Ryan Reames

CS-320-T3331

2/18/2023

Project Two - Summary and Reflections Report

To test code for Project One, I structured my JUnit tests to follow along with software requirements. In the contact, task, and appointment classes there were specific requirements for the length of different attributes. Per the tests I had submitted, one can review and find test cases for various attributes in which data exceeding the length restrictions was provided. Additionally, many fields also had a requirement to not be null, which I tested for in all cases aside from unique IDs which I had set as a private final string which could not be changed once set.

Although near-100% coverage percentage by no means equals perfectly functioning code, I made certain that each line of code was covered by some form of test when possible. To do so, I passed both correct and incorrect input in tests to make sure that the areas being tested were covered for both when they should be working and when they should be throwing errors. I also made sure that in instances where there were multiple valid / invalid criteria (primarily in if statements), the tests were structured so every branch of the code was covered. As a result, all classes were tested with coverage exceeding 90%. Between this and testing for specific code requirements, I was able to ensure that the code was technically sound, either accepting valid input or rejecting invalid input when it should. For example, when testing unique IDs, tests were structured to create an ID, check if it was assigned, delete it, then check if it was deleted by trying to call the deleted ID back. This specific test was in all three service test classes I created, labeled as testDelete(Class). By combining multiple criteria when covering input (ex. phone input based on length, digits, and null value), I kept the code fairly simple and easy to follow. For example, when setting a phone number in the contact class, I set criteria to reject any null values, values greater than a length of 10, and input that did not match digits only. Overall, I was able to ensure the tests used were efficient by keeping test cases consistent and easy to reuse with minor modifications, and results were easy to follow with simple test names and criteria.

In Project One, I utilized both black box and white box testing techniques. The primary black box technique I used was boundary analysis, in which I would test input at the limits of what was required and see if the program I developed would correctly throw an error. For the contact, task, and appointment service classes I created, I tested each method based on the requirements provided. In general, each method needed to accept a value up to a certain length and not allow null values. I tested this passing correct data (within the length limit) as well as data that would exceed the limit by one character.

The primary white box technique I used while testing was statement testing, in which I would provide input and ensure that the expected outcome was met. For example, in each of the classes I was supposed to set up unique IDs. To test this, I created an instance of the class, tried to create another with the same ID (checking that it was not accepted), and then deleted the ID. Once the ID was deleted, I checked that requesting data from the deleted ID would throw an error. Essentially, I was trying to determine if the program would end with the expected results.

A few other techniques I did not employ in Project One were decision tables, use cases, and experience-based testing. Given the size of the program being developed, testing was straightforward to make sure that inputs met requirements, and did not require a lot of exploratory testing, charts to analyze, or use cases beyond the requirements given in the prompts. Regardless, the techniques I used could be implemented when developing different software, as there will always be requirements of software that need to be met (after all, if the project has no end-goal, it is likely not being developed for clients), and making sure requirements are met with boundary analysis or statement testing will nearly always be appropriate.

In testing Project One, I did my best to adopt a growth mindset, “where [I] believe the abilities [I’m] born with are a starting point for learning and advancing,” (Page, 2021). As this was the first time I tested software using unit tests, I tried my best to be cautious with my tests, and reviewed results to really understand why things were (or weren’t) working. For example, I found that when testing variables as being null, I had to use the “==” operator rather than using “variable.equals(null)”. Had I not tested the code, it would’ve compiled and run without errors, but functionally would not have been working correctly.

To limit bias while reviewing my code, I did my best to look at actual output vs. expected output, using a variety of tests. It would be easy to pass one or two tests and claim the software works, but to be thorough I was sure to test each potential use case I could. For example, each attribute for the contact class was tested for getters, setters, and creation, as well as specific requirements such as length, null values, or input properties (particularly for the phone number). Of course, there is the risk that testing my own software could introduce bias, particularly if I think that everything works as it should, but that is why a variety of testing techniques (black box, white box, experience-based, use cases, etc.) are used to gain a thorough understanding of whether the code is doing what it should.

It almost goes without saying, but testing code is always going to be a priority as a software engineer. It’s one thing to assume that code works correctly, but another to test the code and see what actually works (such as my example of using “==” vs “.equals()”). Cutting corners can lead to poor results, ranging from minor inconveniences to major disasters, such as losing $460 million in just 45 minutes (Lynch, 2017). Moving forward, I will be sure to test my code thoroughly to avoid defects and adding technical debt to projects I work on.

References

Lynch, J. (2017, September 14). *Software Bugs in History: Losing $460m in 45 minutes*. BugSnag. Retrieved February 18, 2023, from https://www.bugsnag.com/blog/bug-day-460m-loss

Page, A. (2021, March 2). *The Testing Mindset Myth*. TestProject. Retrieved February 18, 2023, from https://blog.testproject.io/2021/01/19/the-testing-mindset-myth/