deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS Lab activities

v2024-02-21

TQS Lab activities **1**
Introductory notes and setup **1**
Lab 1 Unit testing (with JUnit 5) **1**
1.1 Simple Stack contract 2
1.2 EuroMillions 3
Troubleshooting some frequent errors 4

Lab 2 Mocking dependencies (with Mockito) **5**
2.1 Stocks portfolio 5
2.2 Geocoding 6
2.3 Integration tests (with the failsafe plugin) 7

Introductory notes and setup

Work submission

You should create a personal (git) repository for your TQS **individual portfolio** into which you will be pushing your solutions for the labs, named ***TQS_1235678*** (the number being your student number). Keep a **clean organization** that maps the exercises structure, e.g.: 🗁lab1/lab1_1; 🗁lab1/lab1_2; 🗁lab2/lab2_1; ...

You are expected to keep your repo (portfolio) up to date and complete. Teachers will select a few exercises later for assessment [not all, but representative samples].

Lab activities

Be sure that your developer environment meets the following requirements:

- Java development environment (Long-term support JDK recommended; v17 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. Check with:

  ```
  $ mvn --version
  ```

- Java capable IDE, such as IntelliJ IDEA (version "Ultimate" suggested) or VS Code.

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 5 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 is a small-scoped, coherent subset of a much larger solution. A true "unit" should not depend on the behavior of other (collaborating) modules.

- 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 in the code.
- JUnit and TestNG are popular frameworks for unit testing in Java.

Useful resources

- Book: JUnit in Action. Note that you can access it from the OReilly on-line library.
- Book: "Mastering Software Testing with JUnit 5" and associated GitHub repository with examples
- JetBrains Blog on Writing JUnit 5 tests (with vídeo).

1.1 Simple Stack contract

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

a) Create a new project (**maven project** for a Java standard application).

Add the required dependencies to run JUnit 5 tests.

- Note the elements: **junit-jupiter** and **maven-surefire-plugin** in the side illustration.
- You may adapt from a starter project for Maven.

```
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.2</version> <!-- can be omitted when using the BOM -->
        <scope>test</scope>
    </dependency>
    <!-- ... -->
</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.1.2</version>
        </plugin>
        <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>3.1.2</version>
        </plugin>
    </plugins>
</build>
```

b) Create the required class definition (**just the "skeleton"**, do not implement the methods body yet!). The **code should compile** (you may need to add dummy return values).

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

See also: basic examples.

d) 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.

[Mixing JUnit 4 and JUnit 5 dependencies will prevent the test methods to run as expected!]

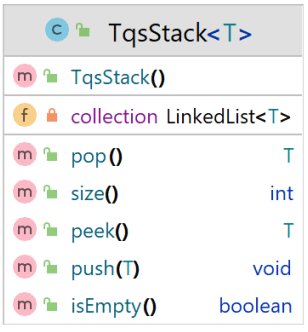
e) Your tests will verify several assertions that should evaluate to true for the test to pass.

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

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

h) Run the unit tests.

i) Iterate from steps f) to h) and confirm that all tests are passing.



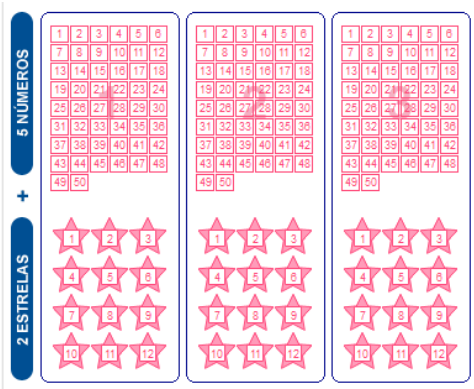
Suggested stack contract:

- 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[1]:

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.
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

1.2 EuroMillions

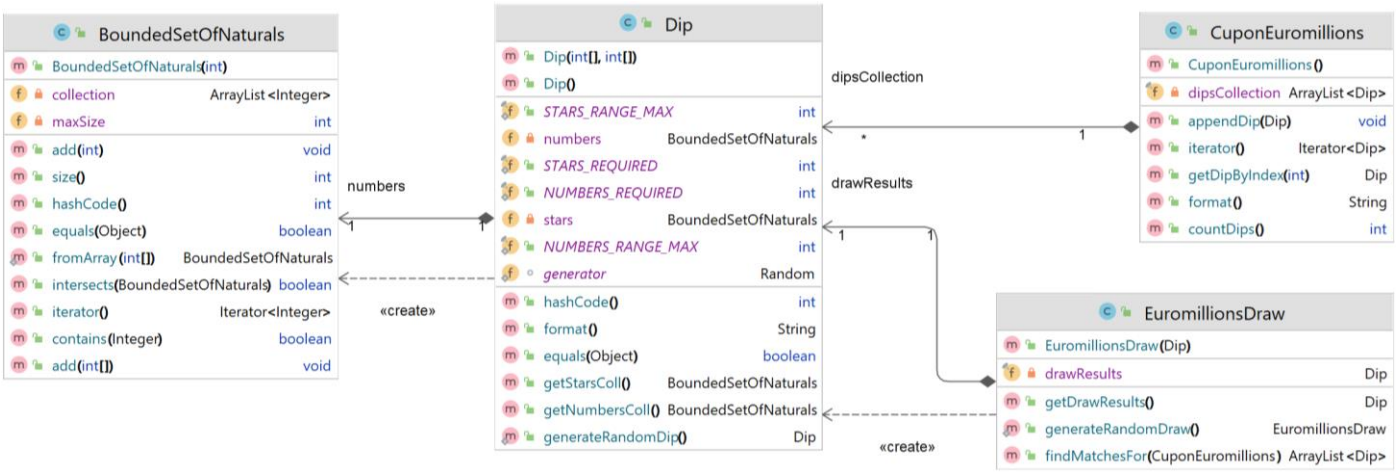


Let us consider the "Euromilhões" use case.

2a/ Pull the "euromillions-play" project
The supporting implementations is visualized in the class diagram that follows.
Get familiar with the solution and existing tests.

Class	Purpose
BoundedSetOfNaturals	Reusable set data structure no duplicates allowed (it is a Set) only natural numbers in the range [1, +∞]. the max size of the set (count of elements) is bounded to a limit allows set operations (contains element?, append element, calculate intersection with another set,...)
Dip	A collection of 5 "numbers" and 2 "stars" (a "column" in the Euromillions playing coupon)
CouponEuromillion	One or more Dips, representing a bet from a player.
EuromillionsDraw	Holds the winning dip and can find matched for a given player coupon.

[1] Adapted from http://cs.lmu.edu/~ray/notes/stacks/



2b/ Make the necessary changed for the existing (non-disabled) unit tests pass.

For the (failing) test:	**You should:**
testConstructorFromBadRanges	Change Dip implementation. Be sure to raise the expected exception if the arrays have invalid numbers (out of range numbers)

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

2c/ 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*.

```
$ mvn clean test jacoco:report
```

Analyze the results accordingly. Which classes/methods offer less coverage? Are all possible [decision] branches being covered?
Collect evidence of the coverage for "BoundedSetOfNaturals".

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.

2c/
Consider the class BoundedSetOfNaturals and its expected contract.
What kind of unit test are worth writing for proper validation of BoundedSetOfNaturals?
Complete the project, adding the tests you have identified. (You may also enhance the implementation of BoundedSetOfNaturals, if necessary.)

2d/
Run Jacoco coverage analysis and compare with previous results. In particular, compare the "before" and "after" for the BoundedSetOfNaturals class.

Troubleshooting some frequent errors

Problem/Symptom	**Solution**
Tests run from the IDE but not from command line	Be sure to configure the Surefire plug-in in Maven to a recent version (example).

My project's pom.xml is a mess! It is too long and uses old artifacts...	Check this POM and adapt (loggers are optional). Delete everything else.

Lab 2 Mocking dependencies (with Mockito)

Learning objectives

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

Preparation

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

Useful resources

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

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 stock, looked up in the stock market service).

1a/

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.

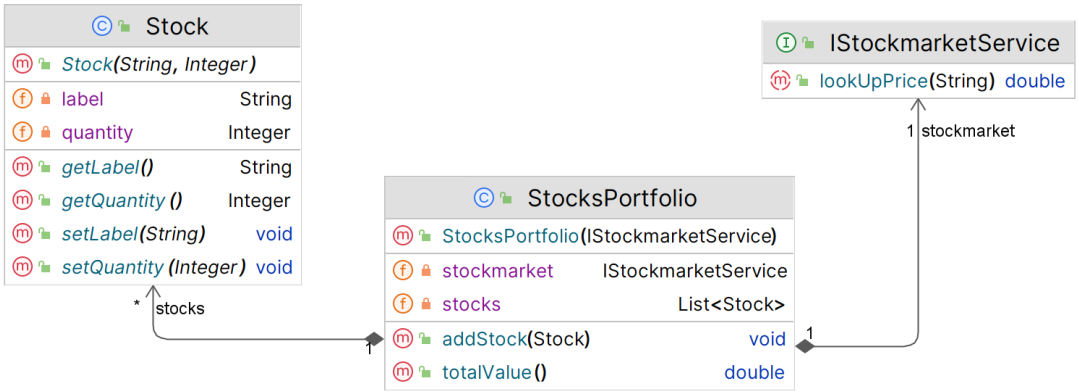


Figure 1: Classes for the StocksPortfolio use case.

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.

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

- See an example of the main syntax and operations.

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

2.2 Geocoding

Consider an application that needs to perform reverse geocoding to **find a zip code for a given set of GPS coordinates**. This service can be assisted by public APIs (e.g.: using the MapQuest API[2]).
Let us create a simple application to perform (reverse) geocoding and set a few tests.

a) Create the objects represented in Figure 1. At this point, **do not implement TqsBasicHttpClient**; in fact, you should provide a substitute for it.

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

   Which is the SuT (subject under test)? Which is the service to *mock*?

c) To create a test for findAddressForLocation, you will need to mimic **the exact (JSON) response of the geocoding service for a request**. Study/try the MapQuest API. See sample of response.

d) Implement a test for AddressResolverService#findAddressForLocation (using mocks where required).

e) Besides the "success" case, consider also testing for alternatives (e.g.: bad coordinates;...).

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

Figure 2: Classes for the geocoding use case.

[2] To use the MapQuest API you need a valid developer API key.

2.3 Integration tests (with the failsafe plugin)

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, not the mocks.
(This section can be included in the same project as the previous one.)

Create a test class (AddressResolver**IT**), in a separate test package (be sure its name ends with "**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).
Correct/complete 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).

Be sure the "failsafe" maven plugin is configured.
You should **get different results** with the following cases (try with and without internet connection):

```
$ mvn test
```

and

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

(Note the number of tests and the time required to run the tests...).