

Software Testing Lab

(Summer School “Software Testing and Beyond”)

Assignment 1 - Set-up

Submission Deadline: **Monday, 30 June 2025, 20:00**

1 ASSIGNMENT

For this assignment, you will learn how to use Maven for automating the build process, Java assert statements for developing built-in tests and applying design-by-contract, JUnit for running Java unit tests, and Cobertura and JaCoCo for measuring the code coverage of the available tests.

1.1 MAVEN

Information regarding Maven is available at <https://maven.apache.org/>. Try multiple Maven goals and familiarise yourself with how Maven works. Use Maven to generate JavaDoc information for the project.

- **Exercise 1.** Describe the activities you performed. **(Required, 5 points)**
- **Exercise 2.** What is the information contained in the generated report (in `target/site` directory)? How can they be used to gain knowledge about the system? **(Required, 5 points)**

- **Exercise 3.** Lookup the warnings in the output after executing `mvn clean site` and fix them. Describe your modifications. **(Only when late, -5 to 0 points)**

1.2 JUNIT

To familiarise yourself with the way JUnit is used in JPacman, take a look at the various available test classes. The Java source code is in the directory `src/main`, while the test cases are in the directory `src/test`, following the directory structure as used by default in Maven projects. Observe that the package structure of these two directories is exactly the same, allowing test cases to access package visible members. If you are not familiar with JUnit, carefully study the article “Test Infected, Programmers Love Writing Tests”, by Gamma and Beck, available at <http://members.pingnet.ch/gamma/junit.htm>.

Next, add the method `adjacent` for the class `Cell` in JPacman with the following responsibility:

```
/**
 * Determine if the other cell is an immediate neighbour
 * of the current cell (up, down, left, or right).
 * @return true if the other cell is immediately adjacent.
 */
public boolean adjacent(Cell otherCell) {
    ...
}
```

- **Exercise 4.** Generate as many functional (also called responsibility-driven) test cases as you think are necessary. Describe each test case. **(Required, 8 points)**
- **Exercise 5.** Turn your test cases into JUnit test cases in `CellTest`, and include a stub `adjacent` method in `Cell` to make sure your code compiles. What happens if you run these tests? **(Required, 7 points)**
- **Exercise 6.** Write a proper implementation of `adjacent` and rerun your test cases. Describe your development process. **(Required, 5 points)**
- **Exercise 7.** In your opinion, what are the benefits and shortcomings of this method of development? **(Only when late, -5 to 0 points)**
- **Exercise 8.** Repeat the previous JUnit exercises using TestNG. **(Only when late, -5 to 0 points)**

1.3 ASSERTIONS

To familiarise yourself with programming with assertions carry out the following steps:

- **Exercise 9.** Analyse the various uses of assertions (also see the attached page of Binder, Testing Object-Oriented systems: p818). Search for assertions that are used as precondition, postcondition, and as class invariant within JPacman. List one example for each category (Pre-condition, post-condition and class invariant). **(Required, 5 points)**
- **Exercise 10.** Explain the differences between the JUnit collection of assert methods and the Java assert statement. **(Required, 5 points)**
- **Exercise 11.** To get a feeling of what happens when an assertion fails, include an assertion (with documentation string) that you know will fail on a point that you know will be executed by one of the tests. Run the tests and explain what happens. **(Required, 7 points)**
- **Exercise 12.** Now modify the Maven build file so that the tests are run with assertion checking disabled. Rerun, and see what happens. Describe the modifications you made, and describe what happens if you run the tests this way. Make your conclusions about asserts: when do you use the Java assert statement and when a testing framework? Finally, undo your changes to the build file, rerun to check that the assertions indeed fail, and remove the failing assertions. **(Required, 8 points)**

1.4 CODE COVERAGE

Information and documentation concerning the open source coverage tool JaCoCo is available at <http://www.eclemma.org/jacoco/>. JaCoCo is available as a plugin for Maven, or Eclipse.

- **Exercise 13.** Introduce the necessary modifications to `pom.xml` to run JaCoCo when executing `mvn site`. Describe the process. **(Required, 10 points)**
- **Exercise 14.** Navigate through the coverage results by clicking on packages or classes. List the three most interesting percentages you found, and explain them. Were there parts of the code that were not covered? **(Required, 10 points)**
- **Exercise 15.** What color are most of the assert statements? Why? How does this affect the percentages provided by JaCoCo? **(Required, 5 points)**
- **Exercise 16.** Are there differences between Line Coverage and Branch Coverage? Take an example and explain the reason. **(Only when late, -5 to 0 points)**
- **Exercise 17.** Repeat the previous code coverage exercises using Cobertura¹. **(Only when late, -5 to 0 points)**

¹<http://www.mojohaus.org/cobertura-maven-plugin/usage.html>

1.5 MOCKS

Read the article by Martin Fowler² to understand the concept of mocking. Familiarise yourself with a mock library for Java such as JMockit, Mockito, JMock, EasyMock, etc.

- **Exercise 18.** Reconsider some of the functional tests you wrote in Exercise 5. Write them now using mocks. Use a mock library of your choice. **(Required, 10 points)**
- **Exercise 19.** Is the coverage resulting from the mock-based test the same as the original? Compare both approaches. When would you prefer mock testing? **(Required, 10 points)**
- **Exercise 20.** Repeat the mocking exercises with a different mocking framework. **(Only when late, -5 to 0 points)**

²<https://martinfowler.com/articles/mocksArentStubs.html>

TABLE 17.1 Implementation-based Assertion Examples

Implementation Relationship	Example Assertion
Reasonable limits on resources	<code>assert(NumBuff < 256);</code>
Incomplete or untested code	<code>assert(FALSE, "foo isn't complete");</code>
Constraints and unenforced assumptions on message arguments	<code>// No more than 24 total hours in day assert ((taskHours + slackHours) < 25);</code>
Range checking on any scalar variable	<code>assert (dx <= maxDx && dx > 10000);</code>
Bounds checking on arrays and other collections	<code>assert (i <= sizeofArray); assert (x == array[i]);</code>
Membership in enumerated types or collections	<code>assert (!myObject.contains(x));</code>
Valid, non-null pointers	<code>assert (this != NULL); assert (anArray[i] != NULL);</code>
"This Should Never Happen" segment in an if, case, or switch expression	<code>assert (FALSE);</code>
Language-specific weaknesses	See Implementation
Redundant or cached values	<code>// Buffers in use don't exceed allocation assert (nBuffAlloc >= buffAllocCount)</code>
Server object invariants	<code>// foo is a server object. assert (foo.checkInvariant());</code>
Arithmetic relationships	<code>assert (x >= (a + b - c));</code>
Items in a collection	<code>assert ((next == last) (numItems > 0));</code>
Structures correct	<code>assert (p->fred != NULL); assert (p->refCount != 0);</code>
Constraints between arguments and instance/class variables	See Invariants
Constraints and unenforced assumptions on global variables	See Invariants
Constraints among instance/class variables	See Invariants
Constraints between arguments and returned values	See Postconditions
Constraints between instance/class variables and returned values	See Postconditions
Variables that must be the same (or different) at entry and exit	See Postconditions
Valid/invalid states	See State Invariants
Contract of a stub	All or part of the contract offered by the stubbed method
Contract of an abstract superclass method	The generic requirements for all subtypes