TQS Lab activities

v2021-03-24

Lab 1: Unit testing with JUnit 5

Learning objectives

- Identify relevant unit tests to verify the contract of a module.
- Write and execute unit tests using the JUnit framework.
- Link the unit tests results with further analysis tools (e.g.: code coverage)

Key points

- Unit testing is when you (as a programmer) write test code to verify units of (production) code. A unit represents a small subset of a much larger solution. A true "unit" does not have depend of the behavior of other (collaborating) components.
- Unit tests help the developers to (i) understand the module contract (what to construct); (ii) document the intended use of a component; (iii) prevent regression errors; (iv) increase confidence on the code.
- When following a TDD approach, typically you go through a cycle of Red-Green-Refactor. You'll run a test, see it fail (go red), implement the simplest code to make the test pass (go green), and then refactor the code so your test stays green and your code is sufficiently clean.
- JUnit and TestNG are popular frameworks for unit testing in Java.

> **JUnit best practices: unit test one object at a time**
> A vital aspect of unit tests is that they're finely grained. A unit test independently examines each object you create, so that you can isolate problems as soon as they occur. If you put more than one object under test, you can't predict how the objects will interact when changes occur to one or the other. When an object interacts with other complex objects, you can surround the object under test with predictable test objects. Another form of software test, integration testing, examines how working objects interact with each other. See chapter 4 for more about other types

Lab activities

Be sure that your developer environment meets the following requirements:

- Java development environment (JDK; v11 suggested). Note that you should install it into a path without spaces or special characters (e.g.: avoid \Users\José Conceição\Java).
- Maven configured to run in the command line. Note: some projects include the Maven wrapper utility (*mvnw*); in these cases, Maven wrapper will download maven as needed.
- Java capable IDE, such as IntelliJ IDEA (version "Ultimate" suggested).

1.1

In this exercise, you will implement a stack data structure (TqsStack) with appropriate unit tests. Be sure to adopt a **write-the-tests-first** workflow:

a) Create a new **Maven-based,** Java standard application.

b) Add the required dependencies to run Junit tests[1]. Here are some examples:
   - JUnit documentation
   - starter project for Maven[2].

c) Create the required classes definition (**just the "skeleton"**, do not implement the methods body yet; you may need to add dummy return values). The **code should compile**, though the **implementation is incomplete** yet.

d) Write the unit tests that will verify the TqsStack contract.

   You may use the IDE features to generate the testing class; note that the IDE support will vary. Be sure to use JUnit 5.x.

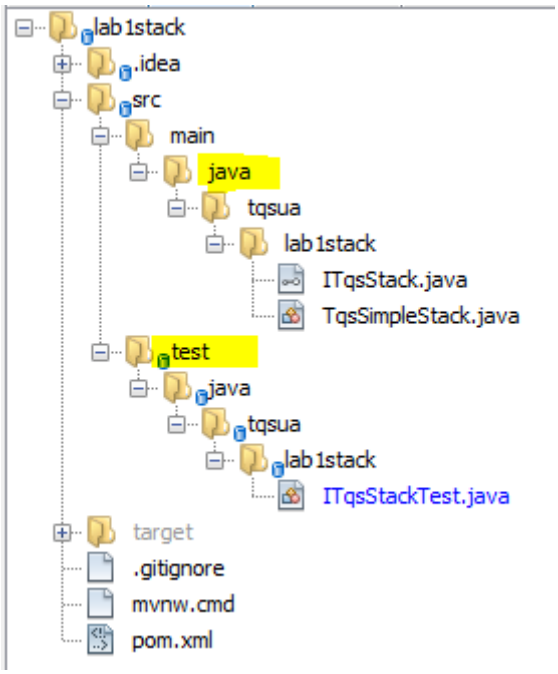
   Your tests will verify several assertions that should evaluate to true for the test to pass. See some examples.

e) Run the tests and prove that TqsStack implementation is not valid yet (the tests should fail for now, the first step in Red-Green-Refactor).

f) Correct/add the missing implementation to the TqsStack;

g) Run the unit tests.

h) Iterate from steps d) to f) and confirm that all tests pass.



Suggested stack contract:

- push(x): add an item on the top
- pop: remove the item at the top
- peek: return the item at the top (without removing it)
- size: return the number of items in the stack
- isEmpty: return whether the stack has no items

What to test[3]:

a) A stack is empty on construction.

b) A stack has size 0 on construction.

c) After n pushes to an empty stack, n > 0, the stack is not empty and its size is n

d) If one pushes x then pops, the value popped is x.

---

[1] If using IntelliJ: you may skip this step and ask, later, the IDE to fix JUnit imports.
[2] Delete the "pom-JITPACK.xml" and "pom-SNAPSHOT.xml", specially before importing into an IDE.
[3] Adapted from http://cs.lmu.edu/~ray/notes/stacks/

e) If one pushes x then peeks, the value returned is x, but the size stays the same

f) If the size is n, then after n pops, the stack is empty and has a size 0

g) Popping from an empty stack does throw a NoSuchElementException [You should test for the Exception occurrence]

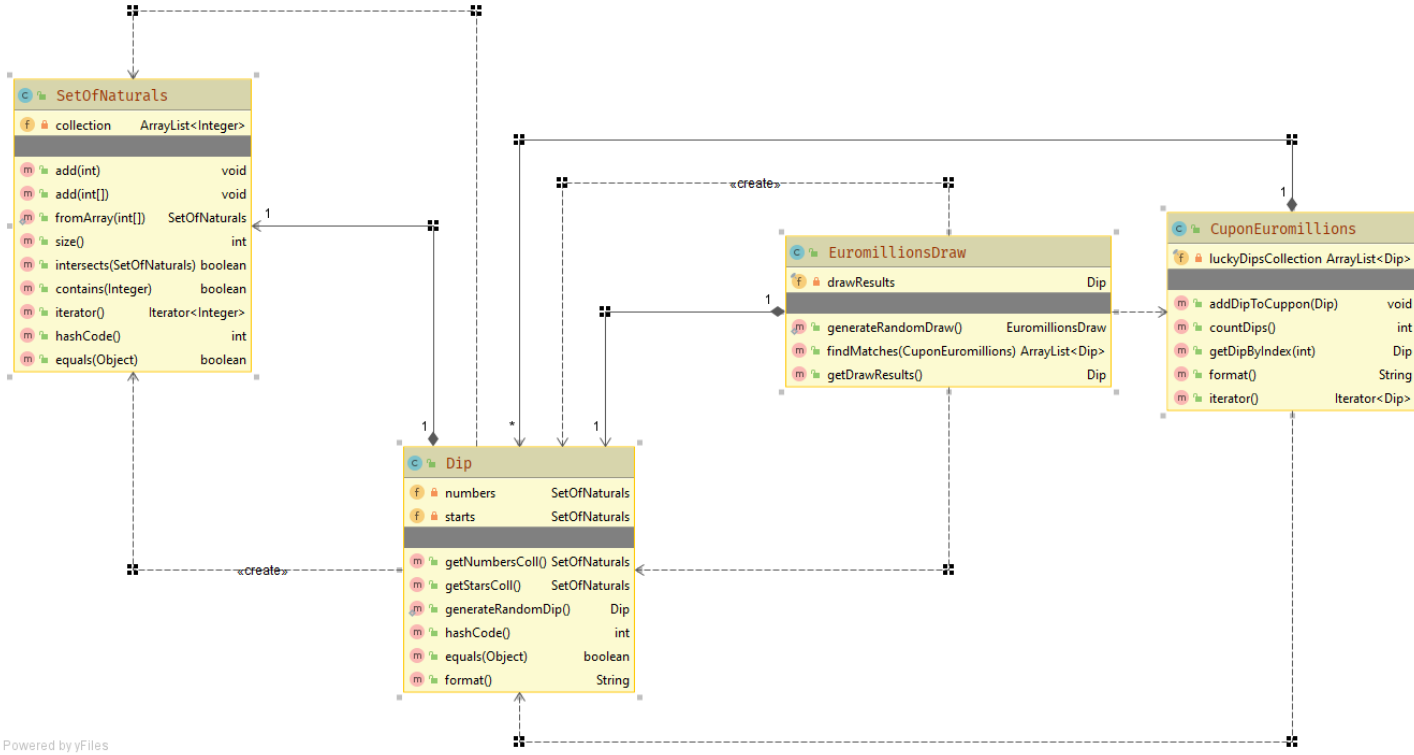
h) Peeking into an empty stack does throw a NoSuchElementException

i) For bounded stacks only, pushing onto a full stack does throw an IllegalStateException

2a/ Pull the "euromillions-play" project and correct the code (or the tests themselves, if needed) to have the existing unit tests passing.

For the (failing) test:	**You should:**
testFormat	Correct the implementation of Dip#format so the tests pass.
testConstructorFromBadArrays	Implement new test logic to confirm that an exception will be raised if the arrays have invalid numbers (wrong count of numbers of starts)

Note: you may suspend temporary a test with the @Disabled tag (useful while debugging the tests themselves).



2b/ The class SetOfNaturals represents a set (no duplicates should be allowed) of integers, in the range [1, +∞]. Some basic operations are available (add element, find the intersection…).
What kind of unit test are worth writing for the entity SetOfNaturals?
Complete the project, adding the new tests you identified.

2c/ Note that the code provided includes "magic numbers" (2 for the number of stars, 50 for the max range,…). Refactor the code to extract constants and eliminate the "magic numbers".

2d/ Assess the coverage level in project "Euromillions-play".
Configure the maven project to run Jacoco analysis.
Run the maven "test" goal and then "jacoco:report" goal. You should get an HTML report under target/jacoco.
Interpret the results accordingly. Which classes/methods offer less coverage? Are all possible decision branches being covered?

Note: IntelliJ has an integrated option to run the tests with the coverage checks (without setting the Jacoco plugin in POM). But if you do it at Maven level, you can use this feature in multiple tools.

Troubleshooting some frequent errors

➔ "Test are run from the IDE but not from command line."
Be sure to configure the surefire plug-in in Maven (example).

Explore

- JetBrains Blog on Writing JUnit 5 tests (with vídeo).
- JUnit 5 cheat sheet.
- Book: JUnit in Action. Note that you can access it from the OReilly on-line library, using your University of Aveiro's user account.
- Vogel's tutorial on JUnit. Useful to compare between JUnit 4 and JUnit 5.
- Working effectively with unit testing (podcast).

Lab 2: Mocking dependencies (for unit testing)

Learning objectives

— Prepare a project to run unit tests (JUnit 5) and mocks (Mockito 3.x), with mocks injection (@Mock).
— Write and execute unit tests with mocked dependencies.
— Experiment with mock behaviors: strict/lenient verifications, advanced verifications, etc.

Lab activities

Get familiar with sections 1 to 3 in the Mockito (Javadoc) documentation.

1a/
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 *Stocksmarket*. **StockPortfolio#getTotalValue()** method calculates the value of the portfolio by summing the current value (looked up in the stock market) of the owned stocks.

Implement (at least) one test to verify the implementation of **StockPortfolio#getTotalValue().**
Given that test should have predictable results, you need to address the problem of having non-deterministic answers from the Stockmarket interface.

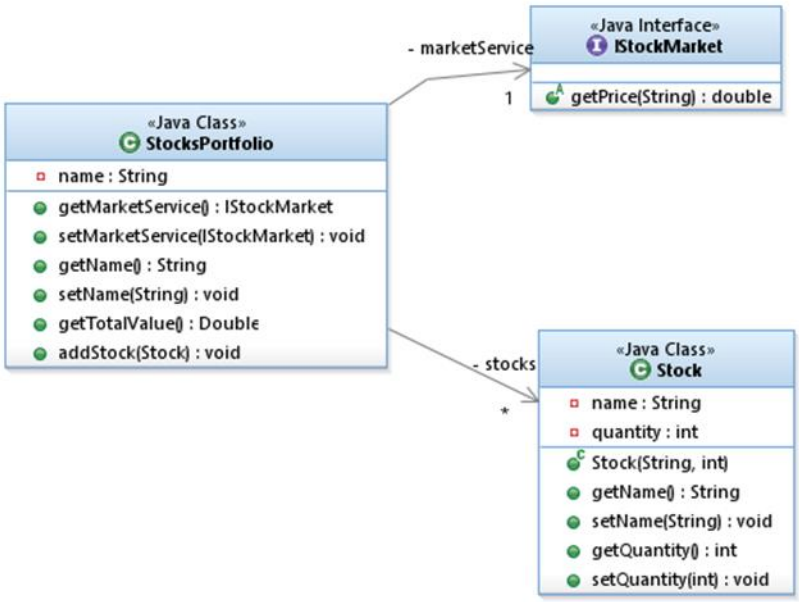


Figure 1: Classes for the StocksPortfolio use case.

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

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

```
1. Prepare a mock to substitute the remote 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.

```
@ExtendWith(MockitoExtension.class)
class StocksPortfolioTest { … }
```

- See a quick reference of Mockito syntax and operations.

1b/ Instead of the JUnit core asserts, you may use the Hamcrest library to create more human-readable assertions. Replace the "Assert" statements in the previous example, to use Hamcrest constructs. E.g.:

```
assertThat(result, is(14.0));
```

2/ Consider an application that needs to perform reverse geocoding to find a zip code for a given set of GPS coordinates. This service can found in public API (e.g.: using the MapQuest API).

a) Create the objects represented in Figure 1. TqsHttpClient represents a service to initiate HTTP requests to remote servers. At this point, **you do not need to implement TqsHttpBasic**; in fact, you should provide a substitute for it.

b) Consider that we want to verify the AddressResolver#findAddressForLocation, which invokes a remote geocoding service, available in a REST interface, passing the site coordinates.

Which is the service to *mock*?

c) To create a test for findAddressForLocation, you will need to know **the exact response of the geocoding service for a sample request**. Assume that we will use the MapQuest API. Use the browser or an HTTP client to try some samples so you know what to test for (example 1).

d) Implement a test for AddressResolver#findAddressForLocation using a mock.

e) Besides de "success" case, consider also testing for alternatives (e.g.: null URL should raise an exception; bad coordinates should not be accepted). This may affect the TqsHttpClient or other classes.

This getting started project [gs-mockForHttpClient] can be used in your implementation.

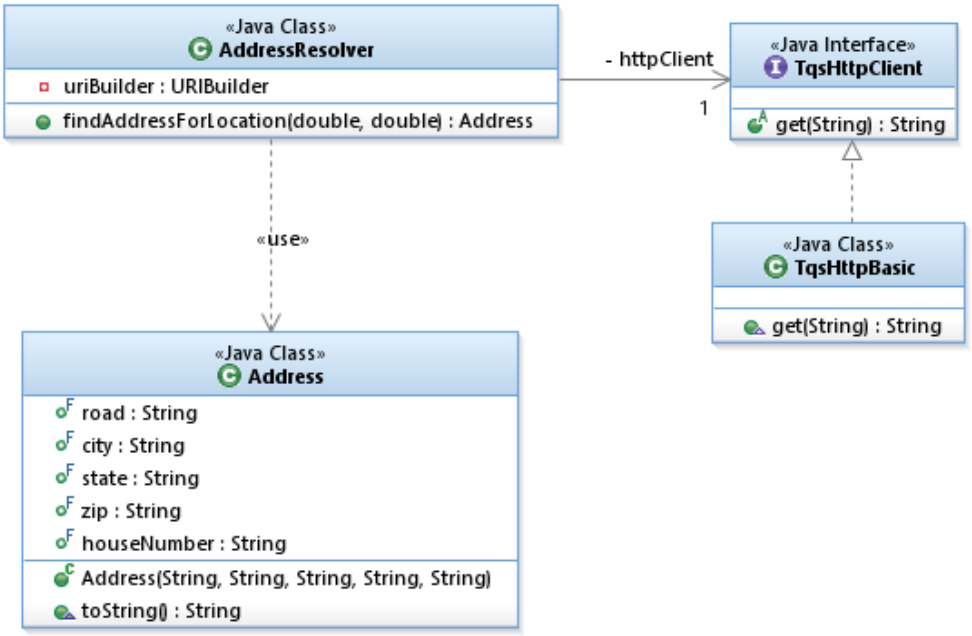


Figure 2: Classes for the geocoding use case.

3/ Consider you are implementing an integration test, and, in this case, you would use the real implementation of the module, not the mocks, in the test.
(This section can be included with the previous, continuing the same project.)

Create new test class (or reuse the existing AddressResolver**IT**), in a separate package, and be sure its name end with "**IT**".
Copy the tests from the previous exercise into this new test class, but remove any support for mocking (no Mockito imports in this test).
Correct the test implementation so it uses the real HttpClient implementation.
Run your test (and confirm that the remote API is invoked in the test execution).

If the "failsafe" maven plugin is configured, you should get different results with:
$ mvn test
$ mvn install failsafe:integration-test

Explore

- There is a recent book on JUnit and Mockito available from OReilly. The lessons are available as short videos too.