Name: Gerardo Gonzalez.

Institution: Southern New Hampshire University.

Date: 2023/08/08 8:21 PM EST

Course: CS-320 Software Test Automation.

**Summary**.

The unit testing approach I have chosen is based on the little to no experience I have in software testing, I wanted to incorporate unit testing to ensure that the tests reflect accurate results that best align with the software requirements. Evidence can be found by running the assertions tests to confirm that the software methods resulted in functional intentions based on the requirements provided. My intentions behind my testing techniques were to provide the tests with dummy data and draw out any defects as shown in the example below.

@Test

public void testIdToLong() {

Assertions.assertThrows(IllegalArgumentException.class, ()-> {

new Task("T222001155665511", "Remodel", "remodel the Interior.");

});

}

To defend the overall quality of the Junit tests by running tests that cover as much code as I can within the files, with an 80% code coverage at a minimum. One observation I made was with the “Contact” and “Contact Service” tests that the code had an overall lower coverage but when you examined the units, you would notice that a large number of tests were written and compared to the “Task” project you could examine that fewer lines of test code were done. This shows with the second attempt at coding the coverage was improving by conducting fewer lines of code and learning more about the different ways code can be tested which was a sign of the tests were becoming more efficient.

Being able to cover unit tests with a larger percentage is efficient but I noticed that being able to acquire a larger percentage with fewer tests also shows the efficiency of the tests as it requires less code to execute and run tests while also acquiring the same test coverage.

One way that I ensured that my code was technically sound was by testing different aspects of one method by incorporating separate code to test both the acceptance test cases and rejection test cases such as assertion True and assertion False for the acceptance tests and Assertion Throws for the rejection’s cases. This way we can confirm that the methods being tested passed as intended with the correct inputs and failed correctly when the code experienced exception cases or the wrong inputs.

One way I made sure that the code I created was efficient was by creating smaller unit tests while also acquiring the same amount of code coverage when testing. In the following example, I created a test to make sure that the deletion code works correctly by adding approximates which can cover both the add function of the code and then deleting one of the tasks added, covering two portions of the code. Then is followed by an assertion throw test case to make sure that a none registered task id results in the appropriate assertion failure.

**Reflection**

For the testing techniques, I primarily used white-box testing for unit testing during all three milestones. For unit tests, I would create sections of code for each method and then create a unit test for that method. Each module would break up the file into chunks down to each method in the code. Within each method, I would create separate testing packages with unit test cases to test each unit and its methods. Each test would check the functionality, out-of-bounds scenarios, and exception handling to ensure that each module operates as intended. I performed all of this through manual testing or exploratory testing of the code.

Another testing techniques that I could have used are integration testing, which is the next level above unit testing. At this level, all the previously tested units would be combined to create a component, to test components for functionality. While I did not use this technique, it would require more time to study and learn how to implement it. There are also more advanced parametrized tests that could be used, involving different tags and methods in JUnit, such as @ValuesSources and @CsvSource. While I have learned about these techniques, I will explore in-depth how to apply them to tests. These techniques can simplify the test cases by creating file inputs and passing these as parameters to unit tests instead of manually creating each test variable within the test and then passing it on. This can result in less readable code.

The practical use that Each of these techniques has its benefits when used in tests. For unit tests and exploratory testing, you break down chunks of code and observe and learn how each unit works. Then you create tests to ensure that each unit fits the requirements and outputs the desired results. Integration testing helps test the previous units to ensure that they work and function as intended, by testing how each group of units interacts with other groups. To become more proficient at software testing, you can apply more advanced methods for each test with different annotations. These methods become more practical when testing different ranges within the test case and improve the test code by shortening it and making it more efficient.

When I looked at the project from a software testing perspective, I learned to be careful when creating tests for different parts of the code. There are many ways to write code, and it's important to understand how these methods interact with each other. For instance, a test that checks a contact ID needs to consider various aspects, like its length, uniqueness, and availability. Testing one part of the code can also impact other tests. For example, if you test adding a new contact, you should also check if the contact is added, ensuring that the contact ID is valid. This way of thinking makes you realize that test code needs to cover different parts of the codebase, making testing more complex, especially when you're new to it. This deeper understanding has been valuable for me as it helps me approach testing more effectively.

I've tried to reduce bias in my code by not just focusing on whether a test will pass. Instead, I found it more helpful to test the code for failures and extreme cases. For example, in many of my tests, I forgot to include a case where I check if an ID is already taken. I learned that considering both positive and negative scenarios is important. Tests should be approached similarly to Boolean methods – thinking about true or false conditions and whether the inputs satisfy those conditions.

Looking back on my experience of learning to test code, I realized that writing test code requires discipline and careful thought. Testing is more involved than writing regular code. It's crucial not to take shortcuts because each line of code can have a big impact on the final product. The success of a project can depend on how well it's tested before launch. To avoid accumulating technical debt in the future, I plan to practice testing on all my side projects. Becoming skilled at writing tests will allow me to meet various requirements, especially when there's limited time available for testing.