# Software Testing

Lab Assignments

March 03, 2025

**Niels Van der Planken**
s0191930@ad.ua.ac.be

**Thomas Gueutal**
s0195095@ad.ua.ac.be

## Contents

## 1 Introduction

This report is a deliverable for one of the testing assignments for the course Software Testing (Master Computer Science 2001WETSWT) at the University of Antwerp in the academic year of 2024-2025.

As per the course description, "*The objective of the lab work of the Software Testing course is to help you learn how you can apply the various testing techniques and test design patterns as discussed during the lectures in practice. You will apply these techniques to a simple Pacman system written in Java. The amount of coding that needs to be done is relatively small: The focus is on testing.*".

This report was written in **Typst**.

## 2 Division of Tasks

The deliverables are as follows.

1. Code solutions to the tasks, located at **this** Github repo.
2. This report, discussing the solutions and code.

We briefly discuss the division of tasks. We should strive for a half-half split of work between the two team members.

| Member | Tasks |
|---|---|
| Niels | Exercises 1 - 10 (including report) |
| Thomas | Exercises 11 - 20 (including report) |

## 3 Assignment 01 – Java Testing Tools

### 3.1 Part 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.

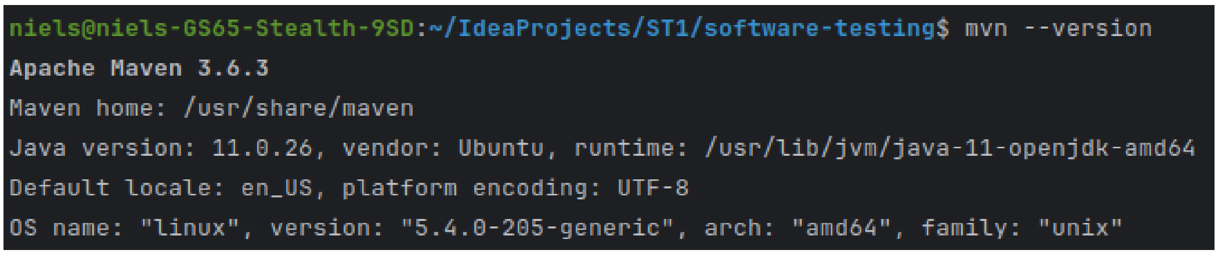
#### 3.1.1 Exercise 1

**Question:** Describe the activities you performed.

We made several changes to the *pom.xml* file.

- Change the *maven-compiler-plugin* source and target versions for the used java compiler from 1.5 to 1.8.

Before trying out Maven goals, we ensured that we have Maven installed and that it was running correctly by using the command `mvn --version`.



```
niels@niels-GS65-Stealth-9SD:~/IdeaProjects/ST1/software-testing$ mvn --version
Apache Maven 3.6.3
Maven home: /usr/share/maven
Java version: 11.0.26, vendor: Ubuntu, runtime: /usr/lib/jvm/java-11-openjdk-amd64
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "5.4.0-205-generic", arch: "amd64", family: "unix"
```

Since everything was working correctly, we then tried out some different Maven goals. We started with `mvn clean`. Activity:

- Deleted the `target/` directory where compiled files are stored
- Ensured a fresh build environment for subsequent steps

We then performed the `mvn compile`. Activity:

- Compiled Java source files from `src/main/java`
- Created `.class` files inside `target/classes`
- Verified that compilation errors (if any) were displayed in the console.

After the normal `compile`, we also did a `mvn test-compile`

- Compiled test source files located in `src/test/java`
- Ensured that unit tests were ready to run without executing them
- Verified that the compiled test classes were placed in `target/test-classes/`

We followed this by `mvn test`.

- Ran unit tests from `src/test/java` using JUnit.
- Displayed test results and failures in the console.
- Verified that errors were logged if any test failed.

Our final goal was `mvn site`.

- Generated project documentation in the `target/site/` directory
- Created HTML reports including project summary, dependencies, plugins and test reports.
- Opened `target/site/index.html` in a browser to view the generated site.

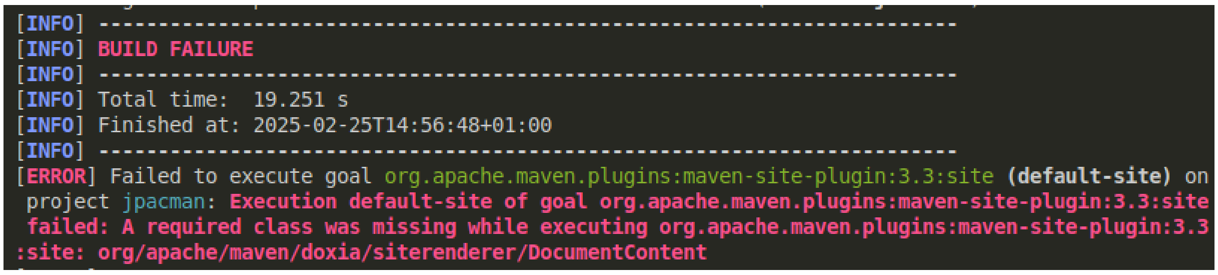
However, we encountered some problems with maven site. **Firstly**, `pom.xml` specified an old maven site plugin version, see the error specified in Figure 1. The specific problem is that the default maven site plugin version, when the `<version>` tag is not specified explicitly, is incompatible with the default version of `maven-project-info-reports-plugin` [1]. We found two viable solutions to resolve the incompatibility, both require editing `pom.xml`.

- We can explicitly specify a newer maven site plugin version than the default version.

```
<build>
  <plugins>
    ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-site-plugin</artifactId>
      <version>3.21.0</version>
    </plugin>
    ...
  </plugins>
</build>
```

- Or we can explicitly add an older version of `maven-project-info-reports-plugin` as a plugin.

```
<reporting>
  <plugins>
    ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
      <version>2.9</version>
    </plugin>
    ...
  </plugins>
</reporting>
```



```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  19.251 s
[INFO] Finished at: 2025-02-25T14:56:48+01:00
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-site-plugin:3.3:site (default-site) on
 project jpacman: Execution default-site of goal org.apache.maven.plugins:maven-site-plugin:3.3:site
 failed: A required class was missing while executing org.apache.maven.plugins:maven-site-plugin:3.3
:site: org/apache/maven/doxia/siterenderer/DocumentContent
```

Figure 1: Incompatible maven site plugin versions.

**Secondly**, sometimes maven cannot find the path to the `javadoc` executable. The `JAVA_HOME` environment variable may currently be empty. To resolve this (on linux), you can add a script to export the java home path for any shell session.

```
# Check manually that JAVA_HOME is empty.
echo $JAVA_HOME

# Append to a startup script to export JAVA_HOME for each shell session.
```

```
echo 'export JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:/bin/java::")' |
sudo tee -a /etc/profile.d/java_home.sh

# (optional) Change permissions.
chmod +x /etc/profile.d/java_home.sh

# Execute the script in the current shell to define JAVA_HOME ...
source /etc/profile.d/java_home.sh
# OR reboot your machine so the new script takes effect for all shell sessions.
reboot
```

If this does not fix the error, then we may also explicitly specify the path to the `javadoc` executable in the `pom.xml` file by adding the following lines to the `maven-javadoc-plugin` plugin in `<reporting>`.

```
<configuration>
  <javadocExecutable>${java.home}/bin/javadoc</javadocExecutable>
</configuration>
```

### 3.1.2 Exercise 2

**Question:** What is the information contained in the generated report (in target/site directory)? How can they be used to gain knowledge about the system?

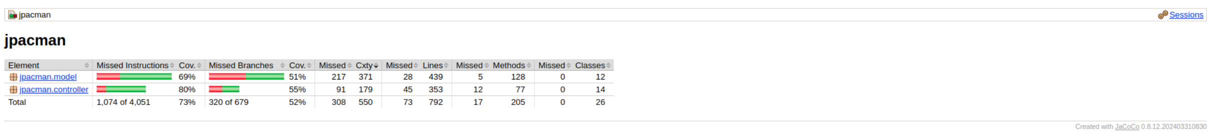
There are several reports available. See **Figure 3** for the main page of the generated site docs.

The Javadoc report is the usual API documentation that collates the package structure, class structure and any docstrings to allow browsing. This is useful to quickly understand the project structure and find high-level implementation details.

The Javadoc Test report gives similar information about the test suite. It gives a good overview of what features are actually tested for each class.

The Surefire report details the actual testing results, after the test suite has been run. It provides a clear and detailed overview of which parts of the testing suite are (in)complete and or have active problems, error, failures, etc.

More information will be given about the JaCoCo coverage reports in a later section. See **Figure 2** for a preview of the high-level coverage statistics.

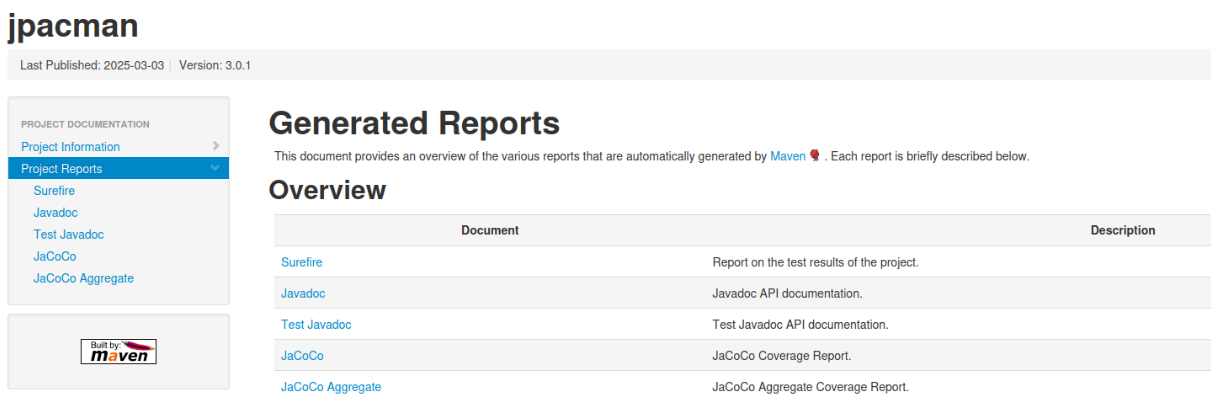


jpacman

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| jpacman.model | 🟥🟩 | 69% | 🟥🟩 | 51% | 217 | 371 | 28 | 439 | 5 | 128 | 0 | 12 |
| jpacman.controller | 🟥🟩 | 80% | 🟥🟩 | 55% | 91 | 179 | 45 | 353 | 12 | 77 | 0 | 14 |
| Total | 1,074 of 4,051 | 73% | 320 of 679 | 52% | 308 | 550 | 73 | 792 | 17 | 205 | 0 | 26 |

Created with JaCoCo 0.8.12.202403310830

Figure 2: The high-level coverage report statistics.



jpacman

Last Published: 2025-03-03 | Version: 3.0.1

PROJECT DOCUMENTATION
- Project Information
- Project Reports
  - Surefire
  - Javadoc
  - Test Javadoc
  - JaCoCo
  - JaCoCo Aggregate

Generated Reports

This document provides an overview of the various reports that are automatically generated by Maven . Each report is briefly described below.

Overview

| Document | Description |
|---|---|
| Surefire | Report on the test results of the project. |
| Javadoc | Javadoc API documentation. |
| Test Javadoc | Test Javadoc API documentation. |
| JaCoCo | JaCoCo Coverage Report. |
| JaCoCo Aggregate | JaCoCo Aggregate Coverage Report. |

Figure 3: The generated JPacman site documentation.

## 3.2 Part 2 – JUnit

### 3.2.1 Exercise 4

**Question:** Generate as many functional (also called responsibility-driven) test cases as you think are necessary. Describe each test case.

You can consider cells $x$ and $y$ to be adjacent if the manhattan distance between $x$ and $y$ equals 1. **You could determine domain-based in, on, off and out points** by partitioning cells based on (integer) manhattan distance from the reference cell. Note that all in points are also on points because the in partition is $\{1\}$.

Below you can find all the functional test cases we could think of.

First set of scenarios: cell is indeed adjacent.
These are **on points**. In these scenarios, the expected result of the function adjacent is `true`. Here are the possible scenarios that fall under this statement:
- Cell directly above: `Cell A` is at (1, 1) and `Cell B` is at (1, 2)
- Cell directly below: `Cell A` is at (1, 2) and `Cell B` is at (1, 1)
- Cell directly left: `Cell A` is at (2, 1) and `Cell B` is at (1, 1)
- Cell directly right: `Cell A` is at (1, 1) and `Cell B` is at (2, 1)

Second set of scenarios: cell is diagonally adjacent
These are **off points**. In these scenarios, the cell under question is diagonally adjacent to the base cell. The expected result of the function adjacent should be `False`. Here are the possible scenarios that fall under this statement
- Cell diagonally adjacent (top-left): `Cell A` is at (2, 2) and `Cell B` is at (1, 1)
- Cell diagonally adjacent (top-right): `Cell A` is at (2, 2) and `Cell B` is at (1, 3)
- Cell diagonally adjacent (bottom-left): `Cell A` is at (2, 2) and `Cell B` is at (3, 1)
- Cell diagonally adjacent (bottom-right): `Cell A` is at (2, 2) and `Cell B` is at (3, 3)

Third set of scenarios: cell is far away
These are **out points**. In these scenarios, the expected result of the function adjacent is `False`.
- Same row, different columns: `Cell A` is at (1, 1) and `Cell B` is at (1, 5)
- Same column, different rows: `Cell A` is at (1, 1) and `Cell B` is at (4, 1)
- Different rows and columns: `Cell A` is at (1, 1) and `Cell B` is at (4, 4)

Same Cell
When given the same cell twice for the adjacency check, the function should return `False`. This is an **off point**.

Negative Coordinates
This is an edge case for possible incorrect input handling. It is not mentioned if negative coordinates are possible or not, but given the fact that we are working with PacMan, a negative coordinate (outside the grid) should be handled with an error message. These are **out points**.

Large Grid scenarios
Does the method work efficiently for large numbers?

- Adjacent in a large grid, expected result `True`. An example is: `Cell A` is at (1000, 1000) and `Cell B` is at (1001, 1000). Here `cell B` is an **on point**.
- Far apart in a large grid, expected result `False`. An example is: `Cell A` is at (1000, 1000) and `Cell B` is at (2, 1000). Here `cell B` is an **out point**.

Uninitialized or Null cells
What happens if `Cell A` is initialized and `Cell B` is not? This should be handled with an `Exception`.

### 3.2.2 Exercise 5

**Question:** 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?

We implemented several test cases from the above scenarios. Since the method `adjacent()` is not yet implemented, the tests fail of course.

### 3.2.3 Exercise 6

**Question:** Write a proper implementation of adjacent and rerun your test cases. Describe your development process.

The adjacent method was not really that hard to write. We need 2 cells to compare: the current `this` cell and the cell that's given as input. Then we take the coordinates of them, take them as absolute values and subtract them. If the result is `(1, 0)` or `(0, 1)`, the cells are adjacent. Note that before returning we must assert the Cell invariant, the method `Cell.invariant()`, in order to satisfy the Cell class interface.

After implementing the method, we reran the tests and passed every one of them.

## 3.3 Part 3 – Assertions

### 3.3.1 Exercise 9

**Question:** 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).

In the file `src/main/java/jpacman/controller/ImageFactory.java`, the method `public int monsterAnimationCount()` specifies two assertions.

```java
public int monsterAnimationCount() {
    assert monsterImage != null;
    int result = monsterImage.length;
    assert result >= 0;
    return result;
}
```

The first assertion is a pre-condition which makes sure the function does not operate on a null image object. The second assertion is a post-condition assertion that ensures the result is within the valid output domain $\mathbb{N} = \{0, 1, 2, ...\}$.

pre-condition : `assert monsterImage != null;`

post-condition : `assert result >= 0;`

The JPacman project makes use of class invariant assertions in multiple places. For example, the Pacman constructor function in `src/main/java/jpacman/controller/Pacman.java` asserts an invariant function invocation. This is consistent with the definition of an invariant: a condition that always hold before and after public API methods, and immediately after the constructor.

class invariant : `assert invariant();`

```java
public Pacman(Engine e) throws IOException {
    this(e, new RandomMonsterMover(e));
    assert invariant();
}
```

### 3.3.2 Exercise 10

**Question:** Explain the differences between the JUnit collection of assert methods and the Java assert statement.

The largest differences are in the context of their use, and the breadth of the features they provide.

JUnit

- The JUnit assert methods are used for testing in a controlled environment, namely unit testing inside methods annotated with `@Test`, that live in purpose built test classes. Inside this context, the thrown exceptions are automatically handled by the JUnit framework, with no additional exception handling required of the test designer. The assertions are a source of information for the test suite runner, to generate statistics and reports, and to provide detailed error reporting. However, outside of this context the JUnit assert methods lose these qualities.
- The JUnit assert methods do not make use of the Java assert. Because JVM explicitly requires the `-ea` flags to enable Java assertions, this avoids misconfiguration errors and tests (not) passing when they should (not) do so. The diversity of the JUnit assert methods promotes test brevity by abstracting away for example comparison logic: `assertEquals`, `assertArrayEquals`, `assertNotNull`, etc. [2]

Java

- Java assert statements focus more on debugging source code. It can be used inside any Java class or method to propagate Exceptions that the caller must also explicitly handle. Java asserts are mostly used for enforcing assumptions (pre-conditions, post-conditions, ...) within the source code. Java asserts can still be called within the JUnit context, but they simply provide fewer reporting benefits to the testing suite.
- There is only the one Java assert statement, `assert`. This lack of convenience methods, such as JUnit's `assertArrayEquals`, makes expression evaluations somewhat less readable.

### 3.3.3 Exercise 11

**Question:** 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.

We add an assert that **always fails** to keep things simple. The function of choice is `Cell.isOccupied()` in `src/main/java/jpacman/model/Cell.java`. The only test that currently depends on it is `jpacman.model.PlayerMoveTest.testApply` in `src/test/java/jpacman/model/MoveTest.java`.

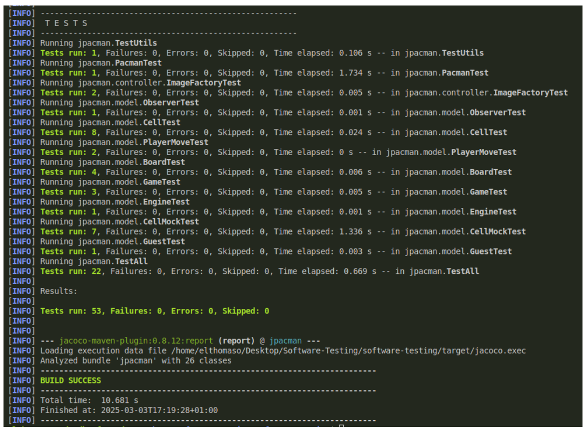
```java
public boolean isOccupied() {
    /** Always fail this function call: "Should not reach here." */
    assert false;

    return inhabitant != null;
}
```

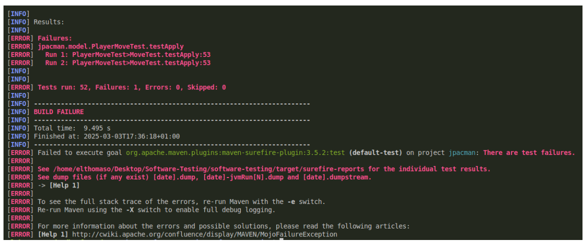
See Table 1 for the test results before adding the assert, and after adding the failing assert. The surefire test report shows that two invocations of the `testApply()` method raised exceptions: once when running `jpacman.model.PlayerMoveTest` and once when running `jpacman.TestAll`.

Both failures simply show that an `java.lang.AssertionError` cause the those particular test cases to fail. The exception was raised in the source code, but the test case itself neither caught it using a try-catch block, not did the tests specify `assertThrows` from JUnit. By default unexpected exceptions cause the test case that cause them to fail, but all other test cases remain unaffected.

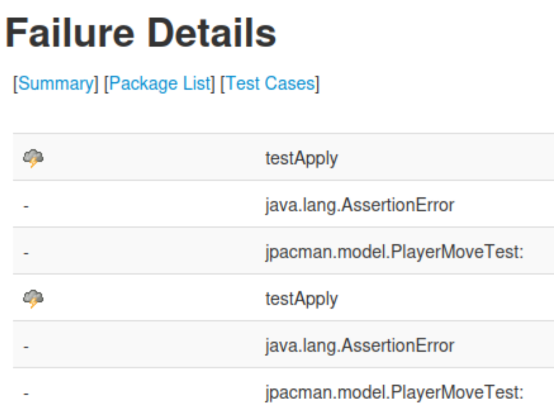
Note that we forgot to exclude the mock-based test cases in the screenshot below, which leads to 53 total test cases.



```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running jpacman.TestUtils
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.106 s -- in jpacman.TestUtils
[INFO] Running jpacman.PacmanTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.734 s -- in jpacman.PacmanTest
[INFO] Running jpacman.controller.ImageFactoryTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.005 s -- in jpacman.controller.ImageFactoryTest
[INFO] Running jpacman.model.ObserverTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s -- in jpacman.model.ObserverTest
[INFO] Running jpacman.model.CellTest
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.024 s -- in jpacman.model.CellTest
[INFO] Running jpacman.model.PlayerMoveTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s -- in jpacman.model.PlayerMoveTest
[INFO] Running jpacman.model.BoardTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 s -- in jpacman.model.BoardTest
[INFO] Running jpacman.model.GameTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.005 s -- in jpacman.model.GameTest
[INFO] Running jpacman.model.EngineTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s -- in jpacman.model.EngineTest
[INFO] Running jpacman.model.CellMockTest
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.336 s -- in jpacman.model.CellMockTest
[INFO] Running jpacman.model.GuestTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 s -- in jpacman.model.GuestTest
[INFO] Running jpacman.TestAll
[INFO] Tests run: 22, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.669 s -- in jpacman.TestAll
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 53, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.12:report (report) @ jpacman ---
[INFO] Loading execution data file /home/elthomaso/Desktop/Software-Testing/software-testing/target/jacoco.exec
[INFO] Analyzed bundle 'jpacman' with 26 classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  10.681 s
[INFO] Finished at: 2025-03-03T17:19:28+01:00
[INFO] ------------------------------------------------------------------------
```



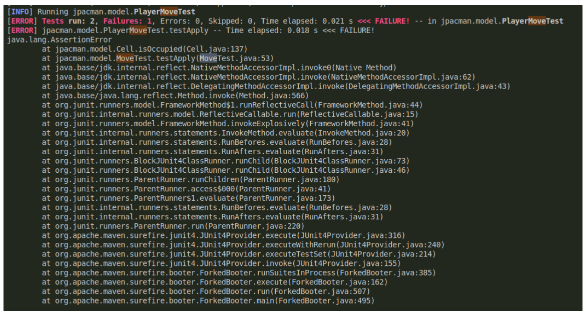
```
[INFO]
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR] jpacman.model.PlayerMoveTest.testApply
[ERROR]   Run 1: PlayerMoveTest>MoveTest.testApply:53
[ERROR]   Run 2: PlayerMoveTest>MoveTest.testApply:53
[INFO]
[INFO]
[ERROR] Tests run: 52, Failures: 1, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  9.495 s
[INFO] Finished at: 2025-03-03T17:36:18+01:00
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:3.5.2:test (default-test) on project jpacman: There are test failures.
[ERROR]
[ERROR] See /home/elthomaso/Desktop/Software-Testing/software-testing/target/surefire-reports for the individual test results.
[ERROR] See dump files (if any exist) [date].dump, [date]-jvmRun[N].dump and [date].dumpstream.
[ERROR] -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
```



**Failure Details**

[Summary] [Package List] [Test Cases]

| | |
|---|---|
| ⛈ | testApply |
| - | java.lang.AssertionError |
| - | jpacman.model.PlayerMoveTest: |
| ⛈ | testApply |
| - | java.lang.AssertionError |
| - | jpacman.model.PlayerMoveTest: |



```
[INFO] Running jpacman.model.PlayerMoveTest
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.021 s <<< FAILURE! -- in jpacman.model.PlayerMoveTest
[ERROR] jpacman.model.PlayerMoveTest.testApply -- Time elapsed: 0.018 s <<< FAILURE!
java.lang.AssertionError
        at jpacman.model.Cell.isOccupied(Cell.java:137)
        at jpacman.model.MoveTest.testApply(MoveTest.java:53)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:566)
        at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:44)
        at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:15)
        at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:41)
        at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:20)
        at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:28)
        at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:31)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:73)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:46)
        at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:180)
        at org.junit.runners.ParentRunner.access$000(ParentRunner.java:41)
        at org.junit.runners.ParentRunner$1.evaluate(ParentRunner.java:173)
        at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:28)
        at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:31)
        at org.junit.runners.ParentRunner.run(ParentRunner.java:220)
        at org.apache.maven.surefire.junit4.JUnit4Provider.execute(JUnit4Provider.java:316)
        at org.apache.maven.surefire.junit4.JUnit4Provider.executeWithRerun(JUnit4Provider.java:240)
        at org.apache.maven.surefire.junit4.JUnit4Provider.executeTestSet(JUnit4Provider.java:214)
        at org.apache.maven.surefire.junit4.JUnit4Provider.invoke(JUnit4Provider.java:155)
        at org.apache.maven.surefire.booter.ForkedBooter.runSuitesInProcess(ForkedBooter.java:385)
        at org.apache.maven.surefire.booter.ForkedBooter.execute(ForkedBooter.java:162)
        at org.apache.maven.surefire.booter.ForkedBooter.run(ForkedBooter.java:507)
        at org.apache.maven.surefire.booter.ForkedBooter.main(ForkedBooter.java:495)
```

Table 1: Add a failing assertion to the source code, and see the effects on the tests.

### 3.3.4 Exercise 12

**Question:** 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.

We can simply add a configuration option to the maven surefire plugin to enable/disable JVM assertions.

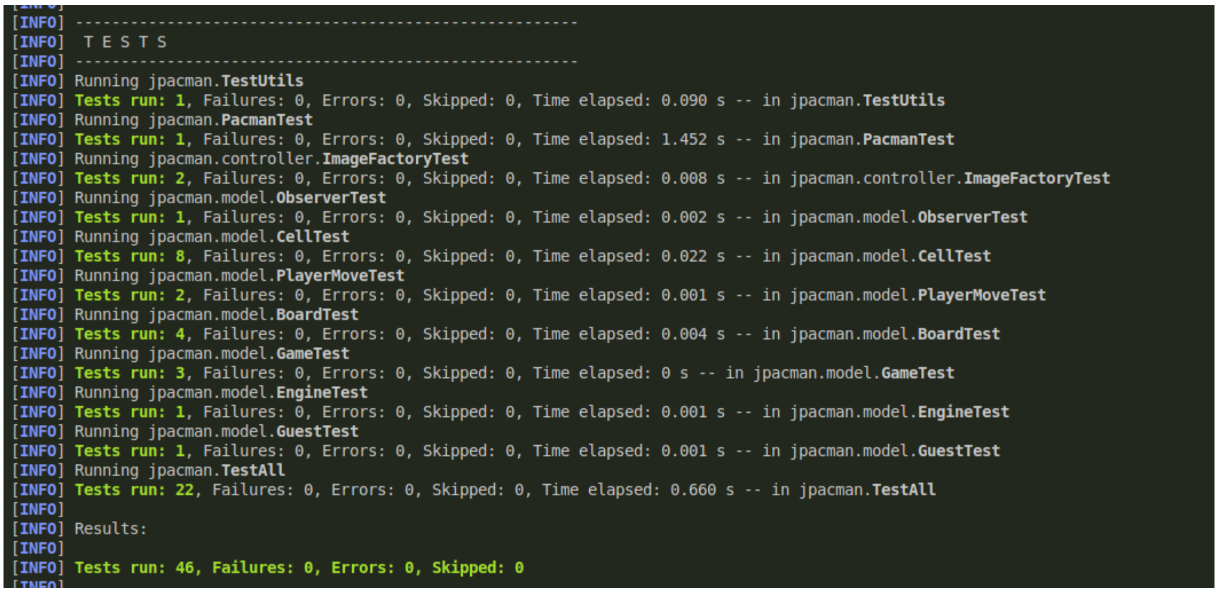
```
<build>
  <plugins>
    ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <!-- "true" to enable, "false" to disable assertions. -->
        <enableAssertions>false</enableAssertions>
      </configuration>
    </plugin>
    ...
  </plugins>
</build>
```

We then rerun the test suite with the above configuration, which disable Java assertions Figure 4. The seven mock-based tests were excluded in the screenshot, leading to $53 - 7 = 46$ total test cases. Since the Java assertions were disabled, the `assert false;` statement in the source code does not fail any tests.

Note that JUnit assertion methods are **not** affected by this configuration change. See Figure 5, where adding a `assertTrue(false);` statement to a test case still fails that case despite the changed configuration.

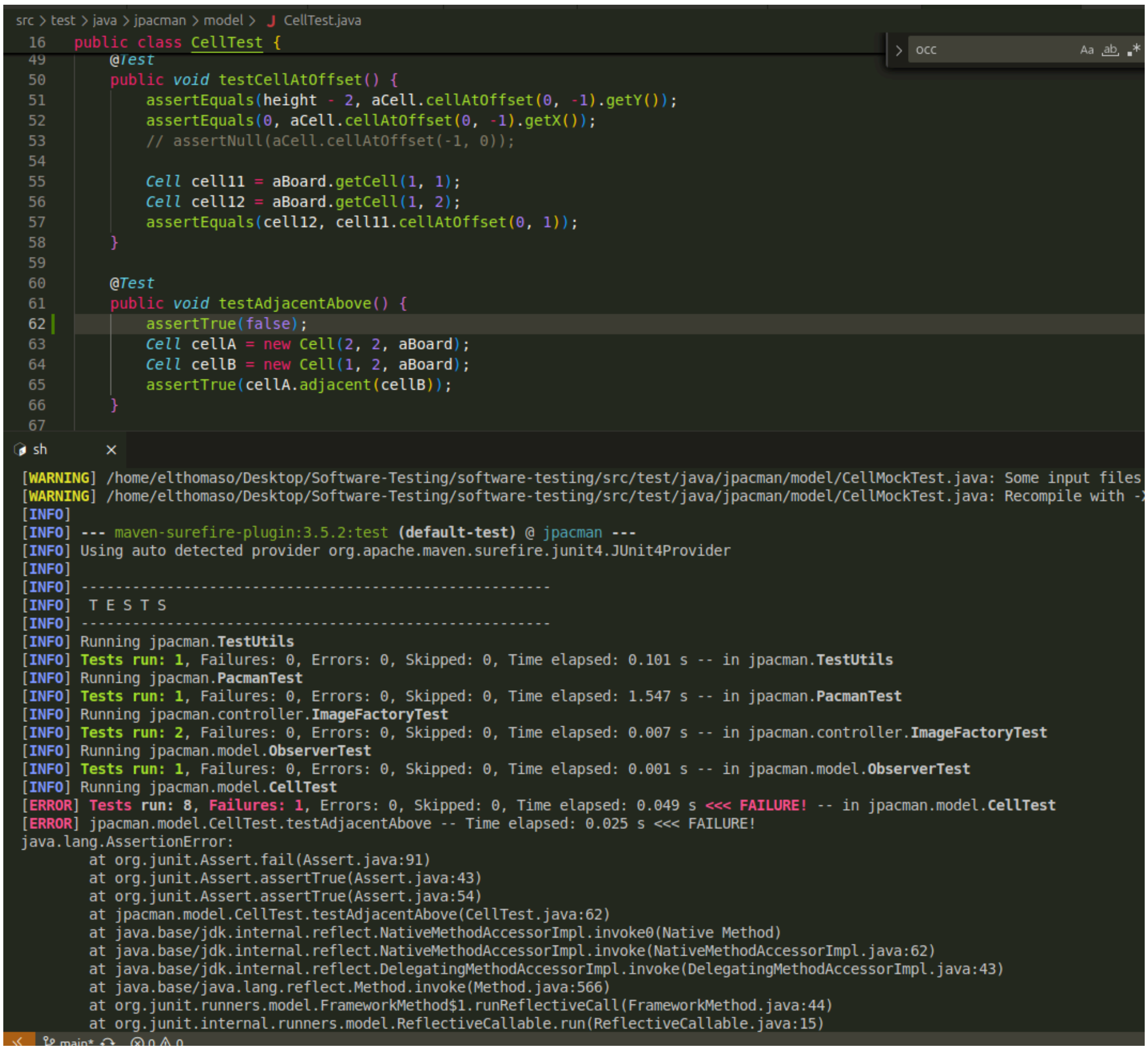
We conclude with the following observations.

- JUnit assertions: These assert methods should only be used within the scope of a testing environment. They circumvent Java's builtin assertion enabling/disabling mechanism, so their use outside the testing context may lead to unintended bugs. Their main purpose lies in verifying test state and results, as they gather statistics and provide detailed, readable test output. So the focus of test library assertions is providing human readable statistics to the user.
- Java assertions: The Java assert should be used (exclusively) in the source code. The main use is of course enforcing class and method contracts: pre-conditions, post-conditions and invariants. So the focus of Java assertions is signaling the occurrence of special circumstances or runtime problems during normal execution of the software.



```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running jpacman.TestUtils
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.090 s -- in jpacman.TestUtils
[INFO] Running jpacman.PacmanTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.452 s -- in jpacman.PacmanTest
[INFO] Running jpacman.controller.ImageFactoryTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.008 s -- in jpacman.controller.ImageFactoryTest
[INFO] Running jpacman.model.ObserverTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.002 s -- in jpacman.model.ObserverTest
[INFO] Running jpacman.model.CellTest
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.022 s -- in jpacman.model.CellTest
[INFO] Running jpacman.model.PlayerMoveTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s -- in jpacman.model.PlayerMoveTest
[INFO] Running jpacman.model.BoardTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s -- in jpacman.model.BoardTest
[INFO] Running jpacman.model.GameTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s -- in jpacman.model.GameTest
[INFO] Running jpacman.model.EngineTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s -- in jpacman.model.EngineTest
[INFO] Running jpacman.model.GuestTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s -- in jpacman.model.GuestTest
[INFO] Running jpacman.TestAll
[INFO] Tests run: 22, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.660 s -- in jpacman.TestAll
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 46, Failures: 0, Errors: 0, Skipped: 0
```

Figure 4: Disable Java assertions and rerun tests. Also exclude mock-based test cases.



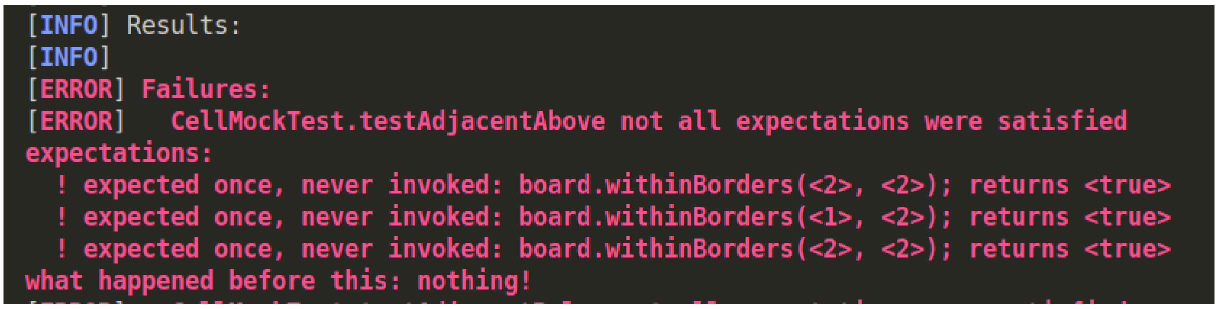
src > test > java > jpacman > model > CellTest.java

```
16  public class CellTest {
49      @Test
50      public void testCellAtOffset() {
51          assertEquals(height - 2, aCell.cellAtOffset(0, -1).getY());
52          assertEquals(0, aCell.cellAtOffset(0, -1).getX());
53          // assertNull(aCell.cellAtOffset(-1, 0));
54
55          Cell cell11 = aBoard.getCell(1, 1);
56          Cell cell12 = aBoard.getCell(1, 2);
57          assertEquals(cell12, cell11.cellAtOffset(0, 1));
58      }
59
60      @Test
61      public void testAdjacentAbove() {
62          assertTrue(false);
63          Cell cellA = new Cell(2, 2, aBoard);
64          Cell cellB = new Cell(1, 2, aBoard);
65          assertTrue(cellA.adjacent(cellB));
66      }
67
```

```
[WARNING] /home/elthomaso/Desktop/Software-Testing/software-testing/src/test/java/jpacman/model/CellMockTest.java: Some input files
[WARNING] /home/elthomaso/Desktop/Software-Testing/software-testing/src/test/java/jpacman/model/CellMockTest.java: Recompile with -
[INFO]
[INFO] --- maven-surefire-plugin:3.5.2:test (default-test) @ jpacman ---
[INFO] Using auto detected provider org.apache.maven.surefire.junit4.JUnit4Provider
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running jpacman.TestUtils
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.101 s -- in jpacman.TestUtils
[INFO] Running jpacman.PacmanTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.547 s -- in jpacman.PacmanTest
[INFO] Running jpacman.controller.ImageFactoryTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.007 s -- in jpacman.controller.ImageFactoryTest
[INFO] Running jpacman.model.ObserverTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s -- in jpacman.model.ObserverTest
[INFO] Running jpacman.model.CellTest
[ERROR] Tests run: 8, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.049 s <<< FAILURE! -- in jpacman.model.CellTest
[ERROR] jpacman.model.CellTest.testAdjacentAbove -- Time elapsed: 0.025 s <<< FAILURE!
java.lang.AssertionError:
        at org.junit.Assert.fail(Assert.java:91)
        at org.junit.Assert.assertTrue(Assert.java:43)
        at org.junit.Assert.assertTrue(Assert.java:54)
        at jpacman.model.CellTest.testAdjacentAbove(CellTest.java:62)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:566)
        at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:44)
        at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:15)
```

Figure 5: Add a JUnit test method `assertTrue(false);` statement. Disable Java assertions and rerun tests. Also exclude mock-based test cases.

We **note a curiosity** that ties into Section 3.5 on mocking. The method `Cell.adjacent(Cell)` asserts that its invariant holds. If we *disable* assertions, then this invariant will never be called! Then, if we make use of a mocked `Board`, then the expected `Board.withinBorders(int, int)` method calls on that mocked `Board` never take place, leading those mock based tests to fail. See Figure 6. So, mocking seems vulnerable to disabling assertions.



```
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   CellMockTest.testAdjacentAbove not all expectations were satisfied
expectations:
  ! expected once, never invoked: board.withinBorders(<2>, <2>); returns <true>
  ! expected once, never invoked: board.withinBorders(<1>, <2>); returns <true>
  ! expected once, never invoked: board.withinBorders(<2>, <2>); returns <true>
what happened before this: nothing!
```

Figure 6: Mock based testing fails when assertions are disabled, because the expected `assert invariant();` assertion never takes place, missing several expected method invocations on the mock object.

## 3.4 Part 4 – Coverage

This section pertains to code coverage reporting using the JaCoCo maven plugin, together with the maven site plugin.

### 3.4.1 Exercise 13

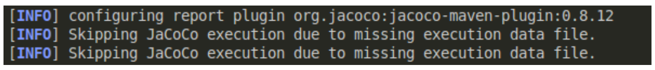
**Question:** Introduce the necessary modifications to `pom.xml` to run JaCoCo when executing `mvn site`. Describe the process.

We cite the JaCoCo java library docs [3] and the JaCoCo maven plugin docs [4], [5] for detailed steps to include JaCoCo as a dependency. We also consulted an article on the maven site lifecyle phases [6].

First, in the `<build>` section we must modify the `maven-surefire-plugin` plugin `<argLine>` tag for compatibility with JaCoCo. This preventively resolves the generic *missing execution data file* error seen in Figure 7. See the aforementioned JaCoCo documentation for other requirements with regards to maven surefire configuration in `pom.xml`, which currently do not apply.

```xml
<!-- Change this line ... -->
<argLine>
  -enableassertions
</argLine>

<!-- ... to this. -->
<argLine>
  ${argLine} -enableassertions
</argLine>
```



```
[INFO] configuring report plugin org.jacoco:jacoco-maven-plugin:0.8.12
[INFO] Skipping JaCoCo execution due to missing execution data file.
[INFO] Skipping JaCoCo execution due to missing execution data file.
```

Figure 7: Maven surefire plugin `<argLine>` incompatibility.

Next, we must add JaCoCo as a plugin to the `<build>` section in `pom.xml`.

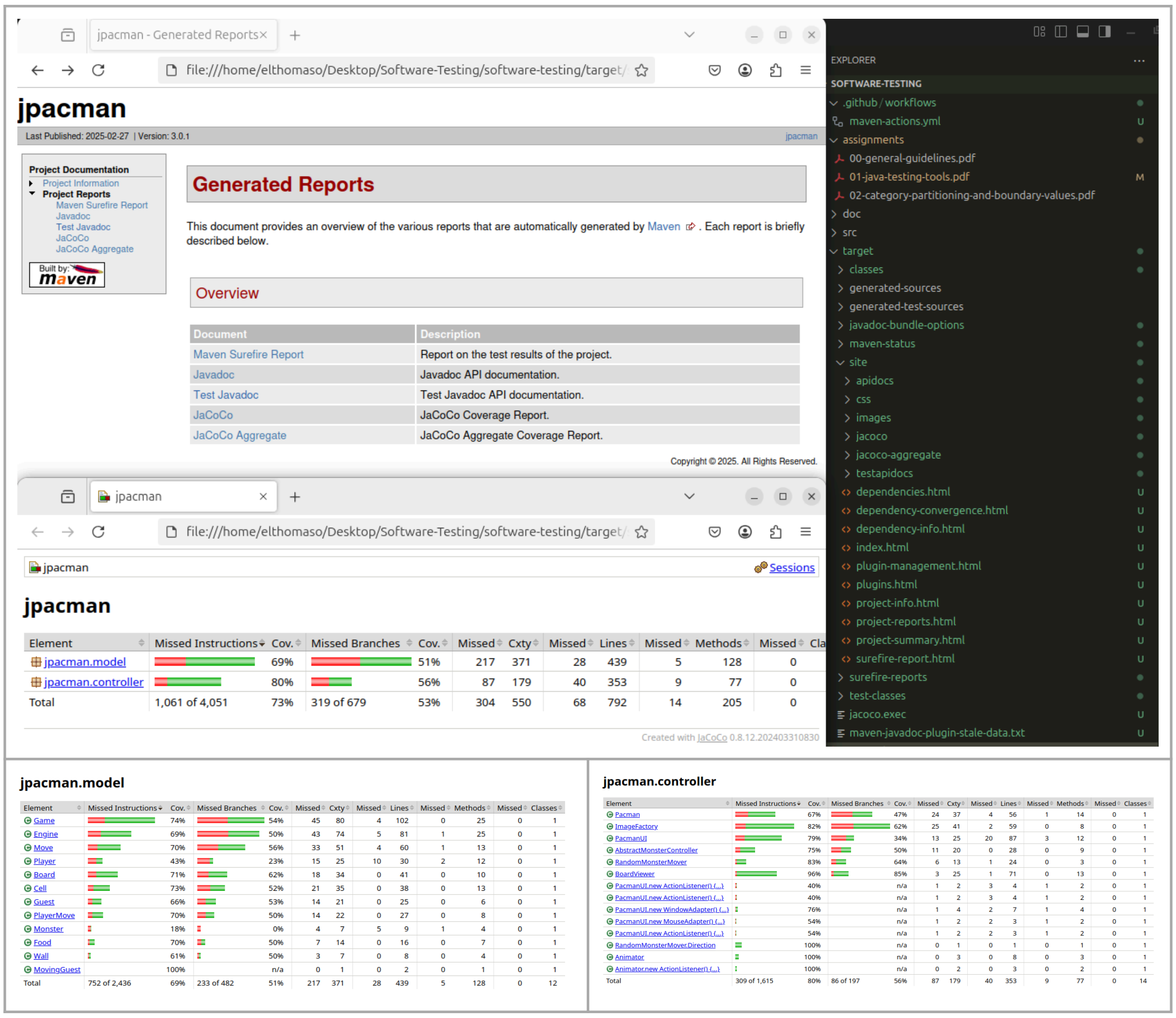
```xml
<build>
  <plugins>
    ...
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.8.12</version>
        <executions>
```

```
          <execution>
            <id>prepare-agent</id>
            <goals>
              <goal>prepare-agent</goal>
            </goals>
          </execution>
          <execution>
            <id>report</id>
            <phase>test</phase>
            <goals>
              <goal>report</goal>
            </goals>
          </execution>
        </executions>
    </plugin>
    ...
  </plugins>
</build>
```

The `prepare-agent` site goal binds to the `initialize` maven lifecycle phase and ensures coverage statistics can be properly collected and stored during the `test` maven lifecycle phase. The `report` site goal then generates the JaCoCo coverage report post testing. It is **crucial** that we bind the `report` execution to the test phase, `<phase>test</phase>`, since by default it binds to the `verify` phase instead, and the `site` lifecycle phase does not run the `verify` phase.

To subsequently present the coverage report in the *Project Reports* section of the website documentation generated by `mvn site`, we should also add JaCoCo as a plugin in the `<reporting>` section. See Table 2 for the completed report, part of the site.

```
<reporting>
  <plugins>
    ...
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
    </plugin>
    ...
  </plugins>
</reporting>
```



jpacman - Generated Reports

file:///home/elthomaso/Desktop/Software-Testing/software-testing/target/

# jpacman

Last Published: 2025-02-27 | Version: 3.0.1

Project Documentation
- Project Information
- Project Reports
  - Maven Surefire Report
  - Javadoc
  - Test Javadoc
  - JaCoCo
  - JaCoCo Aggregate

Built by: maven

## Generated Reports

This document provides an overview of the various reports that are automatically generated by Maven. Each report is briefly described below.

### Overview

| Document | Description |
|---|---|
| Maven Surefire Report | Report on the test results of the project. |
| Javadoc | Javadoc API documentation. |
| Test Javadoc | Test Javadoc API documentation. |
| JaCoCo | JaCoCo Coverage Report. |
| JaCoCo Aggregate | JaCoCo Aggregate Coverage Report. |

Copyright © 2025. All Rights Reserved.

EXPLORER

SOFTWARE-TESTING
```
.github / workflows
  maven-actions.yml                          U
assignments
  00-general-guidelines.pdf
  01-java-testing-tools.pdf                  M
  02-category-partitioning-and-boundary-values.pdf
doc
src
target
  classes
  generated-sources
  generated-test-sources
  javadoc-bundle-options
  maven-status
  site
    apidocs
    css
    images
    jacoco
    jacoco-aggregate
    testapidocs
    dependencies.html                        U
    dependency-convergence.html              U
    dependency-info.html                     U
    index.html                               U
    plugin-management.html                   U
    plugins.html                             U
    project-info.html                        U
    project-reports.html                     U
    project-summary.html                     U
    surefire-report.html                     U
  surefire-reports
  test-classes
  jacoco.exec                                U
  maven-javadoc-plugin-stale-data.txt        U
```

jpacman

file:///home/elthomaso/Desktop/Software-Testing/software-testing/target/

jpacman — Sessions

## jpacman

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed |
|---|---|---|---|---|---|---|---|---|---|---|---|
| jpacman.model | | 69% | | 51% | 217 | 371 | 28 | 439 | 5 | 128 | 0 |
| jpacman.controller | | 80% | | 56% | 87 | 179 | 40 | 353 | 9 | 77 | 0 |
| Total | 1,061 of 4,051 | 73% | 319 of 679 | 53% | 304 | 550 | 68 | 792 | 14 | 205 | 0 |

Created with JaCoCo 0.8.12.202403310830

### jpacman.model

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Game | | 74% | | 54% | 45 | 80 | 4 | 102 | 0 | 25 | 0 | 1 |
| Engine | | 69% | | 50% | 43 | 74 | 5 | 81 | 1 | 25 | 0 | 1 |
| Move | | 70% | | 56% | 33 | 51 | 4 | 60 | 1 | 13 | 0 | 1 |
| Player | | 43% | | 23% | 15 | 25 | 10 | 30 | 2 | 12 | 0 | 1 |
| Board | | 71% | | 62% | 18 | 34 | 0 | 41 | 0 | 10 | 0 | 1 |
| Cell | | 73% | | 52% | 21 | 35 | 0 | 38 | 0 | 13 | 0 | 1 |
| Guest | | 66% | | 53% | 14 | 21 | 0 | 25 | 0 | 6 | 0 | 1 |
| PlayerMove | | 70% | | 50% | 14 | 22 | 0 | 27 | 0 | 8 | 0 | 1 |
| Monster | | 18% | | 0% | 4 | 7 | 5 | 9 | 1 | 4 | 0 | 1 |
| Food | | 70% | | 50% | 7 | 14 | 0 | 16 | 0 | 7 | 0 | 1 |
| Wall | | 61% | | 50% | 3 | 7 | 0 | 8 | 0 | 4 | 0 | 1 |
| MovingGuest | | 100% | | n/a | 0 | 1 | 0 | 2 | 0 | 1 | 0 | 1 |
| Total | 752 of 2,436 | 69% | 233 of 482 | 51% | 217 | 371 | 28 | 439 | 5 | 128 | 0 | 12 |

### jpacman.controller

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pacman | | 67% | | 47% | 24 | 37 | 4 | 56 | 1 | 14 | 0 | 1 |
| ImageFactory | | 82% | | 62% | 25 | 41 | 2 | 59 | 0 | 8 | 0 | 1 |
| PacmanUI | | 79% | | 34% | 13 | 25 | 20 | 87 | 3 | 12 | 0 | 1 |
| AbstractMonsterController | | 75% | | 50% | 11 | 20 | 0 | 28 | 0 | 9 | 0 | 1 |
| RandomMonsterMover | | 83% | | 64% | 6 | 13 | 1 | 24 | 0 | 3 | 0 | 1 |
| BoardViewer | | 96% | | 85% | 3 | 25 | 1 | 71 | 0 | 13 | 0 | 1 |
| PacmanUI.new ActionListener() {...} | | 40% | | n/a | 1 | 2 | 3 | 4 | 1 | 2 | 0 | 1 |
| PacmanUI.new ActionListener() {...} | | 40% | | n/a | 1 | 2 | 3 | 4 | 1 | 2 | 0 | 1 |
| PacmanUI.new WindowAdapter() {...} | | 76% | | n/a | 1 | 4 | 2 | 7 | 1 | 4 | 0 | 1 |
| PacmanUI.new MouseAdapter() {...} | | 54% | | n/a | 1 | 2 | 2 | 3 | 1 | 2 | 0 | 1 |
| PacmanUI.new ActionListener() {...} | | 54% | | n/a | 1 | 2 | 2 | 3 | 1 | 2 | 0 | 1 |
| RandomMonsterMover.Direction | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| Animator | | 100% | | n/a | 0 | 3 | 0 | 8 | 0 | 3 | 0 | 1 |
| Animator.new ActionListener() {...} | | 100% | | n/a | 0 | 2 | 0 | 3 | 0 | 2 | 0 | 1 |
| Total | 309 of 1,615 | 80% | 86 of 197 | 56% | 87 | 179 | 40 | 353 | 9 | 77 | 0 | 14 |

Table 2: Generated JaCoCo coverage report added to the `mvn site` website.

### 3.4.2 Exercise 14

**Question:** 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?

Refer to Table 2 for the high level coverage numbers. JaCoCo specifies both instruction coverage and condition coverage.

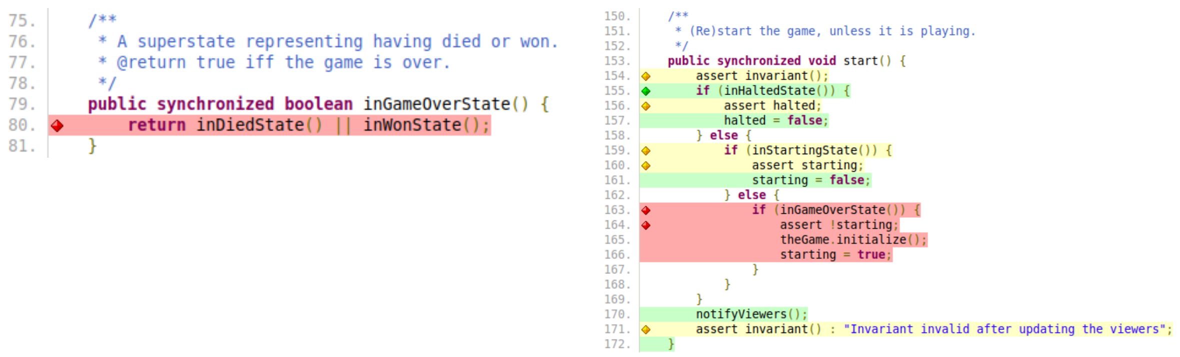
1. See Table 2. Overall, the `jpacman.model` module has 69% instruction coverage and 51% condition coverage, while the `jpackman.controller` module has 80% instruction coverage and 56% condition coverage. This reveals a focus on line coverage over condition coverage. While most lines are covered, conditionals are often sources of bugs since it is hard to reason about all of their possible outcomes, so they should be properly tested (domain-based testing, …).
2. See Table 3. There are multiple functions that specify invariants. Most of these functions have lacking condition coverage of around 50%. The programmer usually assumes the invariants are reliable. Leaving them insufficiently tested leads to hard to uncover bugs.
3. See Table 4. The `jpackman.model.Engine.inGameOverState()` function consists of a single line but evaluates a critical condition: is the game over? This functions has 0% instruction coverage and 0% condition coverage. Related is that the only non-covered lines in `jpackman.model.Engine.start()` are contained in a branch with as branch

condition `inGameOverState()`. These are the only sections of code related to the function `inGameOverState()`.

According to the module-level coverage statistics in Table 2, 14 out of 205 methods and 68 out of 792 lines are not covered by the test suite.

| Function | Line coverage | Condition coverage |
|---|---|---|
| **module** `jpacman.controller` | | |
| `AbstractMonsterController.controllerInvariant()` | 90% | 50% |
| `ImageFactory.invariant()` | 93% | 60% |
| `Pacman.invariant()` | 92% | 50% |
| `Engine.invariant()` | 74% | 37% |
| **module** `jpacman.model` | | |
| `Board.invariant()` | 90% | 50% |
| `Cell.boardInvariant()` | 93% | 50% |
| `Cell.guestInvariant()` | 91% | 75% |
| `Cell.invariant()` | 90% | 50% |
| `Food.foodInvariant()` | 90% | 50% |
| `Game.invariant()` | 90% | 50% |
| `Guest.guestInvariant()` | 91% | 75% |
| `Move.moveInvariant()` | 95% | 70% |
| `Player.playerInvariant()` | 90% | 50% |
| `PlayerMove.invariant()` | 94% | 50% |

Table 3: Invariant coverage rates.



```java
    /**
     * A superstate representing having died or won.
     * @return true iff the game is over.
     */
    public synchronized boolean inGameOverState() {
        return inDiedState() || inWonState();
    }
```

```java
    /**
     * (Re)start the game, unless it is playing.
     */
    public synchronized void start() {
        assert invariant();
        if (inHaltedState()) {
            assert halted;
            halted = false;
        } else {
            if (inStartingState()) {
                assert starting;
                starting = false;
            } else {
                if (inGameOverState()) {
                    assert !starting;
                    theGame.initialize();
                    starting = true;
                }
            }
        }
        notifyViewers();
        assert invariant() : "Invariant invalid after updating the viewers";
    }
```

Table 4: Coverage of `inGameOverState()`.

### 3.4.3 Exercise 15

**Question:** What color are most of the assert statements? Why? How does this affect the percentages provided by JaCoCo?

Note that this exercise refers to Java assert statements, **not** JUnit assert statements. Most assert statements are highlighted in **yellow**. In principle assertions evaluate the boolean value of an expression. For example, Equation 1 shows that $z \in \mathbb{R}$ can be cast to (interpreted as) a boolean value $B_1$, and that for $x, y \in \mathbb{R}$ the operation $x < y$ naturally evaluates to a boolean value $B_2$.

$$(x < y) \vee z = B_1 \vee B_2 \quad \text{where} \quad x, y, z \in \mathbb{R} \tag{1}$$

Java implements **short-circuiting expressions**, meaning that Java keeps evaluating parts of the expression only until the whole expression is sure to evaluate to true. So, every boolean term $B_i$

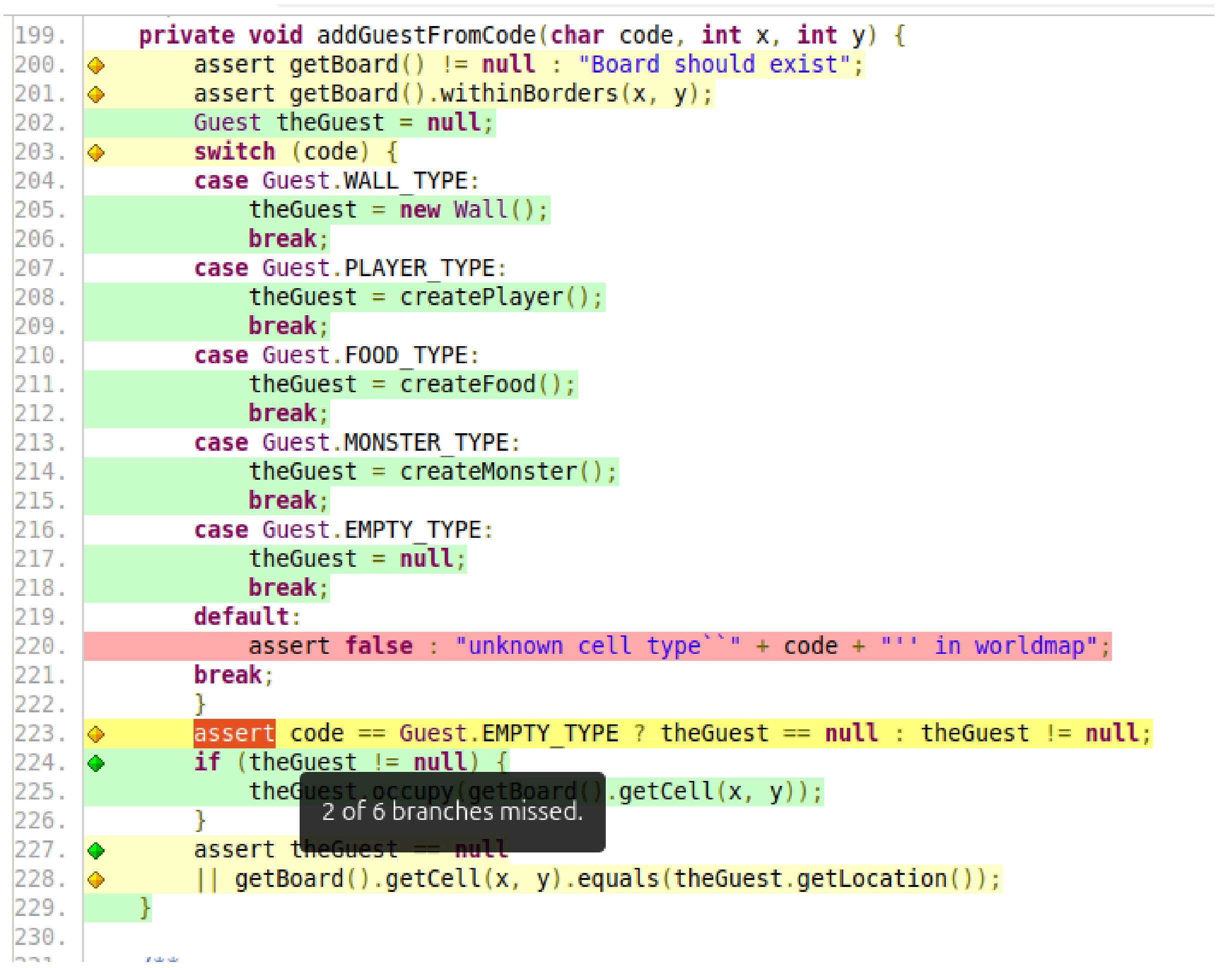
contributes two choices (branches) to the expression: 1) break out of the evaluation, or 2) continue with the evaluation of the next term $B_{i+1}$. Both options make use of a branch/jump instruction. In a sense, this is equivalent to using the ternary operator to implement a boolean expression, which makes the two branches (jumps) of each term explicit: `(x < y) ? true : (z ? true : false)`. In Figure 8 on line 223 this results in each of the three boolean terms of the expression contributing two branches to the assert for a total of six branch instructions.

$$\begin{aligned} \text{code} == \text{Guest.EMPTY\_TYPE} \quad ? \\ &(\text{theGuest} == \text{null} \quad ? \quad \text{true} : \text{false}) : \\ &(\text{theGuest} \neq \text{null} \quad ? \quad \text{true} : \text{false}) \end{aligned}$$

So, the number of branches counted by JaCoCo is literally the number of branch instructions that an expression consists of. This may be clearer in the form of pseudocode if-else statements.

```
if code == Guest.EMPTY_TYPE:  # branch statement nr. 1
    if theGuest == null:      # branch statement nr. 2
        return true
    else:                     # branch statement nr. 3
        return false
else:                         # branch statement nr. 4
    if theGuest != null:      # branch statement nr. 5
        return true
    else:                     # branch statement nr. 6
        return false
```

The “yellow coverage” **negatively reflects the branch coverage**, since yellow indicates only a subset of the branch statements was actually taken during the testing. The **line coverage does not suffer**, since every expression counts as a single line.



```java
    private void addGuestFromCode(char code, int x, int y) {
        assert getBoard() != null : "Board should exist";
        assert getBoard().withinBorders(x, y);
        Guest theGuest = null;
        switch (code) {
        case Guest.WALL_TYPE:
            theGuest = new Wall();
            break;
        case Guest.PLAYER_TYPE:
            theGuest = createPlayer();
            break;
        case Guest.FOOD_TYPE:
            theGuest = createFood();
            break;
        case Guest.MONSTER_TYPE:
            theGuest = createMonster();
            break;
        case Guest.EMPTY_TYPE:
            theGuest = null;
            break;
        default:
            assert false : "unknown cell type``" + code + "'' in worldmap";
        break;
        }
        assert code == Guest.EMPTY_TYPE ? theGuest == null : theGuest != null;
        if (theGuest != null) {
            theGuest.occupy(getBoard().getCell(x, y));
        }
        assert theGuest == null
        || getBoard().getCell(x, y).equals(theGuest.getLocation());
    }
```

2 of 6 branches missed.

Figure 8: An assert statement with > 2 branches.

## 3.5 Part 5 – Mocking

We were provided the article *Mocks Aren't Stubs* [7] as context for mocking. We refer to the *jMock Cookbook* documentation for using JMock as well [8].

We add JMock as a `<dependency>` to `pom.xml`. Note that the JUnit `<dependency>` **must** be listed **after** the JMock one, because JMock includes a library that JUnit requires.

```xml
<dependencies>
  ...
  <dependency>
    <groupId>org.jmock</groupId>
    <artifactId>jmock-junit4</artifactId>
    <version>2.13.1</version>
    <scope>test</scope>
  </dependency>
  ...
</dependencies>
```

### 3.5.1 Exercise 18

**Question:** Reconsider some of the functional tests you wrote in Exercise 5. Write them now using mocks. Use a mock library of your choice.

We make use of the JMock mocking library, since the *Mocks Aren't Stubs* article names it as a good entry point to learning mocking. We introduce a new test file to the testing suite to house the mock-based Cell adjacency tests, `/src/test/java/jpacman/model/CellMockTest.java`.

We assume the following implementation of `Cell.adjacent(Cell)`.

```java
/**
 * Determine if the neighbor is an immediate neighbor.
 * of the current cell (up, down, left or right)
 * @return true if the other cell is immediately adjacent
 */
public boolean adjacent(Cell otherCell) {
    assert invariant();
    int dx = Math.abs(this.x - otherCell.x);
    int dy = Math.abs(this.y - otherCell.y);
    assert invariant();
    return (dx == 1 && dy == 0) || (dx == 0 && dy == 1);
}
```

Since the Cell class is the object under test, it does not make sense to mock the parameter to `adjacent(Cell)`. Inspecting the method implementation, only the invariant assertion potentially interacts with different class objects: `Board` and `Guest`. We note that the Cell constructor requires a `Board` object as input.

```java
public Cell(int xCoordinate, int yCoordinate, Board b) {
    x = xCoordinate;
    y = yCoordinate;
    this.board = b;
    this.inhabitant = null;
    assert invariant();
}

protected boolean invariant() {
    return guestInvariant() && boardInvariant();
}

protected boolean boardInvariant() {
    return board != null && board.withinBorders(x, y);
}

public boolean guestInvariant() {
      return (inhabitant == null) || (inhabitant.getLocation() == this);
}
```

It sets the Cell's `Guest` member to null, and assigns the passed `Board` parameter to a member. Thus in our case the `Cell.invariant()` call only interacts with a non-null `Board` member. **We subsequently mock the `Board` class.**

Note that all tests of `Cell.adjacent(Cell)` are fairly simple; one assert per test suffices. Using JMock, **we only need to expect any function calls upon `Cell.board`**: only `board.withinBorders(x, y)` in the member function `Cell.boardInvariant()` will be called. It is notable that `Board.withinBorders(int, int)` is **not yet implemented**, so mocking Board is actually necessary in this circumstance, for that sake of stability of the tests.

All adjacency tests are very similar. They construct two Cell objects, and then assert their (non) adjacency. Consequently their expectations of which functions will be called on the ´Board´ mock are similar. The two Cell constructor calls plus the `Cell.adjacent` call result in exactly three calls to `Cell.invariant()`, which interacts with the `Board` Cell member.

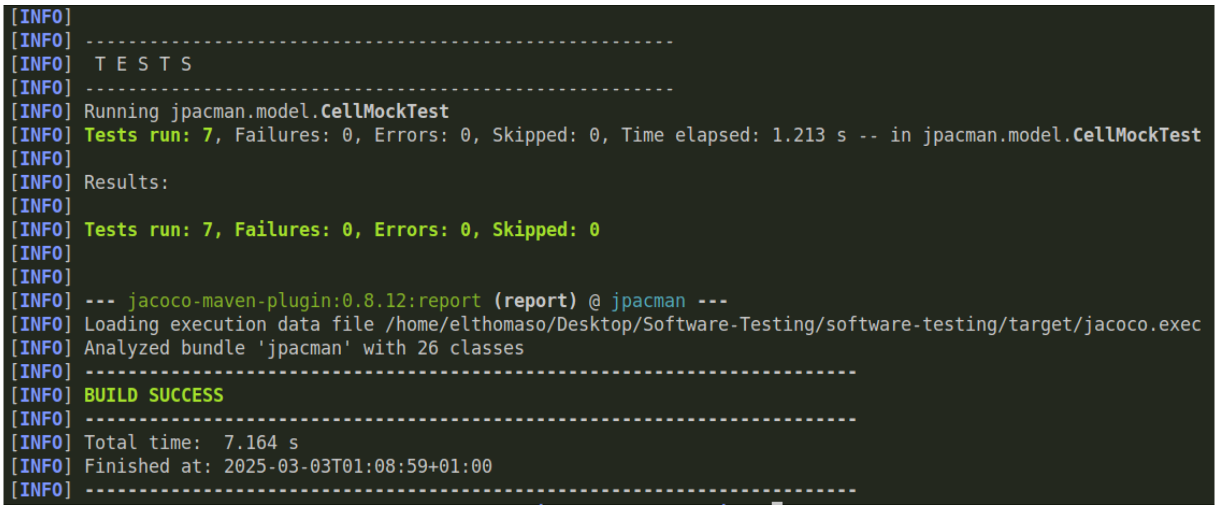
```java
public Board setUpAdjacentMock(int xa, int ya, int xb, int yb) {
    /* Setup. */
    Board mockBoard = context.mock(Board.class);

    /* Expectations. */
    context.checking(new Expectations() {{
        // CellA constructor Cell.invariant() call.
        oneOf(mockBoard).withinBorders(xa, ya); will(returnValue(true));
        // CellB constructor Cell.invariant() call.
        oneOf(mockBoard).withinBorders(xb, yb); will(returnValue(true));
        // CellA.adjacent(CellB) Cell.invariant() call.
        oneOf(mockBoard).withinBorders(xa, ya); will(returnValue(true));
        oneOf(mockBoard).withinBorders(xa, ya); will(returnValue(true));
    }});

    return mockBoard;
}

@Test
public void testAdjacentAbove() {
    Board mockBoard = setUpAdjacentMock(2,2, 1,2);

    /* Execute. */
    Cell cellA = new Cell(2, 2, mockBoard);
    Cell cellB = new Cell(1, 2, mockBoard);
    assertTrue(cellA.adjacent(cellB));
}
```



```
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running jpacman.model.CellMockTest
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.213 s -- in jpacman.model.CellMockTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.12:report (report) @ jpacman ---
[INFO] Loading execution data file /home/elthomaso/Desktop/Software-Testing/software-testing/target/jacoco.exec
[INFO] Analyzed bundle 'jpacman' with 26 classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  7.164 s
[INFO] Finished at: 2025-03-03T01:08:59+01:00
[INFO] ------------------------------------------------------------------------
```

Figure 9: All JMock tests in `jpacman.model.CellMockTest` succeed.

### 3.5.2 Exercise 19

**Question:** Is the coverage resulting from the mock-based test the same as the original? Compare both approaches. When would you prefer mock testing?

Yes, in our case the **coverage result is the same when doing mocking compared to non-mocking**.

Only the `Board` class was mocked, and only the `Board.withinBorders(int, int)` method then gets called on that mock. Although that method has zero coverage, calling that method on the mock does **not** count as coverage for it, since the function implementation does not actually get invoked. The object/function under test is `Cell.adjacent(Cell)`. Its implementation gets invoked by both the non-mock and the mock tests, so there is no change in its coverage.

Functional tests focus on state. They create some objects, assert their state, make some changes and then assert the changed state. The focus is thus on snapshots of the input and results of functions, while the internal implementation of the function is preferably opaque to the test. Stub implementations of assisting classes, those that are not specifically under test, are required. They provide a rudimentary or dummy implementation of the original class' interface. Though even a dummy implementation of a complex service, such as a database, may still constitute a significant amount of work and complexity.

Mock based testing specifies expectations on function invocations on mocked objects. The focus is less on the intermediate state changes of objects, and more on the order, parameters and results of invoked functions. Tests depend on knowledge of the internal implementations, else you can not predict function invocations. But, mock based tests provide a more general and elegant solution than stubs to assisting classes. Even for complex interfaces, you only need to specify parameterized function invocations and what the return value or side effects are for each given test case, separately.

Mock based testing is preferred when you explicitly want to reason about the expectations the object under test has for the interfaces of the assisting classes. The interface of assisting classes, and by extension the implementation details of the class under test, must be known for the tests to fully utilize mocking. The use of mock objects promotes testing singular components in isolation, since mock objects are fairly easy to set up.

State based testing has a preference for evaluating state snapshots of black-box implementations of the class under test. The tests focus on the pre and post conditions, instead of expected behavior during the function executions under test. State based tests either use real implementations of assisting classes, or stubs. Both are relatively more expensive and complex to use than mocks, so there is a preference for testing multiple classes at once.

## References

[1] "maven-site plugins 3.3 java.lang.ClassNotFoundException: org.apache.maven.doxia.siterenderer.DocumentContent." Accessed: Feb. 24, 2025. [Online]. Available: https://stackoverflow.com/questions/51091539/maven-site-plugins-3-3-java-lang-classnotfoundexception-org-apache-maven-doxia

[2] "JUnit, Assert (JUnit API)." Accessed: Mar. 03, 2025. [Online]. Available: https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html

[3] "JaCoCo Java Code Coverage Library." Accessed: Feb. 17, 2025. [Online]. Available: http://www.eclemma.org/jacoco/

[4] "Apache, JaCoCo Maven Plug-in." Accessed: Feb. 26, 2025. [Online]. Available: https://maven.apache.org/plugins/maven-site-plugin/index.html

[5] "Eclemma, JaCoCo Maven Plug-in." Accessed: Feb. 27, 2025. [Online]. Available: https://www.eclemma.org/jacoco/trunk/doc/maven.html

[6] "AVAJAVA Web Tutorials, What are the phases of the maven site lifecycle?." Accessed: Feb. 28, 2025. [Online]. Available: https://www.avajava.com/tutorials/lessons/what-are-the-phases-of-the-maven-site-lifecycle.html

[7] "Mocks Aren't Stubs." Accessed: Feb. 28, 2025. [Online]. Available: https://martinfowler.com/articles/mocksArentStubs.html

[8] "JMock, The jMock Cookbook." Accessed: Mar. 02, 2025. [Online]. Available: http://jmock.org/cookbook.html