Worksheet 8: White-Box Testing and Test Fixtures (Java)

Updated: 27th May, 2021

There are two versions of this worksheet. This is the Java version.

In this worksheet, you'll practice white-box test design, and the setting-up and tearing-down of test fixtures. There are a range of production code methods for you to implement test code for. The code is shown below, and is also available in `Utils.java`.

For each production code method, do the following:

(a) Design your test cases:

- Identify the paths through the production code.
- Select test data for each test case. In other words, for each path, select inputs (parameters, console input, and/or input files) that will cause the production code to follow that path.
- For each test case, determine the expected results. This includes return values, exceptions thrown, console output, and output files.

(b) Implement your test cases.

It's good experience to continue to use JUnit, although you can still perform the exercise without it.

(c) Run your tests against the production code.

To ensure that your test code is working, it's helpful to temporarily *break* the production code. You could do this by editing the production code to alter the output/results very slightly.

1. `printCoordinates()`

`printCoordinates()` takes in x, y and z coordinates, and prints them out in the format (x, y, z), with two decimal places each. The output will end in a new line.

(This is a trivial case for test design. Since there are no conditional statements, there is only one path, and hence one test case.)

```java
public static void printCoordinates(double x, double y, double z)
{
    System.out.printf("(%.2f,%.2f,%.2f)\n", x, y, z);
}
```

Note: Remember to "tear down" everything that you set up. Specifically, to restore console output in Java, you must "save a copy" of the original, like this:

```
import java.io.*;
...
PrintStream originalOut = System.out; // Save original output
System.setOut(...);                   // Redirect output
...                                   // Do testing
System.setOut(originalOut);           // Restore output
```

To delete temporary files:

```
import java.io.*;
...
new File("somefile.txt").delete();
```

Warning: Be very careful with code that deletes files! Ensure the file you specify is definitely the one you want to delete.

2. `readChar()`

`readChar()` reads a single "valid" character from the user. The user enters a character, but must re-enter their input if it's invalid; i.e., if the character they enter does not occur within the `validChars` parameter.

Hint: there are two paths, and hence two test cases here.

```
public static char readChar(String validChars)
{
    Scanner sc = new Scanner(System.in);
    String line = sc.nextLine();
    while(line.length() != 1 || !validChars.contains(line))
    {
        line = sc.nextLine();
    }
    return line.charAt(0);
}
```

Note: Tearing-down console input redirection in Java is much the same as for output:

```
import java.io.*;
...
InputStream originalIn = System.in; // Save original input
System.setIn(...);                  // Redirect input
...                                 // Do testing
System.setIn(originalIn);           // Restore input
```

3. `guessingGame()`

`guessingGame()` runs a console-based number guessing game. The user is repeatedly asked to guess what the number is, and is told whether their guess is too high, too low, or correct (at which point the game ends).

Hint: the *result* of `guessingGame()` consists of all the console output, including the prompt asking for input. Also note that `println()` will print a newline character, while `print()` won't. This will be important when determining the expected output.

```java
public static void guessingGame(int number)
{
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter an integer: ");
    int guess = sc.nextInt();

    while(guess != number)
    {
        if(guess > number)
        {
            System.out.println("Too high.");
        }
        else
        {
            System.out.println("Too low.");
        }
        System.out.print("Enter an integer: ");
        guess = sc.nextInt();
    }
    System.out.println("Correct!");
}
```

4. `sumFile()`

`sumFile()`: opens a file (assumed to contain a list of numbers), and adds up the numbers, and returns the total. If the file could not be opened, it returns -1 instead.

Hint: the empty string `""` is an invalid filename that, by definition, cannot be opened.

```java
public static double sumFile(String filename)
{
    double sum = 0.0;
    try
    {
        Scanner sc = new Scanner(new File(filename)); // Throws IOException
        while(sc.hasNextDouble())
        {
            sum += sc.nextDouble();
        }
    }
    catch(IOException e)
    {
        sum = -1.0;
    }
    return sum;
}
```

Hint 2: your test code will need to create the file that `sumFile()` then reads. When doing this, the Java compiler will likely complain that you need to handle "`IOException`". You can do this as follows:

```
// If using JUnit
@Test
public void testSumFile() throws IOException
{
    ... // Your test code
}

// If not using JUnit
public static void testSumFile()
{
    try
    {
        ... // Your test code
    }
    catch(IOException e)
    {
        assert false;
    }
}
```

5. `saveData()`

`saveData()` opens a given file, and writes several lines of text to it. Returns `true` if the file could be written, and `false` if not (e.g., because an invalid name was given).

```
public static boolean saveData(String filename, String name,
                               int health, int score)
{
    boolean success = true;
    try(PrintWriter writer = new PrintWriter(filename)) // Throws IOException
    {
        writer.println("name: " + name);
        writer.println("health: " + health);
        writer.println("score: " + score);
        if(health <= 0.0)
        {
            writer.println("status: dead");
        }
        else
        {
            if(score >= 100)
            {
                writer.println("status: won");
            }
            else
            {
                writer.println("status: alive");
            }
        }
    }
    catch(IOException e)
    {
        success = false;
    }
    return success;
}
```

Hint: use `Scanner` to help verify that the file was written correctly; e.g.:

```
import java.util.Scanner;
...
Scanner sc = new Scanner(new File("outfile.txt"));
assertEquals("name: myname", sc.nextLine());
... // More assertions
```

Note: You may not have seen anything that looks like this before:

`try(PrintWriter writer = new PrintWriter(filename)) { ... }`

For the purposes of this worksheet, you don't need to understand the significance of this, *except* that:

- It's still a `try-catch` statement.
- This line of code is still considered to be "inside" the `try` block.
- So, if it throws an `IOException`, the code will jump to "`catch(IOException e)`".

End of Worksheet