***TDD:*** is a way of writing code that involves writing an automated unit-level test that first fails, then writing enough code to make the test pass.

***BDD:*** Behavior Driven Development is an *agile* methodology in which an application is designed around the behavior a user expects to experience when interacting with it.

BDD is a refinement of *TDD*. In BDD first we should specify the behavior of the system based in terms of user stories. Each user story should follow this structure:

**Title:** Returns and exchanges go to inventory.

**As a** store owner,

**I want** to add items back to inventory when they are returned or exchanged,

**so that** I can sell them again.

Acceptance of criteria:

We define a few scenarios with the following structure

**Given:** initial context at the beginning of the scenario

**When:** the event that triggers the scenario

**Then:** the expected outcome

***Spock*** is a test framework that uses the *Groovy* programming language.

***Pock vs Junit:***

* Parametrization: technique of changing test data for the same test method. In *spock* we use *data tables.*
* Mocking: the technique of faking external objects behavior. *In junit* we need 3rd party libraries like **Mockito** but in *spock* we need nothing.
* Documenting tests and codes: the syntax of writing test in spock is very self-descriptive. (using **def “should…”** and also blocks like **given, when, then**)
* Versification: to assure a method is executed based on some conditions:

|  |
| --- |
|  |
|  |
|  |

*Spock blocks:*

* **Given**: contains initialization code to prepare the unit test. We can have spock test without *given* but makes it less readable. (expected usage 85%)
* **Setup:** exactly the same as *given*. (0%)
* **When:** contains code to trigger the action of the test target. Should be as short as possible.
* **Then:** to verify the correct behavior of the class under test
* **And:** it allows us to split other blocks into distinctive parts.
* **Expected:** combines the meaning of *given-when-then* trinity. Could be used for very simple tests that need no initialization code.
* **Where:** used for parametrized tests.
* **Cleanup:** to close and release all the resources

|  |
| --- |
|  |

* Using metadata in spock for clarity also has this characteristic that they could be extracted by reporting tools.
* Use @Subject to mark the class under test
* The class which contains test methods must extends *Specification*. This makes it a spock test.
* We cannot name the class with full English text but instead we could use @Title.

The method *setup()* runs before each test. And *cleanup()* runs after each test.

The method *setupSpec()* runs only once before all test methods. (for expensive objects)

The method *cleanupSpec()* runs only once after all test methods.

*setupSpec* and *cleanupSpec* access only static fields and *@shared* objects.

*@shared* determines an object to survive across all tests.

* Ensure that test are self-documenting otherwise it would be less readable

|  |
| --- |
|  |

* If we override *toString()* method, then it would be printed to the output on case of test failure.
* In the *where* block, single pipes denotes a column and the double pipes shows where the input parameters stop and the output parameters start.

|  |
| --- |
|  |

*Where* block has some limitations:

* Must be the last block but can be followed by an *and* block
* They must have at least two columns
* If our test has only one parameter, then we must use **filter** for the second column

|  |
| --- |
|  |

Each scenario in *where* block runs as an individual independent method.

We could also use *table data* in the method name:

|  |
| --- |
|  |

We could also use **left-shift**(<<)operator in the data table:

|  |
| --- |
|  |

***Needs to be completed***(parametrized test)

***MOCKING and STUBBING***

dependency

Email sender

File to email format convertor

File

If we want to write a unit test for the *FormatConvertor* class, we should employ **fake collaboration** to fake the *EmailSender*’s **correct** behavior.

* We only fake the needed method.

**Stub**: a fake class which comes with a preprogrammed returned value(mainly for input).

**Mock:** is a class that can be examined after the test is finished (e.g. for verification).

**Spy:** a real class in which only a subset of methods are fake(partial mock).

***STUBING***

Our class under test requests to our stub and we need to tell Spock what to do when any of the stub methods are called.

Two steps are needed:

* Showing *Spock* which class is not supposed to use its real implementation.
* Declaring what will happen in case of stub method invocation.

|  |
| --- |
|  |

If we don’t care what the argument of the method is, we could use *underscore*(\_).

We can also use *groovy closures*:

|  |
| --- |
|  |

* **Spock also supports *recursive* *stubbing*.**

***MOCKING***

* All capabilities in *stubs* exist in *mocks* as well.

Check whether a method was called:

|  |
| --- |
|  |

Verify the order of interactions



This does not check the order

**BUT**

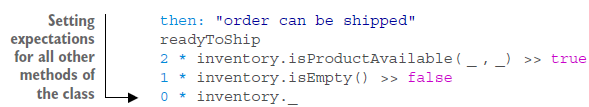
****

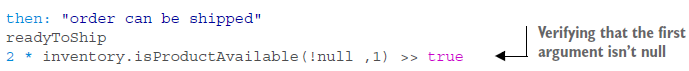
****

**NOTE**

Calling a mocked method which does not have any explicit instruction(e.g. returned value), returns default values. Therefore if we need to make sure that a method is not called, we will write:

|  |
| --- |
|  |







We want to make sure that a method is called with the first argument as *1550* and the second value is an object of type client and its *vip* field is *false.*

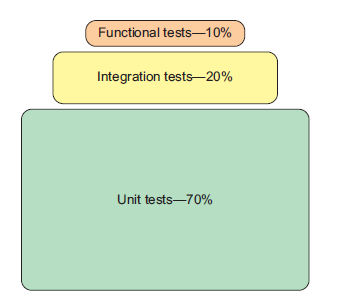


***SPY***

* A well-designed code does not need *spies*.
* If we need *spy* for unit testing, we had better refactor our code.



***TESTING PYRAMID***



Unit test has a heavy weight since it makes no sense to create a complex integration test if we are not sure about the quality of the java classes that compose them.

Using the book “Java Testing with Spock” and also wikipedia