2 Lab: Unit tests with dependency mocking

v2025-02-19

Learning objectives

— Run unit tests (JUnit 5) and mocks (Mockito), with mocks injection (@Mock).

— Experiment with mock behaviors: strict/lenient verifications, advanced verifications, etc.

Useful resources

— B. Garcia's chapter on Jupiter and Mockito integration.

2.1 Stocks portfolio

Consider the example in Figure 1: the StocksPortfolio holds a collection of Stocks; the current value of the *portfolio* depends on the current condition of the *Stock Market*. **StockPortfolio#totalValue()** method calculates the value of the portfolio (by summing the current value of owned stocks, looked up in the stock market service).

Implement (at least) one test to verify the implementation of **StockPortfolio#totalValue().**

Given that test should have predictable results, you need to address the problem of having non-deterministic answers from the **IStockmarketService** interface.

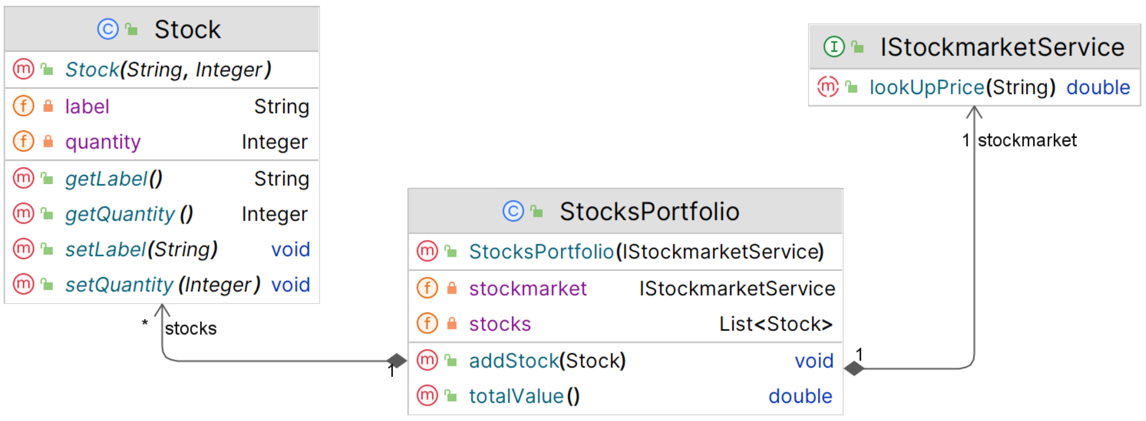

Figure 1: Classes for the StocksPortfolio use case.

a) Create the classes. You may write the implementation of the services before or after the tests.

b) Create the test for the totalValue(). As a guideline, you may adopt this outline:

```
1. Prepare a mock to substitute the remote stockmarket service (@Mock annotation)
```

```
2. Create an instance of the subject under test (SuT) and use the mock to set the
   (remote) service instance.
3. Load the mock with the proper expectations (when...thenReturn)
4. Execute the test (use the service in the SuT)
5. Verify the result (assert) and the use of the mock (verify)
```

Notes:

- Consider use these Maven dependencies for your POM (JUnit5, Mockito).
- Mind the JUnit version. For JUnit 5, you should use the **@ExtendWith** annotation to integrate the Mockito framework.
- Check this example for a quick view on main syntax options and operations.

c) In the previous task, you have implemented a test with a few expectations declared to instruct the mock to provide a few canned returns. Add more expectations; in fact, add more "stocks" types in the expectations that those that will be used/needed in the test.

Run the test. Note the Mockito warnings.

d) Instead of the default JUnit (Jupiter) core assertions, use another assertions library. Consider using the Hamcrest or AssertJ to create more human-readable assertions. Refactor your code. See example.

e) Add a new method to StocksPortfolio to return the top *n* most valuable shares. For example, n=3 would return a list with the 3 most valuable stocks in the portfolio (considering their quantity times the market value). Consider using an AI tool to implement the method.

```
/**
 * @param topN the number of most valuable stocks to return
 * @return a list with the topN most valuable stocks in the portfolio
 */
public List<Stock> mostValuableStocks(int topN) {  …. }
```

f) ....but do not trust AI-generated code to be correct!

Write tests for the new code yourself. Did the AI offer a correct implementation? Are edge cases properly covered? (Prove with tests.)

2.2 Conversion method behavior

If you write code in a TDD style, you'll quickly run into the situation where your code references some (often third-party) dependency or subsystem. You'll typically want to ensure that the code under test is isolated from that dependency.

Consider an application that needs information from a product catalog provided by a B2B partner. The "partner" products API is documented at https://fakestoreapi.com/docs

You will consume the remote data through an API, using an HTTP client, but the team has not yet chosen the implementation or library to use.

a) Create the objects represented in Figure 2. Consider adopting a TDD approach and create just the classes "skeleton" (the code should compile but it is not expected to work, for now).

b) Your first test will focus on the behavior of ProductFinderService#findProductDetails(id)

This method takes a product *id number* and gets the product information from the Products API. For that, ProductFinderService uses an HTTP Client to invoke an HTTP GET method.

Consider that:

- the team has not yet decided on the HTTP Client to use;
- each API call has a cost and the team wants to minimize costs at development time.
- the main aspect to be tested is whether *findProductDetails* can instantiate well formed objects from the JSON obtained in the API response.

How would mocking be used in this scenario?

c) Implement the following tests cases to verify the behavior of *findProductDetails*:

   a. *findProductDetails(**3**)* returns a valid Product object, with id = 3 and title = "Mens Cotton Jacket".

   b. *findProductDetails(**300**)* returns no Product (could return an empty Optional).

   Note: you want to mock the response that doHttpGet() would produce.

d) Implement the required code for the tests to pass.

   You should implement the parsing logic inside findProductsDetails (parse JSON into POJO).

   Remember you don't have yet the implementation for the HTTP Client (and you are not expected to use a real one for now).

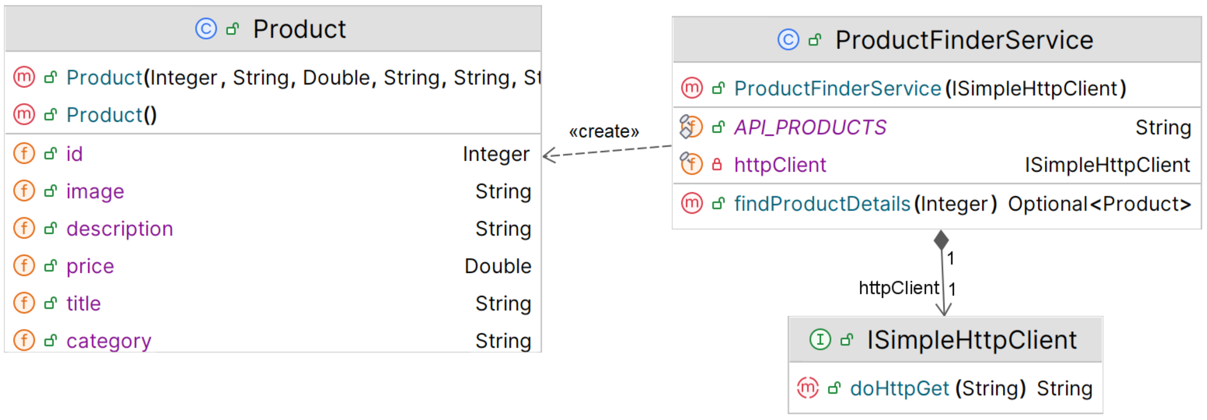
Product
Product(Integer, String, Double, String, String, St
Product()
id Integer
image String
description String
price Double
title String
category String
«create»
ProductFinderService
ProductFinderService (ISimpleHttpClient)
API_PRODUCTS String
httpClient ISimpleHttpClient
findProductDetails (Integer) Optional<Product>
1
httpClient 1
ISimpleHttpClient
doHttpGet (String) String

Figure 2: Classes for the geocoding use case.

2.3 Connect to remote resource and integration tests

The mocked HTTP Client was useful, but, at some point, you also need to test against the real remote API Service.

Consider you are implementing an **integration test** for the previous example; in this case, you would use the real implementations of the modules in the test (no mocks).

a) Implement a working version of an HTTP Client compatible with ISimpleHttpClient. You may base your code on this sample.

b) Be sure the "failsafe" maven plugin is configured. Use the same (up to date!) version for *surefire* and *failsafe* plugins.

c) Create a test class (ProductFinderService**IT**) in a separate file (be sure the suffix is “**IT**”). Copy the tests from the previous exercise into this new test class and remove any support for mocking (**no Mockito imports** in this test).

d) Correct/complete the test implementation so it uses the **real HttpClient** implementation.

e) Run your test. Confirm that the remote API is being invoked in the test execution (e.g.: by connecting/disconnecting from the Internet).

   Confirm that your test is passing.

```
$ mvn failsafe:integration-test
```

f) Concerning test execution, take note of the differences between the following cases and provide an explanation.

```
$ mvn test
$ mvn package
$ mvn package -DskipTests=true
$ mvn failsafe:integration-test
$ mvn install
```