Software Testing Assignment 1: Java Testing Tools

Lars De Leeuw 20205693
Robbe Teughels 20211127

March 2025

Contents

1 MAVEN

1.1 Exercise 1

Describe the activities you performed. - *Robbe*

- **clean**
  "Cleans up artifacts created by prior builds."
- **validate**
  "Validate the project is correct and all necessary information is available."
- **compile**
  "Compile the source code of the project." It generates the binary classes in /target/classes.
- **test**
  "Test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed." The binary tests are generated in /target/test-classes.
- **package**
  "Take the compiled code and package it in its distributable format, such as a JAR." This file is generated in /target. It also runs all tests
- **verify**
  "Run any checks to verify the package is valid and meets quality criteria."
- **install**
  "Install the package into the local repository, for use as a dependency in other projects locally."
- **site**
  "Generates site documentation for this project." The documentation can be found under /target/site and opening any .html file.
- **deploy**
  "Done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects."

All text between quotation marks are found at https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

1.2 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? - *Robbe*

Two parts can be found in the report.

- **Project Information**
  This lists all the information needed to install and use the project. This includes all dependencies, plugins, copyrights and licenses. It "normally" has a description of what the project is.
- **Project Reports** *The Reports include an overview of all packages and their classes in Javadoc. All methods, fields and constructors can be found here with their documentation. For the test classes, the same information can be found in Test Javadoc.*
  The Surefire plugin includes the information of the executed tests when generated. This is the result of each test and the time needed to execute. This information is grouped by package, the tested class and the test itself providing a high-level overview and detailed results.

2 JUNIT

2.1 Exercise 4

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

Since the responsibility description provided in the doc-string does not explicitly mention anything about what to do incase the cell is part of another board or not valid, I have opted to make it a pre-condition for the method that the other cell lies **within** the **same** board. Thus it is not the responsibility of the method to verify this and no test are needed for these cases aswell.

- If the other cell lies and is up-adjacent, return true.
- If the other cell lies and is down-adjacent, return true.
- If the other cell lies and is left-adjacent, return true.
- If the other cell lies and is right-adjacent, return true.
- If the other cell lies and is not an immediate neighbor, return false.
- If the other cell lies on the same location, return false.

```
1     /**
2      * Determine if the other cell is an immediate neighbour
3      * of the current cell (up, down, left, or right).
4      * Precondition: the other cell should lie within the same
      board.
5      * @return true if the other cell is immediately adjacent.
6      */
7     public boolean adjacent(Cell otherCell) {
8         assert getBoard() == otherCell.getBoard();
9         assert otherCell.boardInvariant();
```

Listing 1: Adjacent docstring.

2.2 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?** - *Lars*

The aCell field already present in CellTest lies in the corner of the board and is thus not suited for 2/4 adjacent directions. To reduce code duplication I added a new Cell to the test setup with coordinates (1, 1) so it can be used for all 4 directions. There is also allot of code duplication for the first 4 test cases which I tried to resolve by using a parameterized test case but the repository in question uses JUnit 4 and proper support for this is only available in JUnit 5.

- testCellAdjacent1 and testCellAdjacent 2 test the adjacency for the y-axis.
- testCellAdjacent3 and testCellAdjacent4 test the adjacency for the x-axis.
- testCellAdjacent5 tests that a cell cannot be adjacent to its own location.
- testCellAdjacent6 tests that a cell cannot be adjacent to a cell when the euclidian distance between both locations is more than 1.

When running the freshly written test cases they fail, as expected because the method is not yet implemented. The tests can be found in **/src/test/-java/jpacman/model/CellTest.java**

```
1    @Test
2    public void testCellAdjacent1() {
3        Cell otherCell = new Cell(1, 2, aBoard);
4
5        assertTrue("Cell should be adjacent.", cell11.adjacent(
     otherCell));
6
7        assertTrue(cell11.invariant());
8    }
9
10   @Test
11   public void testCellAdjacent2() {
12       Cell otherCell = new Cell(1, 0, aBoard);
13
14       assertTrue("Cell should be adjacent.", cell11.adjacent(
     otherCell));
15
16       assertTrue(cell11.invariant());
17   }
18
19   @Test
20   public void testCellAdjacent3() {
21       Cell otherCell = new Cell(2, 1, aBoard);
22
```

```
23          assertTrue("Cell should be adjacent.", cell11.adjacent(
    otherCell));
24
25          assertTrue(cell11.invariant());
26      }
27
28      @Test
29      public void testCellAdjacent4() {
30          Cell otherCell = new Cell(0, 1, aBoard);
31
32          assertTrue("Cell should be adjacent.", cell11.adjacent(
    otherCell));
33
34          assertTrue(cell11.invariant());
35      }
36
37      @Test
38      public void testCellAdjacent5() {
39          assertFalse("Cell should not be adjacent to own location.",
     cell11.adjacent(cell11));
40
41          assertTrue(cell11.invariant());
42      }
43
44      @Test
45      public void testCellAdjacent6() {
46          Cell otherCell = new Cell(0, 0, aBoard);
47
48          assertFalse("Cell should not be adjacent.", cell11.adjacent
    (otherCell));
49
50          assertTrue(cell11.invariant());
51      }
```

Listing 2: JUnit testcases.

2.3 Exercise 6

Write a proper implementation of adjacent and rerun your test cases. De- scribe your development process. - *Lars*

Since I already thought about the functional test cases and how I would test the method I had a clear idea of how I wanted to implement the method. Since the responsibility of this method is very simple I implemented it in one go, but if the different edge cases would have been more complex I probably would have written code for each edge-case and ran the tests right after to look if the corresponding test passes.

```
1       /**
2        * Determine if the other cell is an immediate neighbour
3        * of the current cell (up, down, left, or right).
4        * Precondition: the other cell should lie within the same
    board.
5        * @return true if the other cell is immediately adjacent.
6        */
```

```
7       public boolean adjacent(Cell otherCell) {
8           assert getBoard() == otherCell.getBoard();
9           assert otherCell.boardInvariant();
10
11          boolean isAdjacent = false;
12
13          if (otherCell.getX() == getX())
14          {
15              isAdjacent = Math.abs(otherCell.getY() - getY()) == 1;
16          }
17          else if (otherCell.getY() == getY())
18          {
19              isAdjacent = Math.abs(otherCell.getX() - getX()) == 1;
20          }
21
22          assert invariant();
23          return isAdjacent;
24      }
25  }
```

Listing 3: Cell.adjactent() implementation.

3 ASSERTIONS

3.1 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 precon- dition, post-condition, and as class invariant within JPacman. List one example for each category (Pre-condition, post-condition and class invariant). - *Lars*

3.1.1 Pre-condition

In listing 4 the pre-condition of the Cell.setGuest(Guest) function is that the guest's location should be set at this cell. This gets asserted in line 17 of the listing.

```
1       /**
2        * Modify the guest of this cell. This method is needed by the
        Guest's
3        * occupy method which keeps track of the links in the Cell-
        Guest
4        * association.
5        * Precondition: the guest's location should be set at this
        Cell,
6        * and the current cell should not be occupied already
7        * by some other guest.
8        * <p>
9        * Observe that the
10       * class invariant doesn't hold at method entry -- therefore it
        's not a
```

```
11      * public method. On method exit, however, it is valid again.
12      *
13      * @param aGuest
14      *            The new guest of this cell.
15      */
16     void setGuest(Guest aGuest) {
17         assert aGuest.getLocation()==this;
18         assert inhabitant==null;
19 
20         inhabitant = aGuest;
21 
22         assert getInhabitant()==aGuest;
23         assert invariant();
24     }
```

Listing 4: Pre-condition example.

3.1.2 Post-condition

In listing 5 the post-condition of the Guest.occupy(Cell) is that both the cell and the guest have changed their pointers to reflect the occupation. This gets asserted in the lines 19-20 of the listing.

```
1      /**
2       * Occupy a non-null, empty cell.
3       * Precondition: the current Guest must not
4       * have occupied another cell, and the target cell should be
       empty.
5       * Postcondition: both the cell and the guest
6       * have changed their pointers to reflect the occupation.
7       *
8       * @param cell
9       *            New location for this guest.
10      */
11     public void occupy(Cell cell) {
12         assert guestInvariant();
13         assert cell.getInhabitant()==null;
14         assert location==null;
15 
16         location = cell;
17         cell.setGuest(this);
18 
19         assert cell.getInhabitant()==this;
20         assert location==cell;
21         assert guestInvariant();
22     }
```

Listing 5: Post-condition example.

3.1.3 Invariant

In listing 6 the invariant of the Cell class is asserted in line 16 at the end of the constructor.

```java
    /**
     * Create a new cell at a given position on the board.
     *
     * @param xCoordinate
     *            The X coordinate
     * @param yCoordinate
     *            The Y coordinate
     * @param b
     *            The board
     */
    public Cell(int xCoordinate, int yCoordinate, Board b) {
        x = xCoordinate;
        y = yCoordinate;
        this.board = b;
        this.inhabitant = null;
        assert invariant();
    }

    /**
     * Conjunction of all invariants.
     *
     * @return true iff all invariants hold.
     */
    protected boolean invariant() {
        return guestInvariant() && boardInvariant();
    }

    /**
     * A Cell should always be part of the Board given at
    construction.
     *
     * @return true iff this is the case.
     */
    protected boolean boardInvariant() {
        return board != null && board.withinBorders(x, y);
    }

     /**
     * @return true iff the invariant for guest association holds.
     */
     public boolean guestInvariant() {
            return (inhabitant == null) || (inhabitant.getLocation
    () == this);
     }
```

Listing 6: Invariant example.

3.2 Exercise 10

Explain the differences between the JUnit collection of assert methods and the Java assert statement. - *Lars*

The built-in Java assert statements use the **assert** keyword, this keyword then gets followed by a logical expression and optionally a string that will be pasted into the stack trace to aid debugging. For backward compatibility reasons, the built-in Java assertion checking is disables and has to be explicitly

enabled. When a built-in Java assertion's logical expression evaluates to false an AssertionError gets thrown by the JVM, this is an unchecked exception which indicates the occurrence of an unrecoverable error, since it is still an exception user-code can catch it but this is discouraged.

The JUnit collection of assert methods are methods obtained by extending the TestCase class. When their logical expression evaluates to false the test fails and JUnit logs this. Other benefits are the specialized assertions such as assertEqual which will compare two values and log both values incase they differ which allows for quicker understanding why certain testcases failed. These asserts are always enabled.

3.3 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.** - *Lars*

To achieve this I simply added the following code to one of our adjacent testcases.

```
1    // Assertion that will always fail.
2    assert false : "Assertion that will always fail.";
```

When I ran the tests the testcase in question failed. Which was suprising, I expected the testcase to be Error instead since an uncaught exception occurs during the test. To further understand this behavior I commented out the assertion and threw a RuntimeException instead. And this did cause the testcase to be Error. So this means JUnit handles AssertionErrors exceptions differently than other exceptions.

3.4 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 frame- work? Finally, undo your changes to the build file, rerun to check that the assertions indeed fail, and remove the failing assertions.** - *Lars*

In the pom.xml configuration I found the following:

```
1  <argLine>-enableassertions</argLine>
```

Which to my understanding enables the checking of the built-in Java asserts. I expected commenting out this line would disable the assertions but my installation did not cooperate. To disable the assertions when the test are run I had to bump the surefire plugin from version 2.3 -¿ 2.3.1, which is the oldest version which supports the following property:

```
<enableAssertions>false</enableAssertions>
```

Adding this to the configuration of the surefire plugin instead of the argLine property actually disabled the assertions during the test execution causes all tests to pass again. Enabling the asserts causes the testcase in question to fail again.

My conclusions on this topic are that the built-it Java assert should merely be used a tool during development. Since by default these are also disabled you can't rely on these assert statements to enforce any conditions for public functions. They are also useful for debugging especially when provided with a descriptive string to be pasted in the stack trace. Since these can easily be disabled they don't also don't necessarily be removed from production code. The JUnit assertions are clearly superior in the context of using the JUnit testing framework thanks to their multiple common use case variations and useful logging capabilities such as printing both values when assertEqual evaluates to false. The built-in Java assert can also be used for testcases since JUnit treats the AssertionError exception as a failure as opposed to an error which it does with other uncaught exceptions. But the built-in Java assert is inferior in every way for this usecase.

4 CODE COVERAGE

4.1 Exercise 13

Introduce the necessary modifications to pom.xml to run JaCoCo when executing mvn site. Describe the process. - *Robbe*

```
            <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>site</phase>
```

```
                    <goals>
                        <goal>report</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
```

Listing 7: jacoco plugin.

We have added the jacoco plugin to the project to collect data over the code coverage. To correcly setup the plugin, we define an executable: prepare-agent. This will collect the data and stored for future use. We then add the report executable and link it to the site phase, it uses the previous data to create the desired raport.

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <forkMode>pertest</forkMode>
            <argLine>-javaagent:${settings.localRepository}/org/
    jacoco/org.jacoco.agent/0.8.12/org.jacoco.agent-0.8.12-runtime.
    jar=destfile=target/jacoco.exec -enableassertions</argLine>
        </configuration>
      </plugin>
```

Listing 8: surefire plugin.

The prepare agent together with the surefire plugin. We must change the argLine to make sure that the data will be collected.

4.2 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? - *Robbe*

- model.Engine.moveMonster (%100 - n/a)

  This function has test cases and is fully covered. The problem is that there is no implementation present. The fact that the test cases succeed means that the tests of the required function are missing.

- controller.ImageFactory.getImage (%61 - %50)

  This function can throw an IOException when an image is not found. This is an important missing test to see how the system behaves in this case.

- controller.ImageFactory.playerAnimationCount (%66 - %50)

  This function is simple get-function with some asserts. The reason of the low coverage is that only the happy day scenarios are tested.

4.3 Exercise 15

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

Most assert statements are colored yellow. This means that the statement is executed at least once but not all branches of the expression are explored. These count only partially towards the instructions coverage and branches explored.

5 MOCKS

5.1 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. - *Lars*

I decided to use Mockito because it is number one in the Mocking category on mvnrepository and number 8 of all libraries on the same site. The syntax and use also feels very natural and is easy to read.

```
1     /**
2      * The mocked Cell with stubs for adjacent
3      */
4     @Mock
5     private Cell mockedCell;
6     /**
7      * Actually create the board and the cell. *
8      */
9     @Before
10    public void setUpBoard() {
11        aBoard = new Board(width, height);
12        // put the cell on an invariant boundary value.
13        aCell = new Cell(0, height - 1, aBoard);
14        cell11 = new Cell(1, 1, aBoard);
15
16        // Lars
17        // Setup mocked Cell(1,1)
18        mockedCell = mock();
19        // when(mockedCell.getX()).thenReturn(1);
20        // when(mockedCell.getY()).thenReturn(1);
21        when(mockedCell.invariant()).thenReturn(true);
```

Listing 9: Mock setup.

```
1     // Lars
2     @Test
3     public void testCellAdjacent1() {
4         Cell otherCell = new Cell(1, 2, aBoard);
5         when(mockedCell.adjacent(otherCell)).thenReturn(true);
6
7         assertTrue("Cell should be adjacent.", mockedCell.adjacent(
    otherCell));
```

```
8
9           assertTrue(mockedCell.invariant());
10      }
11
12      // Lars
13      @Test
14      public void testCellAdjacent2() {
15          Cell otherCell = new Cell(1, 0, aBoard);
16          when(mockedCell.adjacent(otherCell)).thenReturn(true);
17
18          assertTrue("Cell should be adjacent.", mockedCell.adjacent(
    otherCell));
19
20          assertTrue(mockedCell.invariant());
21      }
22
23      // Lars
24      @Test
25      public void testCellAdjacent3() {
26          Cell otherCell = new Cell(2, 1, aBoard);
27          when(mockedCell.adjacent(otherCell)).thenReturn(true);
28
29          assertTrue("Cell should be adjacent.", mockedCell.adjacent(
    otherCell));
30
31          assertTrue(mockedCell.invariant());
32      }
33
34      // Lars
35      @Test
36      public void testCellAdjacent4() {
37          Cell otherCell = new Cell(0, 1, aBoard);
38          when(mockedCell.adjacent(otherCell)).thenReturn(true);
39
40          assertTrue("Cell should be adjacent.", mockedCell.adjacent(
    otherCell));
41
42          assertTrue(mockedCell.invariant());
43      }
44
45      // Lars
46      @Test
47      public void testCellAdjacent5() {
48          when(mockedCell.adjacent(mockedCell)).thenReturn(false);
49          assertFalse("Cell should not be adjacent to own location.",
     mockedCell.adjacent(mockedCell));
50
51          assertTrue(mockedCell.invariant());
52      }
53
54      // Lars
55      @Test
56      public void testCellAdjacent6() {
57          Cell otherCell = new Cell(0, 0, aBoard);
58          when(mockedCell.adjacent(otherCell)).thenReturn(false);
59
60          assertFalse("Cell should not be adjacent.", mockedCell.
```

```
    adjacent(otherCell));

        assertTrue(mockedCell.invariant());
    }
}
```

Listing 10: Functional JUnit tests using mocks.

5.2 Exercise 19

> **Is the coverage resulting from the mock-based test the same as the origi- nal? Compare both approaches. When would you prefer mock testing?** - *Robbe*

The code coverage without using mock-based test is higher. This is because the functionalities that are mocked are not tested any more. This way we can test the components without relaying on other buggy components. Ensuring that this part works correctly.

This method is useful in case of parallel development where underlying components are still in development. For tested and correct underlying components, the benefit of mocking is lower than the actual setup cost of mocking.