Software Testing and Quality Engineering

STQE Labwork, IN3205-II - Part 1

Giovanni Martina, #1129228
Cees-Willem Hofstede, #1272012

Exercise 1

Code added to build.xml:

```
<target name=”doc” description=”Create Javadoc”>
      <mkdir dir="${basedir}/doc/javadoc/main"/>
      <javadoc sourcepath="src/main/java"
               destdir="doc/javadoc/main"
               author="true"
               version="true"
               use="true"
               windowtitle="Docs for exercise1">
          <doctitle><![CDATA[<h1>Docs for exercise1</h1>]]></doctitle>
          <bottom><![CDATA[<i>Copyright &#169; 2010 Cees-Willem Hofstede &
                  Giovanni Martina. All Rights Reserved.</i>]]>
          </bottom>
          <tag name="todo" scope="all" description="To do:"/>
      </javadoc>
</target>
```

First, we need to make sure that the directory in which we want the javadoc to be created exists, hence the mkdir. Furthermore we have to set some options like sourcepath, for which in this case we chose src/main/java since this is the directory that contains all the java files of the actual program and that is (for now) the thing we are most interested in. Also a destdir was needed to set the destination of the generated files (the folder that was created with mkdir).
Since the author deserves some credit the option author is set to true. Also the version number is set to be visible. The option use is set to true to create class and package usage pages.
What remains is the doctitle, which speaks for itself, and the bottom. We add some copyright information just for the sake of the exercise.

Exercise 2

A cell can only be adjacent to another one if they are both on the same board. Next, we need four checks to test the adjacency itself. We can define four distinct checks for that. Two for when the x of the other cell is equal to this cell. Either the y of the other cell must by equal to the y of this cell plus one or the y of this cell minus one. The same should be valid when x and y are interchanged. In all other cases the function should return false. Since Cell doesn't extend any other class we do not need to check for object type since it will not compile otherwise.

Exercise 3

The code shown here does not include the import statement and added global variables since those are not relevant to the specific test.

code added to CellTest.java:

```
/**
* Test if a cell is adjacent to another cell
* A cell is adjacent to another cell if it lies directly next to
* another cell. We have four possible directions to check
* Check if a cell that lies to the right, left, up or down.
* If so it should be adjacent
*/
@Test
public void testCellAdjacent() {
      //test down adjacency
      cellA = aBoard.getCell(2, 2);
      cellB = aBoard.getCell(2, 3);
      assertTrue(cellA.adjacent(cellB));

      //test up adjacency
      cellB = aBoard.getCell(2, 1);
      assertTrue(cellA.adjacent(cellB));

      //test left adjacency
      cellB = aBoard.getCell(1, 2);
      assertTrue(cellA.adjacent(cellB));

      //test right adjacency
      cellB = aBoard.getCell(3, 2);
      assertTrue(cellA.adjacent(cellB));

      //test if a cell is NOT adjacent
      cellB = aBoard.getCell(3, 1);
      assertFalse(cellA.adjacent(cellB));

      //test if a cell on another board is NOT adjacent
      cellB = differentBoard.getCell(2, 3);
      assertFalse(cellA.adjacent(cellB));
}
```

code added to Cell.java

```
/**
* Determine if the other cell is an immediate
* neighbour of the current cell.
* @return true iff the other cell is immediately adjacent.
*/
public boolean adjacent(Cell otherCell) {
      return false;
}
```

We cannot make an empty body for adjacent since then it will not compile. Instead we let it return false on every input. This caused all the checks to fail.

Exercise 4

Changed code for adjacent() in Cell.java:

```
public boolean adjacent(Cell otherCell) {
      boolean adjacent = false;
      if(this.getBoard() == otherCell.getBoard())
      {
            if(this.getX() == otherCell.getX()) {
                  adjacent = otherCell.getY() == this.getY() - 1 ||
                  otherCell.getY() == this.getY()+1;
            }
            else if(this.getY() == otherCell.getY()) {
                  adjacent = otherCell.getX() == this.getX() - 1 ||
                  otherCell.getX() == this.getX()+1;
            }
      }
      return adjacent;
}
```

After these changes all test pass as expected.

Exercise 5 + 6 + 7 + 8

The following target has been added to our build.xml in order to allow for testing of java source files.

```
<property name="checkstylejar" location="${lib.dir}/checkstyle-all-5.0.jar"/>
<taskdef resource="checkstyletask.properties" classpath="${checkstylejar}"/>
<target name="checkstyle" description="Check all source files for code convention
violations">
<checkstyle config="checkstyle.xml">
      <fileset dir="src/main/java" includes="**/*.java"/>
      <formatter type="plain"/>
</checkstyle>
</target>
```

After extending build.xml with the checkstyle target the following warnings were given (without duplicates):

```
- '3' is a magic number.
- Expected @param tag for 'otherCell'.
- 'if' is not followed by whitespace.
- '{' should be on the previous line.
- Line is longer than 100 characters
- '+' is not followed by whitespace.
```

Most of these warnings were fixed including the 'Line is longer than..' warning. We both agree that although it might be helpful when you have a smaller screen, it takes more effort to make all the lines shorter than 100 characters. The rest of the warnings might be fair although some of them contrary too what we are used to (like placing { after if statement on the next line).

Exercise 9

Module **LineLength**:

This module checks for text lines in the source code that exceed some predefined maximum value for text lines. Checkstyle calculates the length of a line and verifies if it does not exceed the maximum specified in checkstyle.xml which is currently set to 100.

Rationale:
If a developer chooses to print some code on paper, long lines prove hard to read because they won't fit on one line. Screen space could also be limited due to IDEs with large panels on the sides.

Module **RightCurly**:
This module checks the placement of the right curly brace "}" for *try, else* and *catch* statements. Currently it's being ignored due to the setting in checkstyle.xml so placement of this curly brace does not matter. Default value is to make sure every right curly brace starts a new line.

Rationale:
Not given in the checkstyle documentation but we could imagine it being better always having curly braces starting on new lines just to enforce a single coding style and not having people use their own differing personal preferences on a single project.

Module **AvoidStartImport**:
This module checks against package level importing using the * notation. The default has been set in checkstyle.xml so package level importing is not allowed.

Rationale:
Package level importing could lead to unexpected name clashes.

Exercise 10

Enable Module
Module **DefaultComesLast:**
Check that the default is after all the cases in a switch statement.

Rationale:
Java allows default anywhere within the switch statement. But it is more readable if it comes after the last case.

Disable Module
Module **MethodLength:**
Checks for long methods and constructors.

Rationale:
If a method becomes very long it is hard to understand. Therefore long methods should usually be refactored into several individual methods that focus on a specific task.

Exercise 11

Precondition:
In class Food in method meetPlayer there is an assert "theMove != null" that checks that if the function argument isn't null.

Postcondition:
In class Engine in method initialize there is an assert inStartingState that checks whether the object is in fact initialized meaning that it is in the starting state.

Class invariant:
In class AbstractMonsterController in method AbstractMonsterController, there is an assert controllerInvariant which checks for some variables whether they are null.

Exercise 12

In class Pacman in method Pacman we added "assert 1 == 2", which will obviously fail. When running the test this was the output:

```
Buildfile: /home/sdy/workspace/jPacMan/jpacman-4.3.2/build.xml
init:
compile:
compile-tests:
   [javac] Compiling 1 source file to /home/sdy/workspace
         /jPacMan/jpacman-4.3.2/target/test-classes
check:
   [junit] Running jpacman.TestAll
   [junit] Testsuite: jpacman.TestAll
   [junit] Tests run: 17, Failures: 0, Errors: 1, Time elapsed:
          1.252 sec
   [junit] Tests run: 17, Failures: 0, Errors: 1, Time elapsed:
          1.252 sec
   [junit] Testcase:testTopLevelAlphaOmega(jpacman.PacmanTest):
          Caused an ERROR
   [junit] null
   [junit] java.lang.AssertionError
   [junit]     at jpacman.controller.Pacman.<init>
         (Pacman.java:42)
   [junit]     at jpacman.PacmanTest.testTopLevelAlphaOmega
                 (PacmanTest.java:69)
   [junit] Test jpacman.TestAll FAILED
BUILD SUCCESSFUL
Total time: 4 seconds
```

Even though there was an error, caused by the assert, the build ends successfully.

Exercise 13

In target with name "check" in the block "junit" we commented the line
`<jvmarg value="-enableassertions"/>`
so that it read
`<!-- <jvmarg value="-enableassertions"/> -->`
After this change the output of the test is as followes:

```
Buildfile: /home/sdy/workspace/jPacMan/jpacman-4.3.2/build.xml
init:
compile:
compile-tests:
   [javac] Compiling 1 source file to /home/sdy/workspace
         /jPacMan/jpacman-4.3.2/target/test-classes
check:
   [junit] Running jpacman.TestAll
   [junit] Testsuite: jpacman.TestAll
   [junit] Tests run: 17, Failures: 0, Errors: 0, Time elapsed:
          1.506 sec
   [junit] Tests run: 17, Failures: 0, Errors: 0, Time elapsed:
          1.506 sec
BUILD SUCCESSFUL
Total time: 3 seconds
```

The output now doesn't show any errors. If we uncomment the line we just commented and rerun the test the error returns as expected.

Exercise 14

With the addition of assertions the implementation of *adjacent()* in Cell.java has changed to the following.

```
/**
 * Determine if the other cell is an immediate
 * neighbour of the current cell.
 * @return true iff the other cell is immediately adjacent.
 * @param otherCell the other cell to check against.
 */
public boolean adjacent(Cell otherCell) {
  assert otherCell != null;

  boolean adjacent = false;
  if (this.getBoard() == otherCell.getBoard()) {
        if (this.getX() == otherCell.getX()) {
              adjacent = otherCell.getY() == this.getY() - 1
                              ||  otherCell.getY() == this.getY() + 1;
        }
        else if (this.getY() == otherCell.getY()) {
              adjacent = otherCell.getX() == this.getX() - 1
                              || otherCell.getX() == this.getX() + 1;
        }
        else {
              assert !adjacent;
        }
  }
  else {
        assert !adjacent;
  }

  assert invariant();
  return adjacent;
}
```

We've added assertions to check whether the argument otherCell is an instantiated object. Before returning the method result we assert the invariant and at several places where we initially assumed adjacency to be false we places assertions to test our assumptions.

Exercise 15

The main difference of course is that JUnit is used for testing purposes while the java asserts are used to enforce code-by-contract development. Also, the java assert statements are included in the function itself and the JUnit asserts are defined in a testclass. JUnit is used for automated tests which have to be run seperately from the program (or actually the tests run the program), while the asserts are used during compilation of the program itself.

Exercise 16

Browsing through the classes Cell, Player and Engine there are several methods that have a coverage of 0%. These methods include Engine.inGameOverState(), Player.die(), Cell.getX() and Cell.getY(). These values are interesting because they indicate those particular parts of the code were never called. During the run we just moved a couple of spaces and did not die or lose the game so it isn't strange some of these methods were never called. However you could assume during play the x, y values of cells need to be known so getX() and getY() would have to be called. Seeing as that's not the case some part of the game is missing or not yet implemented.

Exercise 17

Some parts were never covered. One example is the adjacent method we implemented in an earlier exercise. The reason for this is simple, the method is never called by any other method. So no matter how many times we run the game in different ways, this method will never be covered. Some other methods suffer from this as well, like the getY() method in the same class. This is currently only called from our adjacent method and in some testcases, but never during the execution of the program.
None of the methods in class GameLoadException ever get covered which basically means that no GameLoadExceptions occurs during execution.

Exercise 18

It's interesting to look at the coverage for our implementation of Cell.adjacent() which shows a percentage of 80% indicating our implementation is pretty good due to our tests causing a high code coverage.

It's also interesting to note that after running the tests PacmanUI.keyPressed() has 0% coverage. None of the tests have reached this method so it is currently not being tested. We can asks ourselves if these user interface methods don't need to be tested as well.

The implementation of Monster.meetPlayer() after executing all tests has 0% code coverage indicating we didn't sufficiently test this method as well.

Exercise 19

Most of the asserts that do get covered are yellow. Lines become yellow when they are only partly covered. Partly covered means that not all of the basic blocks associated with this line have been executed during the coverage session. This means that the lines are covered and that it will add to the percentage, but that it doesn't cover all the blocks and so it will affect the percentage for that.
This happens for the asserts especially since they are mostly actually if-then statements. For example, if we have an "assert someBooleanMethod()", then this is in fact a method that does something like "if someBooleanMethod() then return true else return false". This means that if someBooleanMethod() returns true the else part of the assert is never covered.

Exercise 20

The pattern to use here would be "basic decision logic" since it is basically just checking whether a value is in or out of bounds. As test cases we have for both x and y that it should be larger than or equal to 0 and that is should be smaller than or equal to the width or height accordingly.

Exercise 21

Here's a list of cases we'd test:

- X and Y both within bounds (not equal to width or height). This should return true at all times.
- X and Y within bounds (both equal to their maximum allowed values). This should be true as well.
- One of X and Y in bound and the other out. We need to know that this returns false for both the cases.
- X and/or Y is negative. We have to make sure that negatives will never be inside bounds.

Exercise 22

The test:

```
    /**
     * Do some tests to make sure that the withinBorders
     * function returns the correct boolean value.
     */
    @Test
    public void testWithinBorders() {
      assertTrue(theBoard.withinBorders(0, 0));
      assertTrue(theBoard.withinBorders(width, height));
      assertTrue(theBoard.withinBorders(width - 1, height));
      assertTrue(theBoard.withinBorders(width, height - 1));
      assertTrue(theBoard.withinBorders(width - 1, height - 1));
      assertFalse(theBoard.withinBorders(width + 1, height));
      assertFalse(theBoard.withinBorders(width, height + 1));
      assertFalse(theBoard.withinBorders(width + 1, height + 1));
      assertFalse(theBoard.withinBorders(-width, -height));
    }
```

And the implementation of the withinBorders method:

```
    /**
     * Return true iff (x,y) falls within the borders of the board.
     *
     * @param x
     *            Horizontal coordinate of requested position
     * @param y
     *            Vertical coordinate of requested position.
     * @return True iff (x,y) is on the board.
     */
    public boolean withinBorders(int x, int y) {
         boolean withinBorders = x >= 0 && x <= width;
         withinBorders = withinBorders && y >= 0 && y <= height;
         return withinBorders;
    }
```

All tests pass as expected.

Exercise 23

The only invariant function missing implementation in Cell.java is guestInvariant(). This should check that the location of the Guest inhabitant is equal to the cell or that inhabitant is null.

```
    /**
     * A cell can be occupied by a guest and if so
     * the guest should occupy the cell.
     *
     * @return true iff this is the case.
     */
     public boolean guestInvariant() {
         return inhabitant == null || this.equals(inhabitant.getLocation());
     }
```

This passes the tests (including the checkstyle tests).

Exercise 24

The check is pretty much the same as in the previous exercise yielding the following invariant function.

```
     /**
      * The location is either null (upon creation) or
      * it is a Cell of which this is an inhabitant.
      *
      * @return true iff invariant holds.
      */
     protected boolean guestInvariant() {
       return location == null || this.equals(location.getInhabitant());
     }
```

This passes the tests as well.

Exercise 25

For this we only need to add two assert statements after the first guestInvariant assert.

```
assert this.location == null;
assert !aCell.isOccupied();
```

Exercise 26

Two asserts are needed just before the last guestInvariant assert.

```
assert aCell.equals(this.location);
assert this.equals(aCell.getInhabitant());
```

Exercise 27

We have to make sure that after deoccupy the inhabitant of the cell is null, so we need an assert for that, just before the last guestInvariant assert.

```
assert !oldLocation.isOccupied();
```

Exercise 28

After the addition of pre- and postconditions the setGuest() method takes the following form.

```
void setGuest(Guest aGuest) {
   assert this.equals(aGuest.getLocation());
   assert !this.isOccupied();
   inhabitant = aGuest;
   assert aGuest.equals(this.inhabitant);
   assert invariant();
}
```

Exercise 29

After the addition or pre- and postconditions the free() method takes the following form.

```
void free() {
   assert this.isOccupied();
   assert !this.equals(inhabitant.getLocation());
   inhabitant = null;
   assert !this.isOccupied();
   assert invariant();
}
```

Exercise 30

We test the occupy() and deoccupy() methods of Guest.java by adding the following test cases to GuestTest.java

```
/**
 * Test occupy and deoccupy for Guest cell occupation
 */
@Test
public void testOccupyDeoccupy() {
  theGuest.occupy(theCell);
  assertTrue(theGuest.getLocation().equals(theCell));
  assertTrue(theCell.getInhabitant().equals(theGuest));
  assertFalse(theGuest.getLocation() == null);
  assertFalse(theCell.getInhabitant() == null);

  theGuest.deoccupy();
  assertTrue(theGuest.getLocation() == null);
  assertTrue(theCell.getInhabitant() == null);
  assertNull(theGuest.getLocation());
  assertNull(theCell.getInhabitant());
}
```

We make sure that after occupation both Guest and Cell have correct references to each other and that after deoccupation these references have been cleared correctly.

Exercise 31

First we occupy the deoccupied Guest and then try to check if we end up generating an assertion error when we try to subsequently occupy the Guest a second time. Generation of an assertion error indicates the test has passed.

```
/**
 * Test an invalid occupy-occupy sequence
 */
@Test
public void testDoubleOccupy() {
  theGuest.occupy(theCell);
  boolean failureGenerated;
  if (TestUtils.assertionsEnabled()) {
        try {
              theGuest.occupy(theCell);
              failureGenerated = false;
        }
        catch (AssertionError ae) {
              failureGenerated = true;
        }

        assertTrue(failureGenerated);
  }
}
```

Exercise 32

Like in the previous exercise we try to generate an assertion error by performing an illegal call to Cell.setGuest(). If we end up generating an assertion error because the precondition of Guest having it's location already set to the calling Cell our test passes, if not, our test fails.

```
    /**
     *
     *  Test illegal use of Cell.setGuest()
     */
    @Test
    public void testIllegalsetGuest() {
      boolean failureGenerated;
      if (TestUtils.assertionsEnabled()) {
            try {
                  anotherCell.setGuest(anotherGuest);
                  failureGenerated = false;
            }
            catch (AssertionError ae) {
                  failureGenerated = true;
            }

            assertTrue(failureGenerated);
      }
    }
```