1 Lab: Unit tests & coverage analysis

v2025-09-25

1.0 Introduction

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.
- The JUnit is the “gold standard” framework for unit testing in Java. Build tools (Maven, Gradle) and IDE will recognize and support test execution.

Useful resources

- Short guide to Maven and “cheat sheet”
- 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 video).

1.1 Using tests in Maven project

In this exercise, you will work with a sample project that implements a simple “Calculator” module.

a) Make sure you have set your development environment according to the guidelines (from Lab 00).

b) Get the project “gs-calculator” [get as .zip] and inspect the code structure.

Locate the “production” and the test code.

Note: whenever you are using external code, from a Git repository, **do not clone the remote repo into your TQS working folder** (you would end with a .git inside your .git) Use a different support location and then use/copy just what you need.

c) Run all the tests. If using the IDE, get familiar with the IDE feedback on test results. Make sure you can also run tests from terminal:

```
$ mvn test
```

Note that you have executed the production code, without running the project. (In fact, there is no “main”…)

d) Take a moment to explore other Maven commands:

```
$ mvn clean
$ mvn dependency:tree
$ mvn clean test package
```

e) Does the current set of tests already provide a comprehensive “safety net”?

How does the calculator handle infinite decimal quotients (e.g. 13÷11)?

f) Complete the test cases to cover additional situations you may find useful. Include new operations (at least, “exp” and “sqrt”).

Remember: you should prepare a “**readme.md**” with your class notes, as you work in the exercise, to build your TQS’ portfolio. Collect important concepts, explain your approaches, include evidence of the results (especially if the lab section does not require a code project),…

1.2 Simple Stack unit

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 ahead. Don’t use AI to generate the first tests (we will use AI later…)

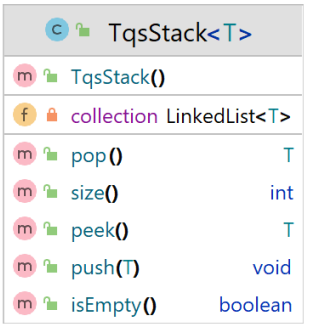
a) Create a new project (**Maven project**; no archetype needed). Add the required JUnit 5 dependencies to the POM, as in the previous exercise, or adapt from here.

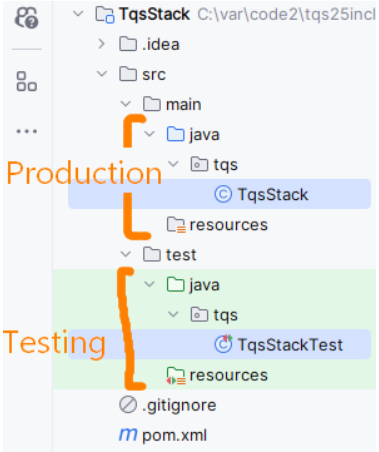
Note the elements: **junit-jupiter** (version 5.x) and plugin **maven-surefire-plugin** (>2.22 required)

b) Create the required class definition (**just the “skeleton”**, do not implement nor generate the methods body yet!).

The **code should compile** (you may need to add dummy return values to silence the compiler). 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





c) Write unit tests that will verify the TqsStack contract, i.e., which set of conditions define the functional correctness of a stack module? Consider testing the following cases[1]:

- A stack is empty on construction
- A stack has size 0 on construction
- After n > 0 pushes to an empty stack, the stack is not empty and its size is n
- If one pushes x then pops, the value popped is x.
- If one pushes x then peeks, the value returned is x, but the size stays the same
- If the size is n, then after n pops, the stack is empty and has a size 0
- Popping from an empty stack throws a NoSuchElementException
- Peeking into an empty stack throws a NoSuchElementException

Keep in mind that maven expects test code to be placed in a specific folder. You may use the IDE features to generate the initial (empty) 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 from running as expected!

Test code should include several assertions that will evaluate to true for the test to pass.

d) 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).

e) Correct/add the missing implementation in the TqsStack.

Run the unit tests and confirm that all tests are passing.

f) Your IDE should be able to provide information on coverage.

Look for it and note it down. Experiment disabling some tests (with @Disabled annotation) and verify the impact on coverage metrics.

g) Rename the testing class you have implemented so it's not executed (e.g., rename it from *xxxTest*.java to *_xxxTest*.java__). Generate a new suite of unit tests, but now using AI (such as Github Copilot,…) and compare the tests with your own previous implementation. Run the tests, check the coverage information, and compare it with the previous one.

h) Add a method **popTopN(int n)** to the TqsStack that returns the n^{th} item, discarding the n-1 top elements, inspired in the implementation snippet provided. Also add one test to cover it, so that coverage level stays the same (aim at 100% method and statement coverage).

```
public T popTopN(int n) {  1 usage   new *
    T top = null;
    for (int i = 0; i < n; i++) {
        top = collection.removeFirst();
    }
    return top;
}
```

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

i) Considering the previous point, can you think of a scenario where the TqsStack will fail despite the high coverage level?

To which extent can one rely on code coverage to assess quality of your code?

1.3 Meals booking service

Let us assume that the University's canteen is implementing a system that requires students to book meals in advance. When the student books a meal, a reservation code (short "token") is provided that can be used later to check details for an active reservation or optionally cancel it. Food-services workers should be able to verify a reservation and mark the code as used when students check-in to eat.

Focus on the business rules for the Meals Booking service. Take note of relevant situations and edge cases to implement a robust booking service, e.g.:

- Only one booking per student allowed, per service shift (no double booking).
- Tickets that have been used (to check-in) cannot be used again.
- Theres a maximum service capacity per shift (lunch, dinner,…).
- Canceled tickets can't be used, etc.

Use this context and:

a) Take note of the business rules that you think may be useful. Go beyond the previous examples…

b) Implement a simple MealsBookingService to code the required functionality (Figure 1). Feel free to use support from productivity/AI tools.

Keep the solution simple, otherwise it would quickly need other complex classes and frameworks (e.g.: storage, web layer,…) and no longer be a case of true "unit" thinking.

c) Prove the code works, by supplying a **comprehensive set of tests**.

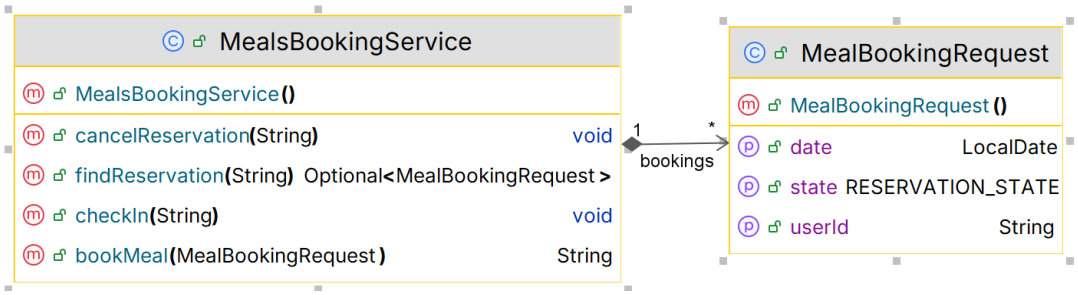


Figure 1: possible skeleton for the Meals booking service.

d) **Analyze the coverage** level in project using Jacoco

Configure the maven project to run Jacoco analysis. Use the up to date version of the plugin.

Run the maven "test" goal and then "*jacoco:report*" goal. You should get an HTML report under <project root>/*target/jacoco*.

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

Analyze the results accordingly. Which classes/methods offer less coverage? Are all possible [decision] branches being covered?

Note: IntelliJ and VS Code have 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 and environments.

e) Add a rule in Jacoco (i.e., in your pom.xml file) to ensure a minimum of 90% line coverage for every class, excluding the test classes. This rule should fail the build if the coverage goal is not met.

   Run the maven "test" goal and then "jacoco:check" goal to make sure it fails (if you need to set a higher line coverage goal, please set accordingly; also, you may disable some tests for the purpose of making the rule fail the build, for now).

```
$ mvn clean test jacoco:check
```

f) Improve the existing coverage levels using the coverage information (e.g., from your IDE and from Jacoco), to meet the previous line coverage goal. You may wish to exclude data-centric classes (e.g.: MealBookingRequest), which have no relevant logic to test, from the coverage analysis.