Assignment 3 - Code Review & White Box Unit Testing

Version: March 23, 2023

Objectives

- Understand how to review code and to integrate reviews into source code control.
- Understand white-box testing
- Understand the various forms of code coverage
- Practice best practices in white box unit testing in JUnit and the EclEmma plugin.

Setup

You will need a few things for this assignment:

- The EclEmma Eclipse plug-in (either in Gradle or Eclipse both will be explained – other IDEs will not work but Gradle is enough)
- You will maintain a separate documentation file.
- This assignment is meant to be used with JUnit 4. For this assignment I expect you to stick with this version. The Gradle file included is meant for JUnit 4. For your project you can choose if you want to use JUnit 4 or 5. I stick with JUnit 4 since I think it is a bit easier to get started, especially in Gradle.

In this assignment we will look into Code Reviews and revisit Unit Tests but now as Whitebox test.
I want well written test cases here, meaning they should be named in a good way, have a description, test something specific so when the test fails it is easy to see what went wrong. This will lead to a bunch of test cases but each of them would be rather small. You will be graded on this!
The master and Blackbox branches DO NOT CHANGE in this assignment!

Task 1: Perform Code Review (10 points)

1. For this assignment we will work with the same code as in assignment 2.
2. Create a new branch based on the "Blackbox" branch (not based off of master) called "Review".
3. On Canvas you will find a review form, your task is to create a markdown file which is based on this Word document (check the DeliverableX.md document you will already see how to create a table). It has to have the base information from the review form, it does not have to be fancy, keep it simple. Add this form to your repository.

4. Review the source code files (any and all of them) for conformance to the defects provided in the coding standards document (Coding standards, for general quality practices - Code Smells, and for logic errors). Log any defects using the review form you created. You (individually) should at least log 7 defects in the given log file. Be sure to provide the information in that form (ID, Location, Problem description Category (CS, CG, FD or MD - defects that do not fit in the other categories). You should log at least one defect for each of the first three categories (CS, CG, FD). It does not count as different defects if you state CS 1 a couple of times (file header). Make sure you find different errors and do not list the same ones over and over.

5. When you are done commit and push your review form to GitHub

Task 2: Fixing the defects (10 points)

1. Select 3 defects from your review form from Task 1 (they have to be from different categories). And create an an issue on GitHub of each defect. Name them defect-# where #s are the number identifiers of the defects on your code review form. Read more on issues: Creating GitHub issues.

2. Pick one defect and resolve it locally on your computer.
   a) Commit the changes with a good commit message naming the defect id, a description, and reference the GitHub issue.
   b) Push the changes to your remote repository (to the Review branch).

3. Repeat step 2 for the second and third bug.

4. If you used the closing keyword correctly and if we would have worked on the default branch (master) then the issue would close on GitHub, since we are not doing that your issues will stay open (which is fine). We will in the next assignment (assignment 4) merge everything together and then your issues should close.

Now you should have 3 branches: master (unchanged), Blackbox with your assignment 2 tests and Review with your review changes.

Task 3: White-box testing (with some black box) - 30 Points

Create a new branch, base this branch off of your "Blackbox" branch and name it "White-box".

Step 1: Implement dealDamage(...), takeDamage(...), and attack() in GamePlay.java

You tested these methods in detail in your last assignment. Now is the time to do Test Driven Development to implement these methods yourself. Do the following:

1. Create a **new** test class and call it TDD.java . Into this class I want you to put all the test cases that you implemented for the Blackbox assignment but instead of testing all the .class files you need to adjust it in a way so it is for the the methods in the original GamePlay.java file (so all this complicated classes under tests is not needed anymore).

2. Delete the file GivenBlackBox.java test, we do not need it anymore at the moment and we want to make sure our build is successful.

3. Now, implement the three methods until all your tests in TDD.java pass.

Take a screenshot at the end of your passing test cases and put it in your PDF. If you cannot make all the tests pass then either your implementation is wrong or your test cases are. Fix whatever is wrong.

Step 2: Whitebox test GamePlay.java

Your task is to test the GamePlay.java class. You will see that some methods have some specification of what they should do included (this gives you an idea of what the method is supposed to do - so some black box information).

1. Look at the given WhiteBoxGiven.java test. It gives you an idea how you can test. It does not do that much though.

2. Draw a control flow graph for the play() method (before any changes are made). You can assume that calling a different method in it is just a node.

   a) Turn on line numbering in Eclipse (Window -> Preferences -> General -> Editors -> Text Editors)

   b) You may hand-draw these, take a photo, and import into your document. USE THE LINE NUMBER as the node label for each statement in your flow graph.

   c) List in your document which sequences you need to achieve for complete i) node coverage and ii) edge coverage.

3. Add all your test cases to WhiteBoxGiven.java.

   a) Write a test method for EACH INDIVIDUAL TEST SEQUENCE you identified in 2.c. Yes, some test sequences may address both node and edge coverage and so only need to be written once. Please put a header comment clearly indicating what test sequences the test method addresses.

   b) In case you find any errors in the algorithm - correct them and add a comment to clearly mark what you changed. Also add a comment why you think this needed changing.

4. Also test the functionality based on the description of the method to make sure it works as it should. In case you find any errors in the algorithm - correct them and add a comment to clearly mark what you changed. Also add a comment why you think this needed changing.

5. You will see more methods in that class. Create test cases for these as well (no node graph needed) and try to find if they work as intended (specified). Fix them (with marking your changes as before) if needed. Your methods might also call other methods, these should then be tested as well to make sure the code works correctly.

6. All your tests in this test class should pass at the end and you should write test cases until you are confident the code is tested enough. Explain why you think your test cases cover everything important.

Task 4: Gradle - 5 Points

This is basically the same as in assignment 2 but now there will actually be some worth while code coverage. Leave the Gradle file as as (it should still be the original one).

Step 1: You already have a Gradle file in your project. Run "gradle build" and then "gradle test". Look at all the things that are in the build folder. Especially what you find in the reports folder.

Step 2: The Gradle file already includes Jacoco. Jacoco is part of the EclEmma Plugin and can also give you information about code coverage. Your test results, which were created by ”gradle test” showed you the tests you have run and if they were successful. Now we also want to see the code coverage (which will not be a lot yet).
Now run "gradle jacocoTestReport". This should generate an HTML test report informing you about the code coverage, find the folder where this report is included in your build directory.
Take a screenshot of your command window, your file structure and your name showing somewhere on the screen (or asurite or something to identify you). Include this screenshot in your document. Put this under Task 4.
Step 3: Now do the following:

1. Answer: What is the overall code coverage for the whole Java source code (excluding Main)?
2. Answer: What is the code coverage for GamePlay.java with all your tests?
3. If it is not already, include more Unit Tests into your test class (WhiteBoxGiven.java) until you reach code coverage (node coverage) of at least 75% for the whole class, this might mean testing methods you have not tested yet.
4. Answer: What is the Code Coverage you were able to reach for the class as a whole? (only needed if you still needed to add tests)
5. Take a screenshot of your code coverage and paste it into your document.

Submission

On Canvas submit

1. GitHub link to your private repo
2. Your PDF document including
    - All answers and screenshots mentioned above.