Becky Scott

CS 320

Project 2

6/5/2022

For my unit testing approach, I found it was easiest to start with the system requirements and turn them all into unit tests. I then test a case where the requirement would not be met, and make sure the code will error out. For example, one of the requirements was that the name should not be longer than 10 characters. I created a test case to test whether a name of 11 characters would execute or whether an error would be thrown. In this case, the error was thrown as expected, and thus my code met the requirement.

During milestone 3, I was unaware of the ability to see the utilization of the testing of your code using coverage percentage. The quality of my coding for milestone 4 is vastly superior to milestone 3, because my testing coverage reached 90%. Even though I did not test milestone 3 using the coverage percentage method, I retroactively ran the test percentage, and the coverage was at 86% with only a few if statements left to be tested.

To ensure my code was technically sound, I walked through my execution line-by-line prior to writing any of my unit tests. Doing this, I was able to weed out logic errors. Once I wrote my unit tests, I did identify some errors that made it through my code analysis. For example, when I wrote my unit test to test the functionality of adding a new contact to the list, I discovered that I had a logic error in my for loop that was only comparing the first value in the list. My unit test was failing unexpectedly, and I walked through the code execution to discover the defect.

One of the ways that I demonstrated the efficiency in my unit tests was in my contactservice test example. I identified the common statements used in all unit tests and moved it up to the @beforeeach section ahead of all the tests. The @beforeeach tag will run the instance before each test is executed to reduce redundancy. I also coded the contact and contactservice code efficiently by pulling validations into their own functions to promote readability. For example, in the contact class, the checks that the name is not null and that it is less than or equal to 10 characters is housed within the function setFirstName.

In the past 3 modules, I’ve applied a variety of different testing techniques using both black-box and white-box techniques. Because I had access to the project specifications, specifically the client requirements, I was able to build some test cases using these specifications. Specifically, I created test cases using equivalence partitioning based on the client specifications (Hambling et al., 2015). For example, in the case of module 3, the contact class should contain a firstName string that cannot be longer than 10 characters. The two equivalence partitions that I created test cases for were a firstName value containing less than 10 characters, and a firstName value containing 10 or more characters.

Since I also had access to the code, I was able to use white-box techniques to test the solution as well. The main white-box technique that I utilized in my testing was using the testing coverage analysis in JUnit. Using the coverage analysis, I was able to see when some branches of my if-then logic were not being tested by my test cases. I then modified and added some test cases to ensure that I had 100% or close to 100% for all of the classes that were being tested.

There were a number of black-box and white-box testing techniques that I considered but ultimately did not employ in my testing. I chose not to employ these due to the relevance to the specifications or the difficulty to implement and the difference in test quality. Boundary value analysis was not very relevant for much of the specifications, because there were not a lot of specifications of the specific bounds. I did use this testing technique when it came to the date in module 5, however for the most part I was testing whether a field had too many digits. I would have used boundary testing far more if there were specifications like “There should not be more than 500 contacts stored in the contact service class”. Decision table testing seemed like it would be much better suited for systems using business rules and various inputs and outputs (Hambling et al., 2015). This testing technique would be useful if we were passing our contact, task, and appointment services into a larger system which acted upon them using some business logic.

While I did read through my code line-by-line and consider how the code logic would execute, I did not create a separate flow chart or control flow diagram for my code. If the code were more complex and harder to walk through in my head, I would have employed this technique to build out test cases to make sure all my paths were being hit. I could see flow charts and control flow diagrams being very useful in large system implementations with several control execution paths. Without such documentation, it would be very difficult and time consuming for the tester to understand how the code is executing, and thus doesn’t allow the tester to effectively create test cases to ensure proper testing coverage.

While crafting the unit tests for the module assignments and project, I was especially mindful of how the entire solution worked, and cautious to not let myself get too tunnel visioned on a particular class. Understanding how the classes work together was a key part of designing test cases, and failure to design them effectively given this knowledge can lead to some false positives that might lead to an insufficient software product. For example, one test case that I was crafting for the task service class was to see whether a duplicate ID being added to the service would throw an exception. If I was testing at this surface level, I might be happy with a test case that throws an error on the 2nd ID added. However, using my knowledge of the system, I also know that there are some IDs that will throw an error on the task class, which will throw a false positive for the task service class. I had to be mindful of these interactions when creating my test cases to ensure the best product was being created.

Through the development lifecycle, as both a developer and as a tester, I had to employ caution to make sure I was developing a well-designed solution. It is easy to fall into the mindset that because my unit tests all passed as I expected, that the solution is working as intended. Since I was acting as both a tester and a developer, I could fall into the trap of introducing some bias into the development to make sure that my code was passing the test cases. This might not be the best way to develop the solution, and may lead to inefficiencies and insecure code. I could try to tailor my code to make sure that the test cases pass, rather than trying to develop the best solution possible. This is even more true if I’m designing test cases and coding to make sure they pass, and not looking at the needs of the customer and the requirements. Per Hamilton, “It is possible that software which is 99% bug-free is still unusable. This can be the case if the system is tested thoroughly for the wrong requirement.” For example, on the checking of the phone number in the contact class, I could have written in a series of if/then statements. These statements would satisfy the unit tests, however it would make the code more difficult to read, less efficient, and possibly open me up to more defects that may not be totally covered by the unit testing.

The practice of coding inefficiently with the goal of passing testing rather than delivering efficient code can lead to some costly mistakes and poor products. The code produced by this method would be difficult to read, have high amounts of technical debt, and possibly contain a larger amount of defects. I think the goal of any developer should be to focus on the effective design and implementation of the solution to meet the requirements of the client from the onset. As testing occurs and some defects are identified, the developer should use the totality of their understanding of the product to craft a solution that still aligns with the requirements and also provides an elegant efficient solution that can be built upon by future updates. It can be tempting for developers to patch together a short term focused solution to ensure the testing passes, but raises the technical debt and makes future iterations more difficult to implement. An example would be to make changes to the task class constructor, adding several if/then statements to pass the specific unit tests I designed. The solution I ended up going with was using a separate set variable function that handles the validation and the setting together. This solution made the code easier to read, and allows me to add additional validations easily in the future, should the requirements of the code change.

References

Hambling, B., Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). *Software testing an ISTQB-BCS certified tester foundation guide*. Swindon: BCS, The Chartered Institute for IT.

Hamilton, T. (2022, April 16). 7 principles of software testing: Learn with examples. Retrieved June 18, 2022, from https://www.guru99.com/software-testing-seven-principles.html