JUnit Utils User Guide

v1.0.0, 2024-04-18

Table of Contents

Abstract

JUnit Utils aim as a set of good practices and tools, is to simplify software developers approach when it comes to planning, designing, and writing unit tests. The present user guide dive into the proposed features and solution, and provide the documentation for users and developers.

There are already a lot of frameworks providing an impressive added value to software testing, like JUnit, Mockito, Hamcrest, Awaitility and many others. Nevertheless, writing tests is sometimes painful for many reasons. You surely faced the desire to modify your production class to make it more 'testable', or even skip (leave it for integration tests) a peace of code because preparing the testcase for it could/will be time consuming.

The *JUnit Utils* brings few additional tools, to ease and simplify the unit tests design and implementation approach, and therefore be gratified by a cleanest and more readable code, an impressive code coverage score, and last but not least, prevent a large amount of bugs to occur.

> *Testing leads to failure, and failure leads to understanding.*
>
> — Burt Rutan, Retired American aerospace engineer and entrepreneur

Chapter 1. Installation

1.1. pom.xml

The journey trip with *JUnit Utils* start by including it in your project *pom.xml* file.

Listing 1. JUnit Utils dependency

```
<dependency>
    <groupId>lu.mms.common</groupId>
    <artifactId>junit-utils</artifactId>
    <version>${junit-utils.version}</version>
    <scope>test</scope>
</dependency>
```

Keep up to date

The *JUnit Utils* latest version is: [**1.0.0**]

It is also important to note that this version requires **Java 11** to work.

Having this done, we do not need anything else to enjoy unit testing our code.

1.2. Maven Dependencies

The following dependencies are ready to be used as soon as *JUnit Utils* has been installed:

1. org.mybatis.mybatis
2. org.apache.commons
    - commons-text
    - commons-lang3
3. commons-io
4. logback-classic
5. org.reflections
6. spring-boot-starter-test
7. hamcrest
8. mockito
    - mockito-junit-jupiter
    - mockito-core
9. H2
10. awaitility
11. apiguardian-api

12. JUnit 5
    - junit-jupiter
    - junit-platform:
        - junit-platform-runner
        - junit-platform-engine
        - junit-platform-commons
        - junit-platform-launcher

Chapter 2. Framework tour

2.1. Configuration Parameters

While using *JUnit Utils* library, you might need to modify some default behavior. You simply have to create a file named *junit-utils.XXX*, where *XXX* stands for the configuration file type. It could be a **YAML** file or a **properties** file. example of valid configuration file names:

- junit-utils.yaml
- junit-utils.yml
- junit-utils.properties

For those you prefer to keep the JUnit configuration together, it is possible to use the JUnit platform configuration to store the *JUnit Utils* configuration in the file named *junit-platform.properties*. The table below lists the configuration entries.

Table 1. JUnit Utils properties

Property	Description	Default value
`component-scan`	The package that will be scanned by the framework	
`log-reflections`	ON/OFF the reflections logging	true
`show-banner`	Shows the library banner	true
`fancy-banner`	Shows the fancy banner if 'show-banner' = true and 'fancy-banner' = true	true

Listing 2. Example of yaml configuration

```
junit-utils:
  component-scan:   lu.mms
  log-reflections:  false
  show-banner:      true
```

Listing 3. Example of .properties configuration

```
junit-utils.show-banner=true
junit-utils.fancy-banner=false
```

2.2. Extensions API

One of the major improvement brought by *JUnit 5* is its extension model. Such extension give us the possibility to customise the test execution. A couple of extension are being proposed by *JUnit Utils* for that purpose.

2.2.1. ReturnsMocksExtension

Unit testing a class implies to mock a lot of classes around. Some of the mocked classes are often declared in the unit test in order to controller their behavior (with stubbers), and we expect from those classes the keep having the same behavior in the text context.

Let set the following context:

Listing 4. Testcase context

```
1 public class Customer {
2
3     private Identity identity;
4
5     public Identity getIdentity() {
6         return identity;
7     }
8 }
9
10 public class Identity {
11
12     private final Integer id = 987456;
13
14     public Integer getId() {
15         return id;
16     }
17 }
18
19 public class Report {
20
21     private Customer customer;
22
23     public Integer getCustomerId() {
24         return customer.getIdentity().getId();
25     }
26
27     public Customer getCustomer() {
28         return customer;
29     }
30 }
```

When unit testing the *CustomerManager* let say we would like to check that the *CustomerManager* return the right ID for a given customer, and that the correct method is fired in *Identity*. Doing this require a lot of operation to be done: **(1)** mock *Identity*, **(2)** stub to *getId()* so that it returns the expected ID, **(3)** mock *Customer* and **(4)** stub *getIdentity()* to return *IdentityMock*.

Listing 5. Unit test straight approach

```
1 class MockExample1Test {
2
```

```
3     private static final Integer ID = 500;
4
5     private Report sut;
6     private Customer customerMock;
7     private Identity identityMock;
8
9     @BeforeEach
10    void init() {
11        // Prepare mocks
12        identityMock = Mockito.mock(Identity.class);
13        customerMock = Mockito.mock(Customer.class);
14        when(customerMock.getIdentity()).thenReturn(identityMock);
15
16        // Prepare SUT
17        sut = new Report();
18        ReflectionTestUtils.setField(sut, "customer", customerMock);
19    }
20
21    @Test
22    void shouldConfirmCustomerIdWhenIdentityHasValidId() {
23        // Arrange
24        when(identityMock.getId()).thenReturn(ID);
25
26        // Act
27        final Integer id = sut.getCustomerId();
28
29        // Assert
30        assertThat(id, equalTo(ID));
31
32        // assert that the mock returned is the one we declared.
33        assertThat(sut.getCustomer().getIdentity(), equalTo(identityMock));
34    }
35 }
```

Please note that the more behaviors to stub and cases to verify, the more we will have to repeat the *init()* section in our example. So lets try to automatise this initialization process. To do this, let us: **(1)** comment the stub *when(customerMock.getIdentity()).thenReturn(identityMock);*, **(2)** configure *customerMock* to return mocks (see line 14), and **(3)** and run the test again.

a bit surprised ... the test failed.

This is because the returned mock (by *customerMock*) is not the same as the one we declared in the our test. Just modify the asserts to confirm this state as you can see in the bellow listing.

Listing 6. Simplifying the testcase initialization step: attempt #1

```
1 class MockExample2Test {
2
3     private static final Integer ID = 500;
4
```

```
    private Report sut;
    private Customer customerMock;
    private Identity identityMock;

    @BeforeEach
    void init() {
        // Prepare mocks
        identityMock = Mockito.mock(Identity.class);
        customerMock = Mockito.mock(Customer.class, Answers.RETURNS_MOCKS);
        // when(customerMock.getIdentity()).thenReturn(identityMock);

        // Prepare SUT
        sut = new Report();
        ReflectionTestUtils.setField(sut, "customer", customerMock);
    }

    @Test
    void shouldNotFindCustomerIdWhenIdentityMockDiffers() {
        // Arrange
        when(identityMock.getId()).thenReturn(ID);

        // Act
        final Integer id = sut.getCustomerId();

        // Assert

        // We will keep in following example to assert that the mock returned
        // is the one we declared, but by now, to keep the test passing we will
        // assert that the mock is different.
        assertThat(id, notNullValue());
        verify(identityMock, never()).getId();

        // Just to confirm we are not talking about the same mock.
        assertThat(id, IsNot.not(ID));
        assertThat(sut.getCustomer().getIdentity(), IsNot.not(identityMock));
    }
}
```

Assuming we are declaring a mock in our test class, we would like to interact with that mock all over the test run, and verify his behavior if needed. It comes to us that in the context of this JUNit 5 custom extension, to have a unique mock instance in the testcase context.

Fortunately, **@ReturnsMocksExtension** helps to resolve this anomaly and returns the mock declared in our test, so that any predefined behavior or stub applies to the test case. We just need to extend our class with the **ReturnsMocksExtension**:

Listing 7. ReturnsMocksExtension installation

```
@ExtendWith(ReturnsMocksExtension.class)
class MockExample3Test {
```

```
3     // ...
4 }
```

Of course, to keep the code readable and let the developer focus in the test case, lets get rid of this initialization phase, which can grow up really fast with the classes to mock and behaviors to stub, and add the **MockitoExtension** as well.

Listing 8. Simplifying the testcase initialization step: attempt #2

```
1 @ExtendWith({MockitoExtension.class, ReturnsMocksExtension.class})
2 class MockExample4Test {
3
4     private static final Integer ID = 500;
5
6     @InjectMocks
7     private Report sut;
8
9     @Mock(answer = Answers.RETURNS_MOCKS)
10    private Customer customerMock;
11
12    @Mock
13    private Identity identityMock;
14
15    @Test
16    void shouldConfirmCustomerIdWhenIdentityIsInitialized() {
17        // Arrange
18        when(identityMock.getId()).thenReturn(ID);
19
20        // Act
21        final Integer id = sut.getCustomerId();
22
23        // Assert
24        // confirms that the mock returned is the one we declared in the test class
25        assertThat(id, equalTo(ID));
26        assertThat(sut.getCustomer().getIdentity(), equalTo(identityMock));
27    }
28 }
```

As you already noted, the example **#4** if much more elegant and simple to read, understand and maintain than example **#1**. We can easily focus on the purpose of the test, by removing the noisy code. Of course, this example is really simplified but as soon as the code grows up, the proposed approach is really helpful.

2.2.2. MockInjectionExtension

When coding, it happens that we have to deal with beans which shares one or more common interfaces, or simply, extends the same base classes. Testing then is naturally simple, but we have to put some effort on declaring/creating the related mocks, prepare the collection and inject them

personally ensure that they will be injected in the correct field, which some chance for the field to be renamed for example.

The *MockInjectionExtension* will help a lot here by doing it for us, and again, help us to keep the noisy code out of what we should focus on. It is also important to notice that this extension will inject the mocks & spies declared by the user in the test class instance. Last but not least, the subject under test is instantiated via appropriate constructor, setter and even lookup methods ('@Autowired' method) with the corresponding mocks as arguments. By mocks here, we understand the ones declared by the user in the test class, otherwise a new mock instance will be created.

Listing 9. MockInjectionExtension & mock/spy injection.

```
@ExtendWith({MockitoExtension.class, MockInjectionExtension.class})
class MockInjectionExtensionTest {

    private static final String BY_CONSTRUCTOR_TEMPLATE = "by_constructor_%s";
    private static final String BY_METHOD_TEMPLATE = "by_method_%s";

    @InjectMocks
    private Computer sut;

    @Mock
    private PointingDevice pointingDeviceMock;

    @Mock
    private Mice miceMock;

    @Spy
    private Keyboard keyboardSpy;

    @Test
    void shouldInjectMockViaAutowiredConstructor() {
        // Arrange

        // Act
        final Map<String, Device> items = sut.getDevicesByConstructor();

        // Assert
        assertThat(items, IsNot.not(anEmptyMap()));
        assertThat(items.values(), hasItem(oneOf(pointingDeviceMock, miceMock,
keyboardSpy)));
    }

    @Test
    void shouldInjectMockWhenAutowiredMethodCalled() {
        // Arrange

        // Act

        // Assert
```

```
38          assertThat(sut.getValueTwo(), notNullValue());
39          assertThat(sut.getDevices(), notNullValue());
40      }
41
42      @Test
43      void shouldInitTheMocksCollection() {
44          // Arrange
45
46          // Act
47
48          // Assert
49          assertThat(sut.getCollection(), iterableWithSize(3));
50          assertThat(sut.getList(), iterableWithSize(3));
51          assertThat(sut.getSet(), iterableWithSize(3));
52          assertThat(sut.getArray(), arrayWithSize(2));
53      }
54
55      static class Computer {
56          // --- attributes initialized via constructor
57          private final Map<String, Device> devicesByConstructor;
58
59          // --- attributes initialized via @Autowired methods
60          private Object valueTwo;
61          private Map<String, Device> devices;
62
63          private Collection<Device> collection;
64
65          private List<Device> list;
66
67          private Set<Device> set;
68
69          private PointingDevice[] array;
70
71          public Computer(final List<Device> devices) {
72              this.devicesByConstructor = Stream.ofNullable(devices)
73                      .flatMap(Collection::stream)
74                      .collect(Collectors.toMap(
75                              item -> String.format(BY_CONSTRUCTOR_TEMPLATE,
    MockUtil.getMockName(item).toString()),
76                              Function.identity())
77                      );
78          }
79
80          @Autowired
81          private void initOne(final String aValue) {
82              this.valueTwo = StringUtils.defaultString(aValue);
83          }
84
85          @Autowired
86          private void initTwo(final List<Device> aValue) {
87              this.devices = aValue.stream().collect(Collectors.toMap(
```

```
88                      item -> String.format(BY_METHOD_TEMPLATE, MockUtil.
   getMockName(item).toString()),
89                      Function.identity()));
90          }
91
92          public Map<String, Device> getDevicesByConstructor() {
93              return devicesByConstructor;
94          }
95
96          public Object getValueTwo() {
97              return valueTwo;
98          }
99
100         public Map<String, Device> getDevices() {
101             return devices;
102         }
103
104         public Collection<Device> getCollection() {
105             return collection;
106         }
107
108         public List<Device> getList() {
109             return list;
110         }
111
112         public Set<Device> getSet() {
113             return set;
114         }
115
116         public PointingDevice[] getArray() {
117             return array;
118         }
119     }
120
121     // Interface used to group the mocks
122     interface Device {
123         // empty interface
124     }
125
126     // simple example of use of the interface
127     static class PointingDevice implements Device {
128         // empty class
129     }
130
131     // extending another a class implementing the desired interface
132     static class Mice extends PointingDevice {
133         // empty class
134     }
135
136     // Another interface implementation to check multiple interface implementation
137     static class Keyboard implements Device {
```

```
138         // empty class
139     }
140
141 }
```

2.2.3. ReinforceMockExtension

This extension is purely technical. It helps to avoid NPE when lifecycle methods are executed, and some method members were not mocked at all or simply injected. This extension will scan the test instance, recover the properties annotated with @Mock, and inject the empty fields with the declared mocks when relevant.

Listing 10. ReinforceMockExtension in action.

```
1  @ExtendWith({MockitoExtension.class, ReinforceMockExtension.class})
2  class ReinforceMockExtensionExampleTest {
3
4      @Mock
5      private Report reportMock;
6
7      @Mock
8      private Customer customerMock;
9
10     @Mock
11     private Identity identityMock;
12
13     @Test
14     void shouldConfirmThatReportMockPropertiesAreAllDefaultedWenRelevant() {
15         // Arrange
16
17         // Act
18         final Customer customer = (Customer) ReflectionTestUtils.getField
   (reportMock,"customer");
19
20         // Assert
21         assertThat(customer, notNullValue());
22         assertThat(customer, equalTo(customerMock));
23         assertThat(MockUtil.isMock(customer), equalTo(true));
24     }
25
26     @Test
27     void shouldConfirmThatMockPropertiesAreAllDefaultedWenRelevant() {
28         // Arrange
29
30         // Act
31         final Identity identity = (Identity) ReflectionTestUtils.getField
   (customerMock,"identity");
32
33         // Assert
34         assertThat(identity, notNullValue());
```

```
35          assertThat(identity, equalTo(identityMock));
36          assertThat(MockUtil.isMock(identity), equalTo(true));
37      }
```

2.2.4. MockitoSpyExtension

As many developers, we try our best to test solitary unit as the require less work in test preparation. Most of the border classes are mocked and we are just dealing with the behaviors we have prepared. But not all unit testers use solitary tests. It is some times required to define sociable units, and indeed without focusing on the neighboring classed we try to reproduce the way the unit behave in production environment. A solution to do so is ito use the spies, and of course, they are not fully initialised.

This is where the *MockitoSpyExtension* will help by injecting the declared test class mocks and spies if possible. As any other JUnit 5 extension, we just need to register the extension for the tests class spy to be initialized with relevant mock and spies.

Listing 11. MockitoSpyExtension in action

```
1 @ExtendWith(MockitoSpyExtension.class)
2 class MockitoSpyExtensionExampleTest {
3     @Spy
4     private House houseSpy;
5
6     @Mock
7     private Room roomMock;
8
9     // ...
10 }
```

In the above listing the 'roomMock' will be injected in the 'houseSpy', if of course, there is a such field (with) the same type in the 'House' class.

2.2.5. BeanLifeCycleExtension

This extension make it possible to test the bean life cycle method. One of the coolest annotation we use to use when coding is the *@PostConstruct*/*@PreDestroy* annotations. As per the *@PostConstruct* javadoc for example, a method annotated with this annotation should be invoked before the class is put into service, and therefore, before the class is being also tested. In integration test for example we do not have so much to do, as it is usually properly executed after dependency injection. Let set the following testcase context.

Listing 12. Testcase context

```
1 public class Identity {
2
3     private String id;
4
```

```
    @PostConstruct
    private void init() {
        this.id = "initial-id";
    }

    public String getId() {
        return id;
    }
}
```

Unit testing this class will required to run the *init()* method, if we want to reproduce its full behavior like in production context. Lets give a try.

Listing 13. PostConstruct Initialization: attempt #1

```
class BeanLifeCycleExtensionExample1Test {

    private final Identity sut = new Identity();

    @Test
    void shouldHaveValueInitializedWhenPostConstructExecuted() {
        // Arrange
        assumeTrue(sut.getId() == null);

        // Act
        ReflectionTestUtils.invokeMethod(sut, "init");

        // Assert
        assertThat(sut.getId(), notNullValue());
    }
}
```

As in the previous section example, we need to remember to aware of all the *@PostConstruct* methods, and execute via reflection feature to init the testcase, or even worse, make the *@PostConstruct* method available for other class in the production code, just because are struggling to access it in the unit test. Moreover, renaming the *@PostConstruct* method implies aligning the test as well.

With the *BeanLifeCycleExtension*, we automatise this process, remove the noisy initialization part of the test case, and ensure that the test case will be modified only when the logic in the production class change, with optimized effort for code maintenance and code review. The improved unit test looks like this:

Listing 14. PostConstruct Initialization: attempt #2

```
@ExtendWith(BeanLifeCycleExtension.class)
class BeanLifeCycleExtensionExample2Test {

    private final Identity sut = new Identity();

```

```
    @Test
    void shouldHaveValueInitializedWhenPostConstructExecuted() {
        // Ac / Assert
        assertThat(sut.getId(), notNullValue());
    }
}
```

We can notice that the noisy code just disappeared, and the developer can focus on the behaviors to test.

2.2.6. MocksContextParameterResolver

This is a helper extension, as its name states, it is a paramter provider, which goal is to provide the mocks context to the test instance method. Any debugging and mocks context check is therefore possible to verify the value using during the test case initiatization.

2.2.7. MyBatisSqlSessionResolver

This extension make it possible to resolve useful beans, which provides methods to interact directly with SQL session and the DataSource.

Table 2. Beans provided by MyBatisSqlSessionResolver

Beans	Description
`SqlSessionFactory`	The MyBatis SQL session Factory
`SqlSession`	The MyBatis Sql Session
`JdbcTemplate`	The JDBC Template (spring-jdbc)

2.2.8. MyBatisMapperExtension

(see annotation *@MyBatisMapperTest*)

2.2.9. DisableTestMethodOnFailureExtension

This extension make it possible to skip up coming tests as soon a test method within a test class fails. In the example bellow, the execution *givenCandidateIsEntryLevel()* pass, then comes the *whenTheTestCaseFails()* which fails. In this case, there is no need to execute the third method (*thenDoNotExecuteAnyOtherTestCase()*), which is ignored thanks the usage of our extension.

Listing 15. Disabling test methods in case of failure

```
@ExtendWith(DisableTestMethodOnFailureExtension.class)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class DisableTestMethodOnFailureExampleTest {

    @Test
    @Order(1)
    void givenASuccessfulTestCase() {
```

```
 8         // GIVEN
 9     }
10
11     @Test
12     @DisplayName("When This test fail, all other tests should be ignored.")
13     @Order(2)
14     void whenThisTestCaseFails() {
15         // WHEN
16         assumeTrue(false, "Example of failure.");
17     }
18
19     @Test
20     @Order(3)
21     void thenDoNotExecuteAnyOtherTestCase() {
22         // THEN
23     }
24
25 }
```

2.3. Basic Annotations

As designed, annotation make it easy to add a given behavior to and object or a class. This is why annotations have been a good candidate to ease developer life when it comes to software testing. *JUnit Utils* comes with couples of annotations to achieve that goal.

2.3.1. *@MockValue*

When coding most of the time with properties values, referring to them via the *@Value* annotation, we need extra steps in the test case settings, like inject value to the annotated fields. Some times, the annotated field is modified after code reviews or simple code refactoring. This implies to comeback to the test and align it as well.

With the ***@MockValue*** annotation, we automatise this process and initialise the subject under test (*@InjectMock* annotated) properties. We will focus on the following context in the upcoming examples.

Listing 16. Testcase context

```
1 public class Identity {
2
3     @Value("${identity-default-value}")
4     private String id;
5
6     public String getId() {
7         return id;
8     }
9 }
```

To unit this class, we usually have to **(1)** modified the production code by adding a setter, or **(2)**

within the testcase, inject a value to the *id* variable.

Listing 17. Injecting a value to the @Value field.

```
1 class MockValueExample1Test {
2
3     private final Identity sut = new Identity();
4
5     @Test
6     void shouldFindIdWhenIdManuallySet() {
7         // Arrange
8         assumeTrue(sut.getId() == null);
9
10        final String id = "id_123";
11        ReflectionTestUtils.setField(sut, "id", id);
12
13        // Act / Assert
14        assertThat(sut.getId(), equalTo(id));
15    }
16 }
```

This operation look quit simple, but lets think about any change in the production code, as shown in *MockValueExample1Test* listing. This implies to align the test with the code, and more code to review for the colleagues. This unit test can be improved using the **@MockValue** (and **MockValueExtension**). Please take a look at the reworked previous unit test case.

Listing 18. MockValue and MockValueExtension in action.

```
1 @ExtendWith({MockitoExtension.class, MockValueExtension.class})
2 class MockValueExample2Test {
3
4     @InjectMocks
5     private Identity sut;
6
7     @MockValue("${identity-default-value}")
8     private String idDefaultValue = "id_123";
9
10    @Test
11    void shouldFindTheIdWhenInitializedWithMockValue() {
12        // Act / Assert
13        assertThat(sut.getId(), equalTo(idDefaultValue));
14    }
15 }
```

One of the coolest functionality brought by ***@MockValue*** and its extension, is the benefit of the @Value defaulting. It simply work in a last move wins mode. That means:

1. First we consider the code defaulting
2. Then the test case defaulting

3. and then if an explicit value is set to the property then it is the one which will be considered (injected).

Listing 19. Context: @Mock with defaulting.

```
private static class DummyTarget {

    @Value("${int_property:5}")
    private int intProperty;

    @Value("${long_property:30}")
    private long longProperty;

    @Value("${string_property:hello}")
    private String stringProperty;

    @Value("${int_array_property:236}")
    private int[] intArrayPropertyWithSingleElement;

    @Value("${long_array_property:78,56}")
    private long[] longArrayWithMultipleElements;

    int getIntProperty() {
        return intProperty;
    }

    long getLongProperty() {
        return longProperty;
    }

    String getStringProperty() {
        return stringProperty;
    }

    int[] getIntArrayPropertyWithSingleElement() {
        return intArrayPropertyWithSingleElement;
    }

    long[] getLongArrayWithMultipleElements() {
        return longArrayWithMultipleElements;
    }

}
```

Listing 20. MockValue defaulting example.

```
@MockValue(
    value = "${int_property:10}",
    testcase = "shouldInitAttributedWhenPropertyIsDefaulted"
)
```

```
5 private Integer defaultedIntegerProperty;
6
7 @Test
8 void shouldInitAttributedWhenPropertyIsDefaulted() {
9     // Arrange / Act
10    final Integer value = sut.getIntProperty();
11
12    // Assert
13    assertThat(value, notNullValue());
14    assertThat(value, equalTo(10));
15 }
```

Mockito inclusion

You probably noted we have included Mockito *@InjectMocks* in the unit test. The benefit is to have the subject under test initialized before any other operation, and the custom extension will look for the field annotated with *@InjectMocks* and inject the desired values.

Extension Registration order

As per JUnit 5 user guide, the extensions registered via *@ExtendWith* will be executed in the order in which they are declared in the source code. So be aware of the interaction between extensions when using them together.

Please find the properties provided by *@MockValue* in the next table.

Table 3. @MockValue properties

Property	Description	Default value
`value`	An array of values to look for in the subject under test	
`testcase`	The test method where to apply the value	

 # value

The value is an array of String, that's define that should be initialized. Lets say we have the following entity:

Listing 21. Class with multiple properties annotated with @value.

```
1 public class Customer {
2
3     @Value("${customer-mother-name}")
4     private String motherName;
5
6     @Value("${customer-father-name}")
7     private String fatherName;
8
9     public String getMotherName() {
10        return motherName;
```

```
    }

    public String getFatherName() {
        return fatherName;
    }
}
```

When unit testing this class, we would like to initialize all the annotated properties with default value. With the traditional way, we need to set each property (via reflection for example). With *@MockValue#value* we simply declare all the target fields as show bellow.

Listing 22. @MockValue#value in action.

```
@ExtendWith({MockitoExtension.class, MockValueExtension.class})
class MockValueExample3Test {

    @InjectMocks
    private Customer sut;

    @MockValue({"${customer-mother-name}", "${customer-father-name}"})
    private String familyName = "no_name";

    @Test
    void shouldFindTheIdWhenInitializedWithMockValue() {
        // Act / Assert
        assertThat(sut.getFatherName(), equalTo(familyName));
        assertThat(sut.getMotherName(), equalTo(familyName));
    }
}
```

 # testcase

Some times we want to initialize the values for some test cases and not the other. We can achieve this with *@MockValue#testcase* as follows:

Listing 23. @MockValue#testcase in action.

```
@ExtendWith({MockitoExtension.class, MockValueExtension.class})
class MockValueExample4Test {

    @InjectMocks
    private Customer sut;

    @MockValue(
        value = {"${customer-mother-name}", "${customer-father-name}"},
        testcase = "shouldFindTheIdWhenInitializedWithMockValue"
    )
    private String familyName = "no_name";

    @Test
```

```
14      void shouldFindTheIdWhenInitializedWithMockValue() {
15          // Act / Assert
16          assertThat(sut.getFatherName(), equalTo(familyName));
17          assertThat(sut.getMotherName(), equalTo(familyName));
18      }
19
20      @Test
21      void shouldNotInitPropertyWhenTestcaseNotMatching() {
22          // Act / Assert
23          assertThat(sut.getFatherName(), nullValue());
24          assertThat(sut.getMotherName(), nullValue());
25      }
26  }
```

2.3.2. *@MyBatisMapperTest*

One part of application development is oriented as well in data retrieval and manipulation from database. Mybatis mappers is being used for this purpose in many projects and as other part of the code, need to be tested. It is possible to apply the configuration manually, but ... of course to save the time we need to test and maintain the production code, this task has been automatized and standardized with the *@MyBatisMapperTest* annotation (and the relevant extension *MyBatisMapperExtension*, which registered by default by *@MyBatisMapperTest_*).

Listing 24. @MyBatisMapperTest at method level.

```
1  @MyBatisMapperTest(script = "sql/oracle/schema.sql")
2  void shouldUseSpecificConfigConfigWhenConfigAtMethodLevel(final JdbcTemplate
   jdbcTemplate) {
3      // Arrange
4      final int firstIdInDB = 1; // ID in 'data.sql'
5
6      // Act
7      final String name = sut.findCustomerNameById(firstIdInDB);
8
9      // Assert
10     assertThat(name, nullValue());
11
12     // Mock not initialized as not declared in my batis config.
13     assertThat(nonMapperBean, nullValue());
14
15     final int itemCount = ObjectUtils.defaultIfNull(jdbcTemplate.queryForObject
   ("select count(*) from CUSTOMER", Integer.class), 0);
16     assertThat(itemCount, equalTo(0));
17 }
```

Full test class example

see full class in: *MyBatisMapperTestExample1Test.java*

Listing 25. @MyBatisMapperTest at class level.

```
@MyBatisMapperTest(script = {"sql/oracle/schema.sql", "sql/oracle/data.sql"})
class MyBatisMapperTestExample2Test {

    @InjectMapper
    private CustomerMapper sut;

    @Test
    void shouldInitMyBatisMapperWhenClassConfig() {
        // Act
        final String name = sut.findCustomerNameById(1);

        // Assert
        assertThat(name, equalTo("alpha"));
    }
}
```

Please find the properties provided by *@MyBatisMapperTest* in the next table.

.*@MyBatisMapperTest* properties

Property	Description	Default value
`script`	The SQL scripts to run when configuring the DataSource	
`testIsolation`	Manage the connection to the database. If `true` the database connection will be closed after each test method	`true`

2.3.3. *@InjectMapper*

As you probable noticed in previous example (*MyBatisMapperTestExample2Test*), the subject under test is *@InjectMapper* annotated. In fact, initializing MyBatis context is not enough, we still need to retrieve the target mapper we want to test. This is where *MyBatisMapperExtension* come into the game and inject the target mapper in the field annotated with *@InjectMapper*.

2.3.4. *@SpringContextRunner*

When the project we are involved in grows up, with many modules and classes, the project configuration as a critical should keep under control to avoid tricky issues.

Spring provide the *AbstractApplicationContextRunner* which is very well suited for configuration test. However, when using directly it, we face some minor issue like:

1. trouble to add a mock in the context
2. copy/paste the code in case of multiple testcase

3. not easy maintenance of the test afterwards

JUnit Utils provide the annotation *@SpringContextRunner* to solve the points listed above and helps by way to keep the testcase readable. Please find a use case of the *@SpringContextRunner* usage (together with the related extension *SpringContextRunnerExtension.class*).

Listing 26. Testing with SpringContextRunner

```
@ExtendWith({MockitoExtension.class, SpringContextRunnerExtension.class})
class SpringContextRunnerExample1Test {

    @SpringContextRunner(
        withMocks = UserAccount.class,
        withUserConfiguration = ConfigUser.class
    )
    private ApplicationContextRunner appContextRunner;

    @BeforeEach
    void init() {
        assumeTrue(appContextRunner != null, "The <appContextRunner> cannot be
null");
    }

    @Test
    void shouldInjectMockDefinedInOutTestInTheApplicationContextWhenSimpleContext()
{
        // Arrange
        assumeTrue(appContextRunner != null, "the ApplicationContext is not
initialized");

        // Act
        appContextRunner.run(assertProvider -> {
            // Assert
            assertThat(assertProvider).doesNotHaveBean(AnnotatedComponent.class);

            // This entity is declared in the User configuration
            assertThat(assertProvider).hasSingleBean(EntityBrown.class);
        });
    }
}
```

You can denote here how easy it is to add a mock in the context. Please find the properties provided by *@SpringContextRunner* in the next table.

Table 4. @SpringContextRunner properties

Property	Description	Default value
`withActiveProfiles`	Set the active spring profiles	
`withAllowBeanDefinitionOverriding`	Allow bean definition overriding	true

Property	Description	Default value
withPropertyValues	Add the specified Environment property pairs	
withSystemProperties	Add the specified System property pairs	
withPropertySource	Apply the specified Property source	
withConfiguration	Register the specified configuration class with the ApplicationContext	NoClass.class
withUserConfiguration	Register the specified user configuration classes with the ApplicationContext	
withBeans	Register the specified user beans with the ApplicationContext	
withMocks	Register the specified classes as mocks with the ApplicationContext	
injectDeclaredMocks	Register the mocks & spies declared in the test class	true
mappersPackage	Mock the mappers under given packages	

There another point where *@SpringContextRunner* will be very interesting for us. In fact, when it comes to mock something and/or declare a mock, we need to insert it again the the context. Now with the *@SpringContextRunner* we just need to declare the mock in the test for it to be injected in the context, as shown below:

Listing 27. Friendly mocks in @SpringContextRunner

```
1 @ExtendWith({MockitoExtension.class, SpringContextRunnerExtension.class})
2 class SpringContextRunnerExample1Test {
3
4     @SpringContextRunner(
5         withMocks = UserAccount.class,
6         withUserConfiguration = ConfigUser.class
7     )
8     private ApplicationContextRunner appContextRunner;
9
10    @BeforeEach
11    void init() {
12        assumeTrue(appContextRunner != null, "The <appContextRunner> cannot be
  null");
13    }
14
15    @Test
16    void shouldInjectMockDefinedInOutTestInTheApplicationContextWhenSimpleContext()
  {
17        // Arrange
18        assumeTrue(appContextRunner != null, "the ApplicationContext is not
  initialized");
19
20        // Act
21        appContextRunner.run(assertProvider -> {
```

```
            // Assert
            assertThat(assertProvider).doesNotHaveBean(AnnotatedComponent.class);

            // This entity is declared in the User configuration
            assertThat(assertProvider).hasSingleBean(EntityBrown.class);
        });
    }
}
```

2.3.5. *@Fixture*

This is a class and field level annotation, that's enable *JUnit Utils* with powerful fixture files. As you may already know, a fixture is a standalone unit or functionality of a project. It is usually best practice, when testing that before deriving the test scripts, we should determine the fixture to be tested. Such a fixture usually contains a list of steps, which, combined in a given sequence, give a test case.

In context of *JUnit Utils*, the fixture file contains the atomic steps that can be applied to prepare a test case execution. Please take a look of an example of fixture file.

Listing 28. An example of @Fixture file.

```
@Fixture
public class FixtureFileExample {

    private Report report;

    void givenCustomerIdIsTwenty() {
        if (MockUtil.isMock(report)) {
            when(report.getCustomerId()).thenAnswer(arg -> 20);
        }
    }
}
```

Having already defined our fixture file, can therefore use it in our unit test as show below.

Listing 29. @Fixture file in action.

```
@ExtendWith({MockitoExtension.class, FixtureExtension.class})
class FixtureExample1Test {

    @Mock
    private Report reportMock;

    @Fixture
    private FixtureFileExample fixture;

    @Test
    void shouldHaveAConfiguredFixtureFile() {
        // Arrange
```

```
13          fixture.givenCustomerIdIsTwenty();
14
15          // Act
16          final Integer id = reportMock.getCustomerId();
17
18          // Assert
19          assertThat(id, equalTo(20));
20      }
21  }
```

As you already noticed, we are free of initializing the fixture file by injecting the mocks from our test class. Indeed, the mocks (and spies) declared in the test class are injected in the fixture, if the relevant field hasn't yet been initialized. Doing so help to keep cleaner and readable test and the developer can focus on the functionality to test.

The table below shows *@Fixture* annotation the properties.

Table 5. @Fixture properties

Property	Description	Default value
`injectMocks`	Inject test class (test case) mocks to the class with *@Fixture* or not	`true`

2.3.6. *@ExtendWithTestUtils*

Having all the previous features up and running, some behaviour has been defaulted to simplify unit test design, plan, implementation and configuration. The *@ExtendWithTestUtils* annotation, in addition to emphasizing the fact our test is a unit test, is setting in the main time de most common default behavior by registering the following extension (in order):

- MockitoExtension
- ReturnsMocksExtension — *[#1] ensure the created mock will return the test instance declared @Mock/@Spy*
- MockValueExtension — *[#2] initialize the @Value (via @MockValue) when relevant*
- ReinforceMockExtension — *[#3] enrich the declared mocks to avoid NPE on lifecycle methods call*
- MockitoSpyExtension — *[#4] enrich the @Spy fields with the previous (#1 and #2) knowledge*
- MockInjectionExtension — *[#5] ensure the SUT (@InjectMocks) is properly initialized ...*
- FixtureExtension
- MocksContextParameterResolver

Please find the properties provided by *@ExtendWithTestUtils* in the next table.

Table 6. @ExtendWithTestUtils properties

Property	Description	Default value
`returnMocks`	Answer (*Mockito.Answers*) with mocks defined in user test (see: *ReturnsMocksExtension*)	`true`
`initMocks`	The MockitoSettings strictness	`true`
`initSpies`	The flag to init mockito spies (@Spy) with test case mock and spies if relevant	`true`
`reinforceMock`	The flag to enable/disable the mock reinforcement	`true`

Listing 30. Testing with @ExtendWithTestUtils

```
@ExtendWithTestUtils
class ExtendWithTestUtilsExample1Test {

    @InjectMocks
    private Report sut;

    @Mock(answer = Answers.RETURNS_MOCKS)
    private Customer customerMock;

    @Mock
    private Identity identityMock;

    @Test
    void shouldEnsureThatIdentityIsRecognizedInTestcase() {
        // Arrange
        final Integer id = 456;
        when(identityMock.getId()).thenReturn(id);
        // Act
        final Integer result = sut.getCustomerId();

        // Asset
        assertThat(result, equalTo(id));
    }
}
```

2.3.7. *@WithBeanLifeCycle*

This annotation extends the test class with *BeanLifeCycleExtension*, which allows to execute and test the subject under test life cycle if relevant. The *@WithBeanLifeCycle* annotation makes also possible to decide whether to execute the life cycle method at JUnit5 '@BeforeEach' stage or '@BeforeTestExecution' stage), via its *beforeEach* property.

Table 7. @WithBeanLifeCycle properties

Property	Description	Default value
`beforeEach`	The flag to switch lifecycle methods execution between '@BeforeEach' stage or '@BeforeTestExecution' stage	`false`

2.4. Other delicacies

2.4.1. DataSourceMockBuilder

This builder has been introduced to provide a ready to go mocked DataSource. Up to the developer to declare and inject it in the test context, the spring context or event to use it directly in the test class.

Listing 31. Mocked DataSource as a Bean

```
@Configuration
class DataSourceMockExample1Test {
    @Bean
    @Primary
    public DataSource dataSource() {
        return DataSourceMockBuilder.newDataSourceMock("oracle").build();
    }
}
```

Listing 32. Verifying the mocked DataSource properties

```
@Configuration
class DataSourceMockExample1Test {

    @Test
    void shouldBuildH2DataSourceMockWhenCallingH2Builder() throws SQLException {
        // Act
        final DataSource dataSource = DataSourceMock.newH2Mock().build();

        // Assert
        assertThat(dataSource, notNullValue());
        assertThat(MockUtil.isMock(dataSource), equalTo(true));
        assertThat(MockUtil.getMockSettings(dataSource).getDefaultAnswer(),
  equalTo(Answers.RETURNS_DEEP_STUBS));

        final Connection connection = dataSource.getConnection();
        assertThat(MockUtil.isMock(connection), equalTo(true));

        final DatabaseMetaData databaseMetaData = connection.getMetaData();
        assertThat(databaseMetaData.getDatabaseProductName(), equalTo("H2"));
        assertThat(MockUtil.isMock(databaseMetaData), equalTo(true));
    }
}
```

2.4.2. DB Reverse Engineering

One of the most interesting moment when testing our code, is the time when we really feel like we are ready to submit 'production' like data to the developed feature. The easiest way to achieve this is to collect the target database extract, throw everything in '.sql' file, call it all when setting up our

datasource and that's it. It could be interesting to gather only the data related to the target test case, in with case we will be able to assert more efficiently the target DML (insert/update/delete). The proposed DB Reverse Engineering could accelerate and ease the data retrieval and analysis, and even make it possible to have the proper migration script for each test case.

Listing 33. Extracting the test case context

```
class ReverseEngineeringWizardExample1Test {

    private DataSource dataSource;

    @BeforeEach
    void init() {
        final HikariConfig dbConfig = new HikariConfig();
        dbConfig.setUsername("U_NAME");
        dbConfig.setPassword("pwd");
        dbConfig.addDataSourceProperty("poolName", "test_pool");
        dbConfig.addDataSourceProperty("maximumPoolSize", 30);
        dbConfig.setJdbcUrl("jdbc:oracle:thin:@//my_data_base:1521/DB_NAME");
        dataSource = new HikariDataSource(dbConfig);
    }

    /**
     * Should create mig script in the same package when package provider exists.
     * We will be using DDL for this test, but the same works for DML as well.
     */
    @Test
    void shouldCreateMigScriptInSamePackageWhenPackageProviderExists(){
        //  Arrange
        final Schema schema = new ReverseEngineeringWizard(
                dataSource,
                "SCHEMA_NAME"
        ).withTable(
                "CUSTOMER",
                // match in the CUSTOMER table
                Expression
                    .value("CD_TYPE").eq("AXE")
                    .and(Expression.value("CD_LANGUE").in("FR", "EN"))
                    .or(Expression.value("ID_CUST").between(468, 700)),
                Expression.value("CITY").in("Luxembourg", "Madrid", "Paris")
        ).build();
        assumeFalse(schema.getTables().isEmpty());

        // DDL with package provider
        final Ddl ddl = Ddl.with(schema, this.getClass());

        // Act
        // create the DDL file in the same package as this class
        final boolean ddlCreated = ddl.createFile();

        // Assert
```

```
45          assertThat(ddlCreated, equalTo(true));
46      }
47
48      /**
49       * Should create mig script in the 'test/resources/sql' folder when no package
   provider.
50       * We will be using DDL for this test, but the same works for DML as well.
51       */
52      @Test
53      void shouldCreateMigScriptInSamePackageWhenMissingPackageProvider(){
54          //  Arrange
55          final Schema schema = new ReverseEngineeringWizard(
56                  dataSource,
57                  "SCHEMA_NAME"
58          ).withTable(
59                  "CUSTOMER",
60                  // match in the CUSTOMER table
61                  Expression.value("CD_TYPE").eq("AXE"),
62                  Expression.value("CD_LANGUE").in("FR", "EN", "LU"),
63                  Expression.value("ID_CUST").between(468, 700)
64          ).build();
65          assumeFalse(schema.getTables().isEmpty());
66
67          // DML without package provider
68          final Ddl ddl = Ddl.with(schema);
69
70          // Act
71          // create the DDL file in the 'test/resources/sql' folder
72          final boolean ddlCreated = ddl.createFile();
73
74          // Assert
75          assertThat(ddlCreated, equalTo(true));
76      }
77
78  }
```

Example of DDL generated at the [ReverseEngineeringWizardExample1Test] line 37, will look like the following one:

```
1  CREATE SCHEMA IF NOT EXISTS SCHEMA_NAME;
2
3  DROP TABLE SCHEMA_NAME.CUSTOMER IF EXISTS; --null
4
5  CREATE TABLE SCHEMA_NAME.CUSTOMER (
6      ID_CUST VARCHAR2(5) PRIMARY KEY
7  ,   CD_TYPE VARCHAR2(3) NOT NULL
8  ,   CD_LANGUE VARCHAR2(2)
9  ,   LI_NAME VARCHAR2(20) NOT NULL
10 ,   LI_SURNAME VARCHAR2(30)
11 ,   LI_COMMENT VARCHAR2(600)
```

```
12 ,   CONSTRAINT CHK_CUSTOMER CHECK (
13         CD_LANGUE IN ('FR', 'NL', 'EN', 'DE') and CD_TYPE IN ('AXE', 'PPT')
14     )
15 );
```

And here is an example of DML generated at the [ReverseEngineeringWizardExample1Test] line 38:

```
1 DELETE FROM  SCHEMA_NAME.CUSTOMER;
2
3 INSERT INTO SCHEMA_NAME.CUSTOMER (
4     (ID_CUST, CD_TYPE, CD_LANGUE, LI_NAME, LI_SURNAME, LI_COMMENT)
5 VALUES
6     (8356, 'AXE', 'FR', 'FITZGERALD', 'MATTEW', null)
7 ;
```

Following the DDL and DML generation, it is possible to persist then to .sql (in the `test/resources` file using the method `Ddl.createFile` and `Dml.createFile` like presented in [ReverseEngineeringWizardExample1Test] line 42. Please notice that a class provided (line 38) so that the target migration script (.sql file) will be created in the same package as the provided class. Otherwise, the .sql file is created in the '/sql' folder.

2.5. Specification Testing

2.5.1. ScenarioTesting

The `ScenarioNameGenerator` is a custom Test Method name generator, which turn how the test class simple name, and the test method name as well into a human readable name. In addition, each class is considered as a scenario (a 'Scenario: ' key word is added to the result class mapped name). This functionality is handling the snake case method as well.

Table 8. Examples

type	To map	Result
`class`	UserNameGeneratorTest.class	Scenario: User name generator test
`method`	shouldMapNameWhenDefaultName	Should map name when default name
`method`	should_map_name_when_snake_case_name	Should map name when snake case name

Listing 34. Example of use of ScenarioTesting

```
1 class ScenarioTestingExampleTest extends ScenarioTesting {
2
3     @DisplayName("Scenario: Feature #1")
4     @TestFactory
5     public Stream<DynamicNode> myScenariosTestFactory() {
6         return super.scenariosTestFactoryTemplate();
7     }
```

```
 8
 9     void givenNewCustomer() {
10         // Given
11     }
12
13     void whenCustomerAppliesForNewAccount() {
14         // When
15     }
16
17     void thenAccountIsCreated() {
18         // Then
19     }
20
21 }
```

2.5.2. EnumArgument

This is an annotation that allows to repeat a test scenario execution with a set of provided data, and is to be used together with the ´ScenarioTesting´ abstract class.

Table 9. EnumArgument properties

Property	Description
value	Enum class to be used (enums constants) during test scenario execution
include	The enum constants name subset, to be included during the test scenario execution
exclude	The enum constants name subset, to be excluded during the test scenario execution

Listing 35. Example of use of EnumArgument

```
 1 @EnumArgument(GenericSize.class)
 2 class EnumArgumentExampleTest extends ScenarioTesting {
 3
 4     @TestFactory
 5     public Stream<DynamicNode> shouldValidateMyFeature() {
 6         return super.scenariosTestFactoryTemplate();
 7     }
 8
 9     void givenCitizenClothesSize(final GenericSize value) {
10         // Given
11         assertThat(value, notNullValue());
12     }
13
14     void whenWeCalculateCitizenTaxes() {
15         // When
16     }
17
```

```
18     void thenTheTaxAreRatedForCategoryA() {
19         // Then
20     }
21
22 }
```

Listing 36. Example of use of EnumArgument with 'include'

```
1 @EnumArgument(
2         value = GenericSize.class,
3         include = {"L", "XL", "XXL"}
4 )
5 class EnumArgumentIncludeExampleTest extends ScenarioTesting {
6
7     @TestFactory
8     public Stream<DynamicNode> shouldValidateMyFeature() {
9         return super.scenariosTestFactoryTemplate();
10    }
11
12    void andThenCustomerHaveALargeSuitSize(final GenericSize value) {
13        // Then
14        assertThat(value, notNullValue());
15        assertThat(value, oneOf(GenericSize.L, GenericSize.XL, GenericSize.XXL));
16    }
17
18    void whenWeCalculateCitizenTaxes() {
19        // When
20    }
21
22    void givenTheCitizenCars() {
23        // Given
24    }
25
26 }
```

Listing 37. Example of use of EnumArgument with 'exclude'

```
1 @EnumArgument(
2         value = GenericSize.class,
3         exclude = {"L", "XL", "XXL"}
4 )
5 class EnumArgumentExcludeExampleTest extends ScenarioTesting {
6
7     @TestFactory
8     public Stream<DynamicNode> shouldValidateMyFeature() {
9         return super.scenariosTestFactoryTemplate();
10    }
11
12    void whenNoOneInCustomerFamilyDoNotHaveALargeSuitSize(final GenericSize value)
  {
```

```
13          // When
14          assertThat(value, allOf(
15                  notNullValue(), not(GenericSize.L), not(GenericSize.XL), not
    (GenericSize.XXL))
16          );
17      }
18
19      void thenTheTaxAreRatedForCategoryA() {
20          // Then
21      }
22
23      void givenTheCitizenFamily() {
24          // Given
25      }
26
27  }
```

2.5.3. ValueArgument

This is an annotation that allows to repeat a test scenario execution with a set of provided values, and is to be used together with the ´ScenarioTesting´ abstract class. For the moment, only strings can be provided as 'Data Provider', but other arguments sources can be added in the future such as: - data files: excels, csv, jValueArgumentson - arrays / iterables

Listing 38. Example of use of ValueArgument with 'strings'

```
1  @ValueArgument({"A", "B", "C"})
2  class StringValueArgumentExampleTest extends ScenarioTesting {
3
4      @TestFactory
5      public Stream<DynamicNode> shouldValidateMyFeature() {
6          return super.scenariosTestFactoryTemplate();
7      }
8
9      void givenCitizenType(final Object value) {
10         // Given
11         assertThat(value, notNullValue());
12         assertThat(value, oneOf("A", "B", "C"));
13     }
14
15     void thenCitizenIsChargedByTemplate(final String value) {
16         // Then
17         assertThat(value, notNullValue());
18         assertThat(value, oneOf("A", "B", "C"));
19     }
20
21     void angGivenCitizenTaxCategory() {
22         // Given
23     }
24
```

```
25     void whenCitizenTaxAreRatedForCategoryA() {
26         // When
27     }
28
29 }
```

2.5.4. ScenarioTesting

The ´ScenarioTesting´ is an abstract which brings test factory behavior to our test. The test method named with the behavioral pattern will be ordered in as per 'GIVEN, WHEN, THEN' key words and executed. In case one of the step fails, the test plan is aborted and the remaining tests method to executed are skipped.

2.5.5. SpecificationDrivenOrderer

Usually the feature (or use case) we are implementing are specified and much respect a set of rules. To be sure that our implementation match with the requirement, we use to implement some use case testing for this purpose. The `SpecificationDrivenOrderer` method orderer ease this process by order the test method in a GIVEN-WHEN-THEN order.

Listing 39. Example of test class with SpecificationDrivenOrderer and ScenarioNameGenerator

```
1  @TestMethodOrder(SpecificationDrivenOrderer.class)
2  @DisplayNameGeneration(ScenarioNameGenerator.class)
3  class SpecificationDrivenOrdererExampleTest {
4
5      @Test
6      void andGivenASimpleCitizen() {/*Given*/}
7
8      @Test
9      void andThenCitizenHouseWillNotBeCharged() {/*Then*/}
10
11     @Test
12     void thenTheTaxAreRatedForCategoryA() {/*Then*/}
13
14     @Test
15     void givenTheCitizenFamily() {/*Given*/}
16
17     @Test
18     void whenWeCalculateCitizenTaxes() {/*When*/}
19
20     @Test
21     void givenTheCitizenCarsAndHouse() {/*Given*/}
22
23     @Test
24     void givenTheCitizenCurrentJobAndPosition() {/*Given */}
25
26     @Test
27     @DisplayName("Given the citizen pets")
28     void citizen_has_pets() {/* Given*/}
```

```
29
30 }
```

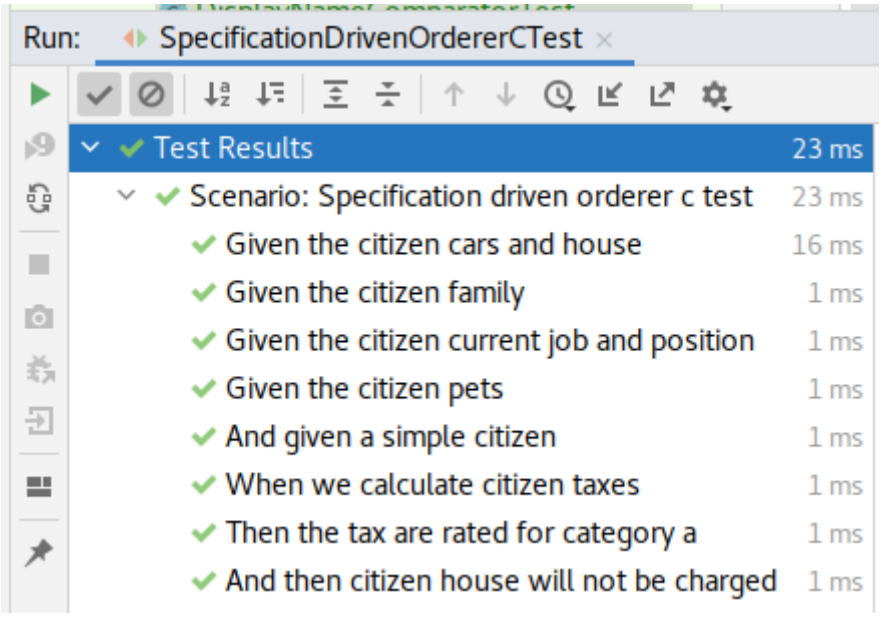

Figure 1. and the resulting test plan output

Chapter 3. Future works

There are still many subject uncovered in the *JUnit Utils* framework and in this user guide as well, that are in the area of interest of most of the developers. Including some tools for integration tests are in the bags as well, as integration with tools like *Cucumber* and *Selenium*, when it comes to integration tests.

Chapter 4. Last thought

The solutions and features presented in this user guide presents a the general approach we adapted to improve the tests quality. We hope they will be helpful to solve impediment software developer are facing during coding and testing. In case they do not fit to your project, do not hesitate to contribute with new ideas and propositions.

Happy coding !