Tyler Hollingsworth

CS-320

08/14/22

**Summary and Reflections Report**

**Summary:**

My testing approach fully aligned with the software requirements because the unit tests were designed to ensure the correct functionality. In other words, I created a test for each requirement to not only test the passing case scenario but also the failure scenario. Testing based on the requirements is not only a way to verify that the code works properly but also that it aligns with the software requirements. For example, the following test case tests whether an error is thrown for an id that is too long according to the software requirement that an id can’t be longer than ten characters.

**@Test**

**void testTaskClassIdTooLong () {**

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

**new Task ("01234567890", "name", "description test");**

**});**

**}**

To ensure that I had adequate test coverage for my units I ran the program under coverage as a JUnit test and verified the percentage of my code and paths that were covered. To ensure that all the paths were covered in my code I wanted a 100% test coverage even though typically 80% or higher is acceptable on large enterprise-level projects. Creating and running JUnit tests that address both pass and fail scenarios for each requirement and ensuring that every path in all my units was tested ensured that my code was thoroughly verified against the requirements without a coverage gap.

I ensured my code was technically sound by first ensuring that I was using the correct and proper syntax. I then made sure my code was modularized into units making them easier to maintain and interchangeable. Lastly, I used the principles of object-oriented programming (OOP) by ensuring that encapsulation, abstraction, inheritance, and polymorphism were used when appropriate. An example is using encapsulation so that all the variables were set to either private or protected and accessible only through the programs’ methods.

**public class Task {**

**private static final int task\_Id\_Length = 10;**

**private static final int task\_Name\_Length = 20;**

**private static final int task\_Description\_Length = 50;**

**protected final String taskId;**

**protected String taskName;**

**protected String taskDescription.**

I ensured that my code was efficient by using the KISS principle or keep it simple stupid. This means that I wrote my code to accomplish each task in the simplest way without over-embellishing it. By coding in this way, I was able to create code that fulfilled the requirements and kept the risk of adding errors at a minimum. I also created test cases in this manner by testing pass/fail scenarios for each requirement in the smallest amount of code while still maintaining good test coverage. For example, the following code was used after every test case to ensure that the next test case started with an empty list and prevented the need for this one block of code to be rewritten on each test case.

**@AfterEach**

**void tearDown () {**

**if (listOfTasks.size () != 0) {**

**int index = 0;**

**while (index < listOfTasks.size()) {**

**listOfTasks.remove(listOfTasks.get(index));**

**System.out.print("Task deleted! \n");**

**}**

**}**

**}**

**Reflection:**

The main software testing techniques that I employed for my three milestones were white and black box testing. I used black box testing by performing unit tests on my code that were based on the requirements outlined in the specification. This allowed me to test the individual units to ensure the functionality of those units met the requirements. I also used white box testing by testing my code against its structure or the internal workings of the individual units. This allowed me to enhance the test coverage by verifying that all possible paths were tested. Finally, we were able to test valid and invalid parameters by using boundary testing techniques to verify that valid parameters give the correct output and invalid parameters throw errors.

Some software testing techniques that I did not use for my three milestones were experience-based testing. Error guessing is one of the methods that I chose not to use which is where the tester guesses where a weakness in the system may arise based on experience. The second experience-based testing method I chose not to use was exploratory testing which is when the tester creates tests parallel with execution and has the intent of maximum test coverage. The primary reason that I excluded both testing techniques is that they are based on extensive experience with testing such that the tester can recognize problem areas instinctively.

The three primary testing techniques mentioned above are white box, black box, and experience-based testing. White box testing is great for development projects where the internal components and structures of the units are known in advance so that testing can be performed to cover all the different paths and states of the individual units. Black box testing is great for development projects where the tests need to be focused on specification sheets and requirements to ensure that the software meets the needs of the user. Lastly, experience-based testing is great for development projects where experienced testers and those that thoroughly understand the software can implement tests on areas of the code that they believe may be an issue based on experience. The best scenario however is to use all three of these testing techniques in development projects so that the software is tested from all possible angles that may result in an error with the final deliverable product.

The mindset that I adopted on this project acting as a software tester is one where my primary goal was to find errors that may cause failures in the system and to essentially try to break the system. This was a difficult mindset to take on as I was also the developer of the system and the act of flipping a switch from the creation of sound and efficient code to trying to break the system competed with one another. The biggest caution in performing these dual roles was not to put blinders on as the tester thinking that the code I developed had no errors or bugs. The reason this was my biggest caution was due to bias that may arise in the testing practices such as going into the testing procedure and believing the classes were well-polished final products without errors or bugs. Incorporating these types of bias into the tester role means that the testing procedures to ensure that the system both functions properly and meets the requirements will be done with minimum effort and thus result in a lower quality final deliverable product.

As a software engineer, it is also highly critical to remain professional and always disciplined with every product that is created. When software engineers cut corners in writing and testing code, they are not only taking monetary risks but also risking many lives in the process. An example of a monetary loss includes the Y2K bug which cost companies and governments across the world billions of dollars to update their systems to include a four-digit date instead of the previously used two-digit date. An example of situations where software bugs resulted in the loss of lives is the patriot missile failure which resulted in the loss of 28 American soldiers due to a bug that resulted in a failure to track and intercept an Iraqi scud missile. These are only two examples among many others throughout history that show the importance of why software engineers have a moral obligation to remain professional and disciplined with every single line of code that they both create and test. Ultimately, I plan to avoid technical debt by remembering these situations as well as many others so that I am constantly reminded of the risks present in writing and testing code. This will allow me to remain professional and disciplined on every project and serve as a reminder to never cut corners under any circumstances or pressures.

**References**

Arnold, D. (2000). The Patriot Missile Failure. Retrieved August 11, 2022, from https://www-users.cse.umn.edu/~arnold/disasters/patriot.html

Rutledge, K., McDaniel, M., Teng, S., Hall, H., Ramroop, T., Sprout, E., Hunt, J., Boudreau, D., & Costa, H. (2020). *Y2K Bug*. National Geographic Society. Retrieved August 10, 2022, from https://education.nationalgeographic.org/resource/Y2K-bug