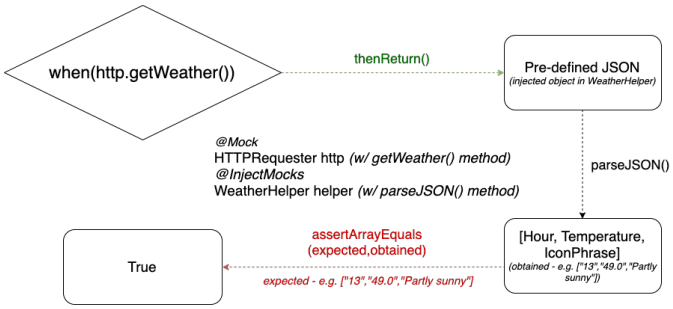

Fig. 3: Flowchart of weather forecast tests

The output values are compared with the expected ones, using the *assertArrayEquals()* method from JUnit. A representation of this process is detailed in figure 3. The results were then documented:

Injected object	Expected result	Assert Result
{..."13"..., ..."Partly sunny"..., ... "29.5"...}	["13", "Partly sunny", "29.5"]	True

TABLE I: Results from the Weather Forecast Testing

As it can be seen in Table I, the expected result matched the injected object and therefore the assert method returned a "True" output. From this result it can be concluded that the *JSONparse()* method in the WeatherHelper class was well developed and the values presented in the weather forecast are in accordance with the requirements.

C. Test User Inputs

This section covers a series of tests done to the different types of user inputs available in the case study. In each test a different input is used with the objective of documenting the application's response to it. The InputChecker class is responsible for that task and will be the class under test moving forward. There are four different inputs subjected to this tests and each one is represented by one of the following tables. In each table the first column represents the input that is going to be tested and the last two represent what the result should be and what it actually is. Hopefully, the tests provided are a solid representation of the real world variations, thus ensuring

resilient code. Figure 4 shows how the process of checking user input is handled:

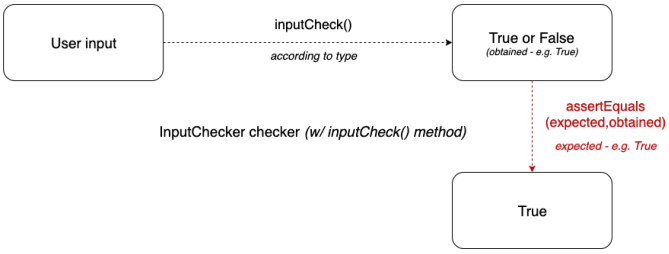

Fig. 4: Flowchart of user inputs tests

The first set of tests (Table II) covers the phone number input type. Here an accepted value would be a nine number input. Every other case should be processed as an invalid input:

Test input	Expected Result	Assert Result
912345678	True	True
(empty input)	False	False
91234567	False	False
abcdefghi	False	False
9876abcde	False	False
!#$%&/()=	False	False
98765!#$%	False	False

TABLE II: Results from the Phone Number Input Tests

As the table shows, different types of values were tested, for example, empty inputs, alphabetical values and special characters. With the exception of the first test (the valid nine number input) every other value was rejected by the application, as expected. It is then possible to conclude that the methods to verify phone number inputs were correctly implemented.

The following table (Table III) covers a more complex input type, email input. In this case, the rules for a valid input were the following: the input must contain one occurrence of the special character "@", the string before the "@" must not contain special characters, the domain must only have one the three prefixes, "google", "hotmail" or "ubi" and the suffix must either be ".pt" or ".com". This validation process is described in figure 4.

Test input	Expected Result	Assert Result
teste@gmail.com	True	True
teste@ubi.com	True	True
teste@hotmail.com	True	True
(empty input)	False	False
testehotmail.com	False	False
teste@gmailcom	False	False
teste@gmail.random	**False**	**True**
te%&@gmail.com	False	False
teste@hotm#?l.com	False	False

TABLE III: Results from the Email Input Tests

As expected, the first three inputs are accepted, as they are in accordance with the established rules. Every other input is rejected, as expected, with the exception of the value "teste@gmail.random". It is then possible to conclude that the validation process has a flaw in the suffix verification and after reviewing the code this problem was indeed found.

Up next, in Table IV, the inputs regarding names were put to the test. Naturally, a name is composed of only letters and this document's authors opted for a non-case sensitive approach, as it can be seen in the first two tests. Every other example should be deemed incorrect, except when the name is made up of more than one valid word (e.g. name and surname, with a space in between):

Test input	Expected Result	Assert Result
Maria	True	True
maria	True	True
Maria Silva	True	True
(empty input)	False	False
12345	False	False
Maria123	False	False
#@!?=	False	False
Maria#@!?=	False	False

TABLE IV: Results from the Name Input Tests

Once again, the class performed as expected, letting through only the inputs that followed the aforementioned rules. Of note, the class doesn't raise objections if the input has more than one space, as long as the difference between spaces and words is of just one (e.g. one space means two words, two spaces mean three words and so on).

Finally, the forth type of input comprises passwords, which should follow these guidelines: have at least one uppercase letter, one digit and one special symbol, as well as, a minimum length of 6 characters. Table V presents a possible set of password variations, both valid and invalid:

Test input	Expected Result	Assert Result
P@ssword1	True	True
#passwoRD123	True	True
(empty input)	False	False
P@ss1	False	False
password	False	False
12345	False	False
#@%&=	False	False
password123	False	False
PASSWORD123	False	False

TABLE V: Results from the Password Input Tests

Notably, the class fulfilled the expectations and seems likely to handle most, if not every, password variation successfully.

VIII. Results Discussion

The previous sections covered a broad range of testing including external dependencies and user inputs. The objective of these tests was to validate the implementation and coding of the different functionalities associated with these parts of the application.

The first set of tests, section VII-B, had two main purposes: testing the external dependency used to get the weather forecast and testing how the application processes the received data. From the documented results two main conclusions were drawn: the communication with the external API, that provides the weather data, was successfully implemented and the *parseJSON()* method, whose objective was to process the received data and store only the useful information, was also coded correctly.

In the user input section (VII-C), the main objective was to test how the application would process different types of inputs and if it would be able to identify and prevent the insertion of invalid values. Four types of inputs were tested: phone number input, email input, name input and password input.

Three out of the four test groups fulfilled the expectations, by always returning the same expected result as the actual result, but, in the email test group, one case returned a different result that the expected one. The objective of this test was to verify if a domain different that the authorised ones was able to pass the verification method. This proved to be true and the flaw was easily identified in the source code.

In conclusion, the results obtained were satisfactory, as the majority of the implemented code passed the tests. The few cases that didn't, were also very helpful because they allowed for the errors to be immediately identified, as it became easier to understand what caused an erroneous behaviour.

IX. Conclusions

The aim of this document was to present two testing frameworks used in unit testing. It was shown how these frameworks aim to automate software testing processes and increase the overall software efficiency and quality. Is was also possible to conclude that this frameworks provide faster feedback and accelerated results when compared to traditional methods.

This paper also provided a case study where these benefits were put to the test. Because the chosen application was previously developed by the authors of this document it was possible to make a detailed description of it's functionalities and how certain parts of the source code where implemented.

Is was then provided a description of what it was going to be tested within the application and how these tests would be implemented. Finally, the test results were documented and a brief resume of each one was made.

In the last section an overall result discussion is presented. Here it is described how the implemented tests allowed for code validation and how they also found some flaws within this code.

The tests applied to the case study validate what was said in the beginning of the section and prove the usefulness of this frameworks.

References

[1] Mockito. Mockito framework site. [Online] https://site.mockito.org

[2] Java Code House. Mockito Tutorial (A comprehensive guide with examples). [Online] https://javacodehouse.com/blog/mockito-tutorial/

[3] Unit tests with Mockito. [Online] https://www.vogella.com/tutorials/Mockito/article.html

[4] Unit tests with JUnit. [Online] https://www.vogella.com/tutorials/JUnit/article.html

[5] TitchersPET: O Caderno Diário de uma Educadora de Infância. [Online] https://drive.google.com/open?id=1mQq-oKLaHBQz1tFmIpdqhbWYOFQTwSto

[6] Meet Android Studio. [Online] https://developer.android.com/studio/intro