Test Plan

Name: José Duarte de Sousa Barroca
LNU e-mail: jd222qf@student.lnu.se
Link to GitHub 1DV600 repo: https://github.com/JBarroca/jd222qf_1dv600

Table of Contents

Objectives

Testing can have many different objectives, depending on when and how it is performed and the characteristics of the software being developed. These tests were performed by me (the developer) during an iteration of the software's development process, and can therefore be characterized as development testing activities.

The main objective with development testing is to find defects in the current state of the software's implementation. In this particular case, the objective was to find defects in the code implemented in previous iterations (more specifically, during assignment 2).

Time plan

Task	Estimated	Actual
Writing test plan introductory sections and reflection	01:00	01:45
Planning/writing manual test cases	02:00	02:30
Running manual tests	00:10	00:05
Writing manual tests' results	00:30	00:20
Automated unit tests	01:00	04:30

The most relevant difference is how much longer it took to perform automated unit testing. As discussed in the overall test plan reflection, this difference was due to the changes I had to make in several methods and constructors to make the code testable, which really was a valuable lesson.

Writing the test report also took a bit more time than anticipated, mostly spent on writing an adequate reflection and organizing my thoughts on the Test Objects and Testing Techniques sections.

Test objects

1. Manual tests

Both manual tests were conducted on the Use Case (UC) 2 (play single word game), because it is the best implemented and most complex use case in the project and there are many possible different scenarios to be tested.

For the first manual test, I chose a scenario that tests both UC1 (start game) as well an alternate scenario of UC2 with one of the many ways for the user to win the game (guessing the entire word). Encompassing UC1 in this test is good because, if successful, it may allow the assumption of a successful game start for further tests involving UC2. And testing a win-game scenario is also important and this stage, to find defects in this crucial step of the game.

For the second manual test, I chose a scenario where the user resets the game, because its implementation involves the system asking the user for confirmation, which is appropriate for manual testing.

2. Automated unit tests

The signatures of the two methods selected to test via automated unit testing are:

- public boolean inputIsValid (String input, ArrayList<String> guessedLetters)
- public String updateWord (String gameWord, ArrayList<String> guessedLetters)

I chose these methods for unit testing because, after having re-implemented them (see Reflection), they have simple inputs that can be manipulated when testing as well as simple outputs which can be easily used with JUnit's assertions.

Testing techniques

For this testing process, only dynamic testing techniques were used (i.e. techniques whereby the system is tested through its execution) and no static test were used.

Unit testing was performed on two separate methods using automated test cases written and executed in JUnit 5. Again, in the context of development testing, these were conducted to find faults in the implementation of these methods.

System testing was performed using use case-based testing and manual tests. The goal with system testing during development is to test the interaction between the system's different components and to test the system's emergent behaviours. Given the multiplicity of different interactions and the many resulting outcomes, it is often impractical to use automated tests for system testing.

Manual test cases

TC 2.1 Guessing an entire word correctly

Use cases tested: UC1 – Start game and UC2 – Play a single-word game

Scenario: The Player tries to guess the hidden word and succeeds (UC2 alternate scenario 5.6)

Precondition: The application is executed by specifying the String “starlight” as an argument to the Hangman constructor in the game’s Main.java class.

Test steps

- Start the app
- The system shows the welcome message “---- Welcome to the world's best Hangman game ever! ----” as well as the start menu and waits for user input.
- Press “1” and enter.
- The system shows the new game menu and waits for user input.
- Press “1” and enter.
- The system displays a single game board containing the hidden word, the number of tries left (9) and the previously entered letters (none).
- The system shows “Please enter a letter or guess the entire word (or write 'resetgame' to return to start menu or 'quitgame' to quit the program):” and waits for user input.
- Write “starlight” and press enter.

Expected

- The system shows the win game board with the text “you won” and “you found starlight in 1 tries”, “1 – Return to start menu, 0 – Quit the application”.
- The system shows “Please press a key to select an option” and waits for user input.

Result

Succeeded	☒
Failed	☐
Comments:	-

TC 2.2 Resetting an ongoing game

Use case tested: UC2 Play a single-word game
Scenario: Resetting an ongoing game (alternate scenario 5.4, where the player enters the command ‘resetgame’ during an ongoing game)

Precondition: A single-word game is running and the system is waiting for the player’s input.

Test steps:

- The system shows “Please enter a letter or guess the entire word (or write 'resetgame' to return to start menu or 'quitgame' to quit the program):” and waits for user input.
- Write “resetgame” and press enter.
- The system shows “Do you really want to end this game and return to the start menu? ("y" = yes, "n" = no):”
- Write “y” and press enter.

Expected

- The system shows “resetting the game...”
- The system shows the welcome message “---- Welcome to the world's best Hangman game ever! ----” and the start menu.

Result

Succeeded	☒
Failed	☐
Comments	-

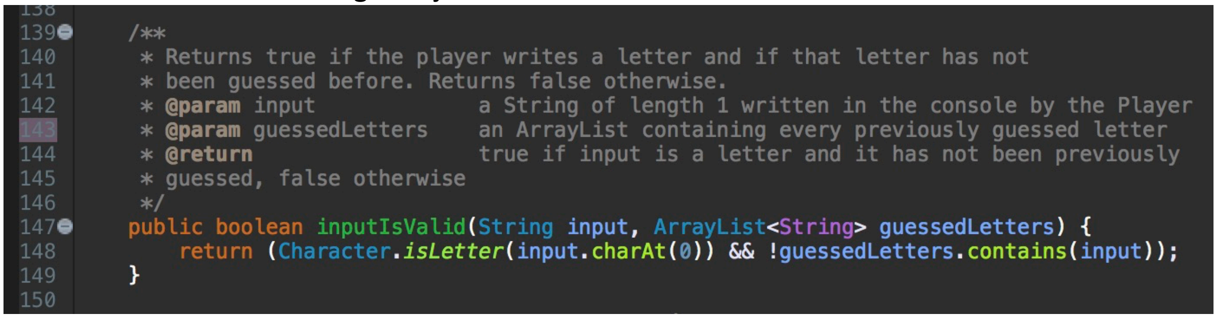
Manual tests report

Manual tests	UC1	UC2	UC3	UC4
TC2.1	1/OK	1/OK	0	0
TC2.2	0	1/OK	0	0
COVERAGE & SUCCESS	1/OK	2/OK	0	0

Automated unit tests

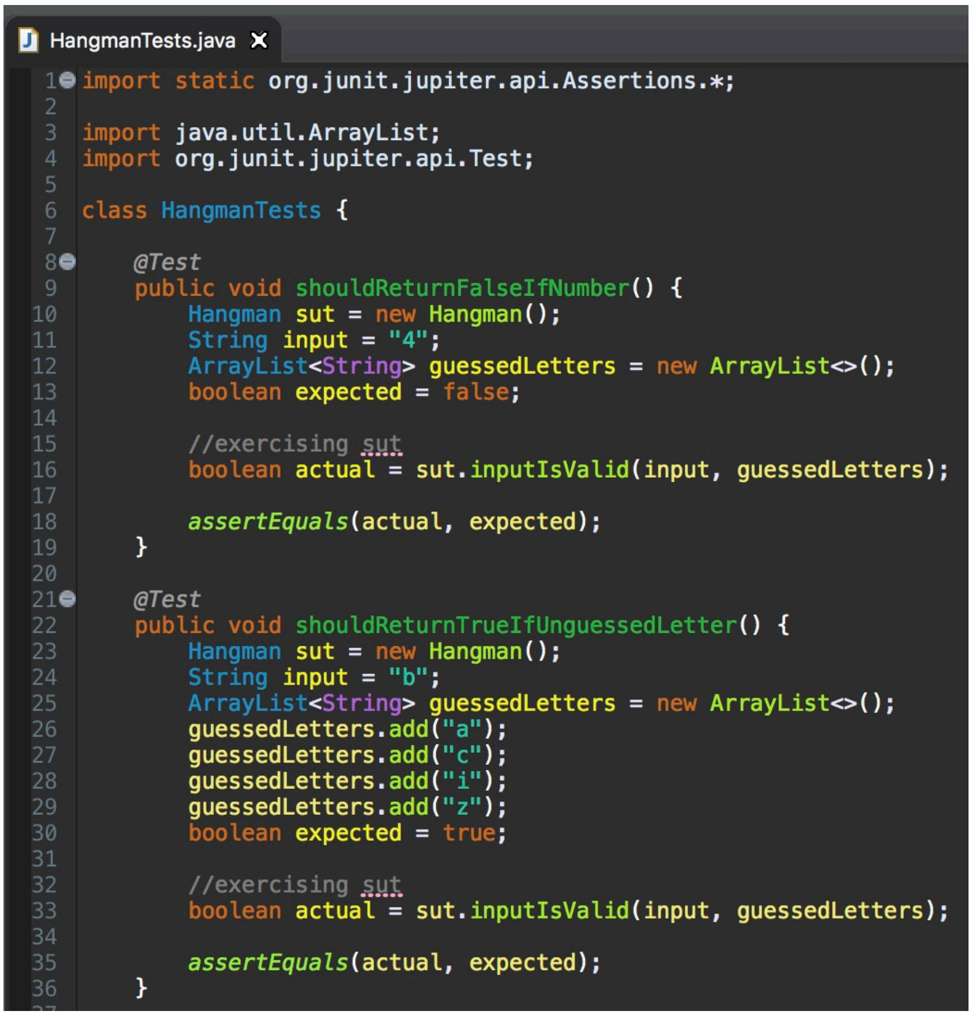
1. Testing method **inputIsValid(String input, ArrayList<String> guessedLetters)**:

Method declaration in Hangman.java:



```java
    /**
     * Returns true if the player writes a letter and if that letter has not
     * been guessed before. Returns false otherwise.
     * @param input           a String of length 1 written in the console by the Player
     * @param guessedLetters    an ArrayList containing every previously guessed letter
     * @return                true if input is a letter and it has not been previously
     * guessed, false otherwise
     */
    public boolean inputIsValid(String input, ArrayList<String> guessedLetters) {
        return (Character.isLetter(input.charAt(0)) && !guessedLetters.contains(input));
    }
```

Unit tests in JUnit:

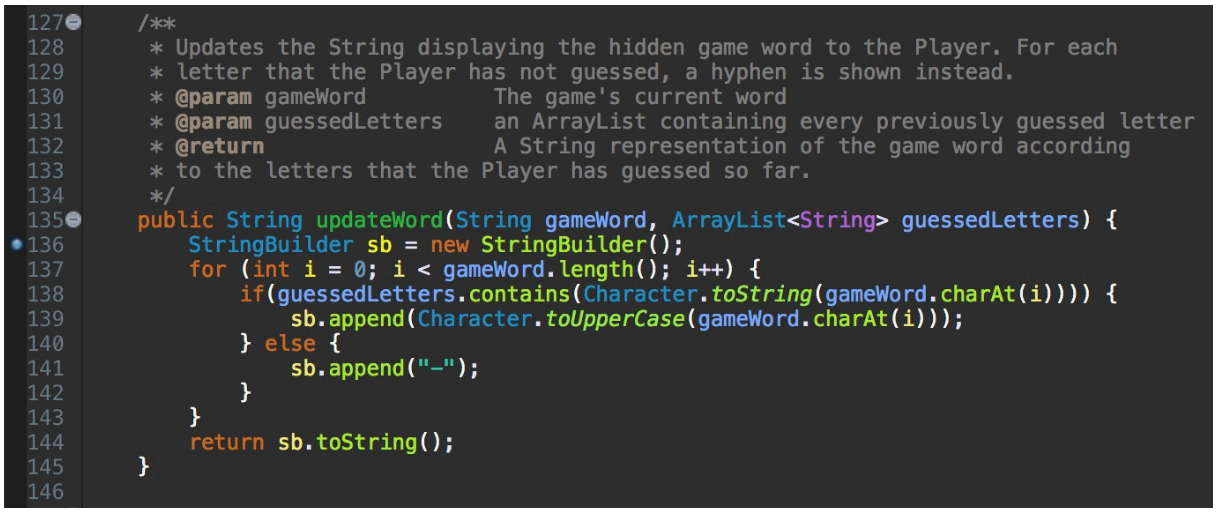


HangmanTests.java

```java
import static org.junit.jupiter.api.Assertions.*;

import java.util.ArrayList;
import org.junit.jupiter.api.Test;

class HangmanTests {

    @Test
    public void shouldReturnFalseIfNumber() {
        Hangman sut = new Hangman();
        String input = "4";
        ArrayList<String> guessedLetters = new ArrayList<>();
        boolean expected = false;

        //exercising sut
        boolean actual = sut.inputIsValid(input, guessedLetters);

        assertEquals(actual, expected);
    }

    @Test
    public void shouldReturnTrueIfUnguessedLetter() {
        Hangman sut = new Hangman();
        String input = "b";
        ArrayList<String> guessedLetters = new ArrayList<>();
        guessedLetters.add("a");
        guessedLetters.add("c");
        guessedLetters.add("i");
        guessedLetters.add("z");
        boolean expected = true;

        //exercising sut
        boolean actual = sut.inputIsValid(input, guessedLetters);

        assertEquals(actual, expected);
    }
```

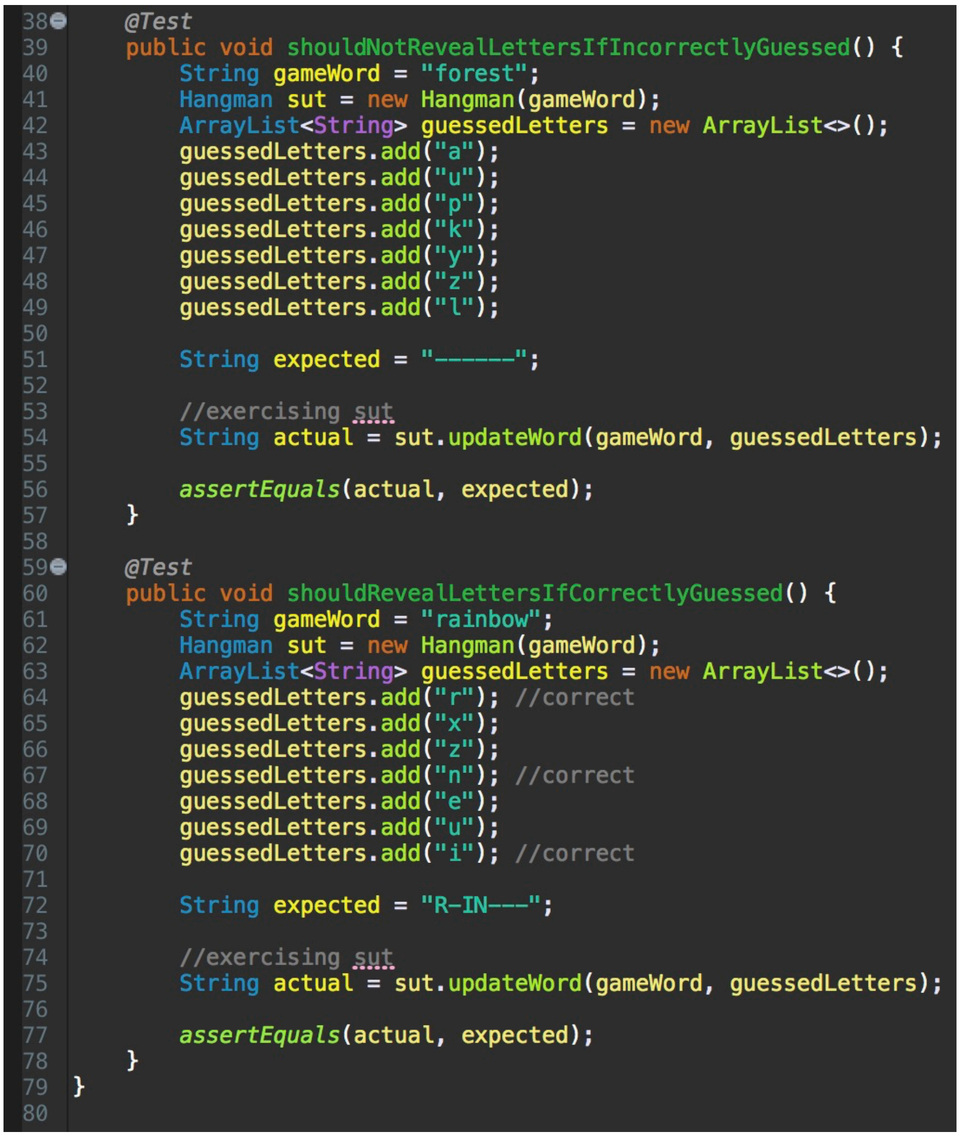
2. Testing method updateWord(String gameWord, ArrayList<String> guessedLetters):

Method declaration in Hangman.java:



```java
    /**
     * Updates the String displaying the hidden game word to the Player. For each
     * letter that the Player has not guessed, a hyphen is shown instead.
     * @param gameWord          The game's current word
     * @param guessedLetters    an ArrayList containing every previously guessed letter
     * @return                  A String representation of the game word according
     * to the letters that the Player has guessed so far.
     */
    public String updateWord(String gameWord, ArrayList<String> guessedLetters) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < gameWord.length(); i++) {
            if(guessedLetters.contains(Character.toString(gameWord.charAt(i)))) {
                sb.append(Character.toUpperCase(gameWord.charAt(i)));
            } else {
                sb.append("-");
            }
        }
        return sb.toString();
    }

```

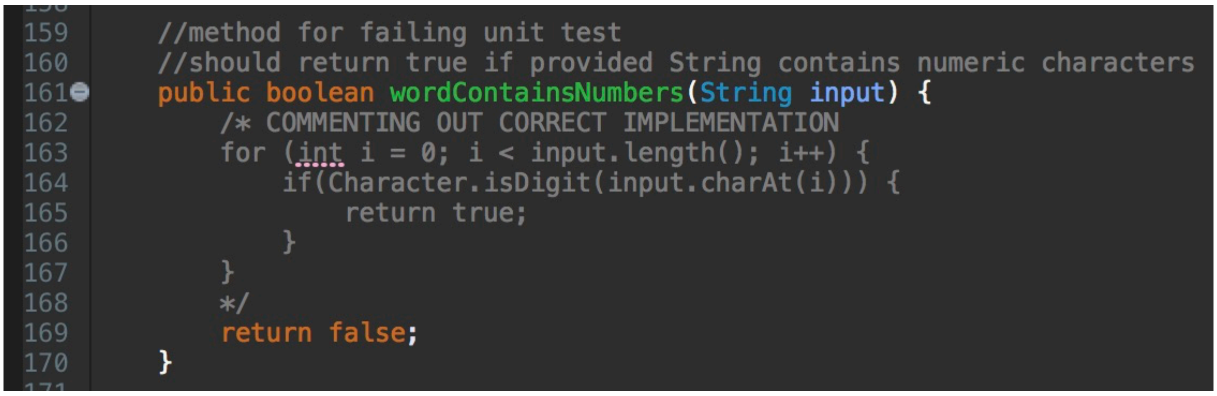
Unit tests in JUnit:



```java
    @Test
    public void shouldNotRevealLettersIfIncorrectlyGuessed() {
        String gameWord = "forest";
        Hangman sut = new Hangman(gameWord);
        ArrayList<String> guessedLetters = new ArrayList<>();
        guessedLetters.add("a");
        guessedLetters.add("u");
        guessedLetters.add("p");
        guessedLetters.add("k");
        guessedLetters.add("y");
        guessedLetters.add("z");
        guessedLetters.add("l");

        String expected = "------";

        //exercising sut
        String actual = sut.updateWord(gameWord, guessedLetters);

        assertEquals(actual, expected);
    }

    @Test
    public void shouldRevealLettersIfCorrectlyGuessed() {
        String gameWord = "rainbow";
        Hangman sut = new Hangman(gameWord);
        ArrayList<String> guessedLetters = new ArrayList<>();
        guessedLetters.add("r"); //correct
        guessedLetters.add("x");
        guessedLetters.add("z");
        guessedLetters.add("n"); //correct
        guessedLetters.add("e");
        guessedLetters.add("u");
        guessedLetters.add("i"); //correct

        String expected = "R-IN---";

        //exercising sut
        String actual = sut.updateWord(gameWord, guessedLetters);

        assertEquals(actual, expected);
    }
}

```

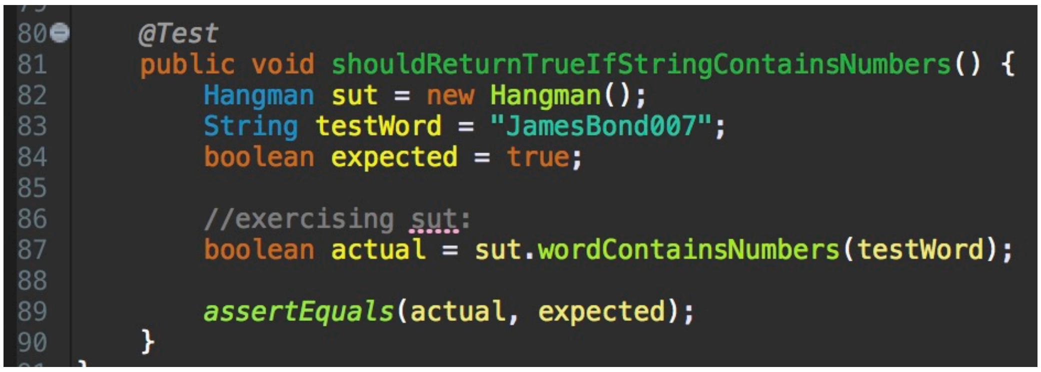
3. Failing unit test – method **wordContainsNumbers(String input)**

Method declaration in Hangman.java (complete implementation commented out as requested for assignment):



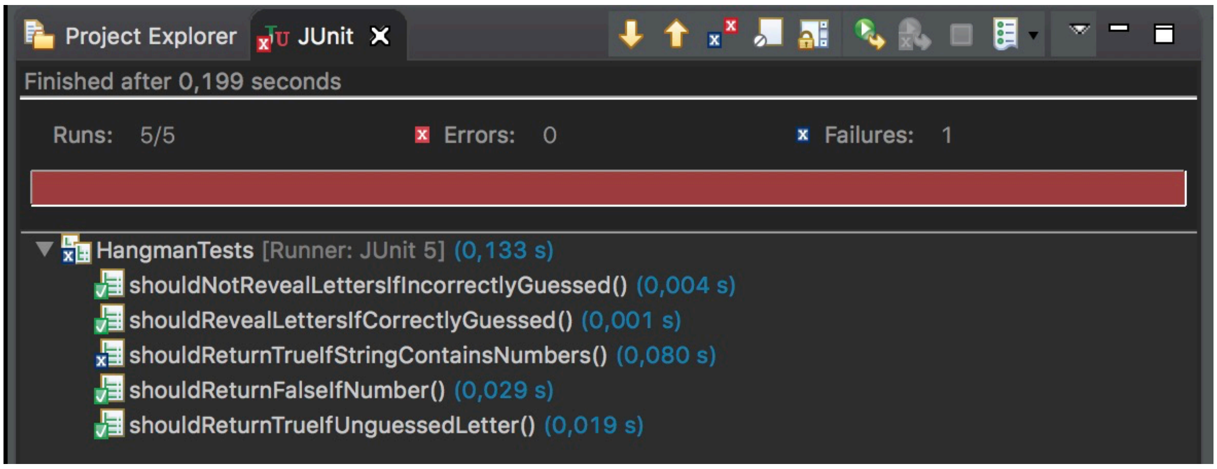
```
    //method for failing unit test
    //should return true if provided String contains numeric characters
    public boolean wordContainsNumbers(String input) {
        /* COMMENTING OUT CORRECT IMPLEMENTATION
        for (int i = 0; i < input.length(); i++) {
            if(Character.isDigit(input.charAt(i))) {
                return true;
            }
        }
        */
        return false;
    }
```

Unit test in JUnit (failing):



```
    @Test
    public void shouldReturnTrueIfStringContainsNumbers() {
        Hangman sut = new Hangman();
        String testWord = "JamesBond007";
        boolean expected = true;

        //exercising sut:
        boolean actual = sut.wordContainsNumbers(testWord);

        assertEquals(actual, expected);
    }
```

4. Unit test results



```
Finished after 0,199 seconds

Runs:   5/5          Errors:   0          Failures:   1

HangmanTests [Runner: JUnit 5] (0,133 s)
    shouldNotRevealLettersIfIncorrectlyGuessed() (0,004 s)
    shouldRevealLettersIfCorrectlyGuessed() (0,001 s)
    shouldReturnTrueIfStringContainsNumbers() (0,080 s)
    shouldReturnFalseIfNumber() (0,029 s)
    shouldReturnTrueIfUnguessedLetter() (0,019 s)
```

Reflection:

Although we spent some time reflecting and discussing non-functional requirements such as ‘testability’ in previous assignments, it has only now become clear what this means. In the previous iteration, I was concerned only about the program’s execution. Then, it seemed perfectly fine – advantageous, even – to manipulate the game’s attributes (class variables) directly in each method’s body, not using any arguments to pass these values between different methods (because they had class scope). After trying to design automated tests for such methods, I now understand that this isn’t the correct way to do it, because it removes the possibility of assigning custom values to these methods while testing them. This realization required several modifications.

Two good examples of changes to enhance testability were performed on the two methods that were tested with automated unit tests. Neither of these methods received any parameters, because the values used in these methods were assigned to class variables and therefore could be accessed in the methods’ bodies. This removed the possibility to test these methods in isolation using predefined values. I changed their implementations to receive these values as arguments and was then able to perform automated unit testing on these methods in JUnit.

I have no doubts that this insight will change the way I write code in the future, because it gave me a practical example of why it is desirable to pass values between method calls as arguments, independently of the scope of the variables that hold these values.