Brian Hartong

Professor Pettit

CS-320

16 October 2022

Project Two

The extent that my testing approach was aligned to the software requirements was that I took the requirements given and made sure that I had a test that followed what each requirement was. The first requirement that was given was for Task objects to require a unique task ID that cannot be larger than 10 characters. Given this information, I created a blank function that would test for an ID that was more than 10 characters, that I then used later to fill in code to test that requirement.

I knew that my JUnit tests were effective by checking on the percentage of coverage of the class I was testing. This gave me insight into what parts of the class did not have a test. Since my tests were able to get a 100% coverage of the class I was testing, I can say that my tests were effective at making sure all requirements of the class were covered.

To ensure that my code was technically sound, I made sure to follow the class and variable names given through the requirements. This would allow other developers to work on the code and understand what each part is supposed to do. To make sure that the test code was technically sound, I made sure to leave comments providing information about what each of the

tests should be doing, as seen in the screenshot of my code below.

Text

Description automatically generated

To make sure that my code was efficient I made sure that each test only tested a single item. This allows for individual tests to be run testing only parts of the code that were updated without testing everything. Having specific tests to test different sections of code helps to speed up the testing process. You do not want to run a large test that checks for every single requirement for a task when you only updated a single section. An example of these specific tests is shown in the screenshot below.

Text

Description automatically generated

A software testing technique that I used was equivalence partitioning. This technique takes the test data and splits it into partitions of values. These values are determined to be valid or invalid data partitions. A single value is then selected from each partition and used for testing. Since each value in a partition would produce the same validity, using a single value allows for the validation of the data without testing every data point. An example of this from the milestones is the requirement of an ID being 10 characters or less. The test data can be split into character length of 1 to 10 and 11 and greater. You can then select a single value from each, for example, 5 characters and 15 characters, and use those values to determine if the code is working properly.

A software testing technique that I did not use is boundary value analysis. This technique uses testing values on the boundaries of the valid input criteria. The importance of this type of technique comes from the knowledge that boundary values are not normally handled properly. An example of this type of testing from the milestones would be the requirement that the description field must be 50 characters or less. The boundary values that should be tested would be, 0, 1, 50, and 51. Since both 0 and 51 characters are invalid inputs and 1 and 50 are valid, this properly tests the boundary values for the description field.

The Equivalence Partitioning testing technique is useful for projects where there are too many valid inputs to be able to test all valid and invalid options. Determining the range of values that are valid or invalid reduces the number of tests that need to be run, which in turn decreases the time and expense of running the tests. While this type of testing reduces the number of tests needed to cover most of the application, it does not validate all possible inputs. “…while partitioning helps us minimize our test cases to maximize coverage, we need to be aware that it doesn't cover all the combinations required to test the application successfully” (Sharma, 2022a).

Boundary Value Analysis is a useful testing technique when the valid inputs are numeric or sequential. Since many issues with code are the result of the improper handling of boundary values, testing these values becomes highly important. While boundary value analysis also reduces the number of tests that need to be run, like equivalence partitioning, it has a similar issue of not validating all possible inputs. “Boundary value and equivalence partitioning assume that the application will not allow you to enter any other characters or values. Such characters, like @ or negative values or even alphabets, will not be allowed to enter” (Sharma, 2022b). While this is adequate for some applications, more tests should still be run to determine if such data values function properly.

I employed caution when making sure I was testing for the right conditions. When writing tests, it is easy to make mistakes and not test for the correct conditions. As I wrote each test, I made sure to run the test to confirm I got the expected output. After reviewing my code, I should have made sure the edge cases were covered since a common mistake is when the code only allows “less than” a number when it should be “less than or equal to”. In my AppointmentTest file, when testing the creation of an appointment with an ID that was too long, I should have tested for a 10-character long ID along with the 11-character test I did. This would have tested the edge case by confirming that the limit of 10 characters was accepted. It is important to appreciate the interrelationships of the code that is being tested because while individual parts of the code may work correctly, testing them together may lead to an unexpected result.

During the review of my code, I limited my bias by trying to step back and write them as a person who has not seen the code. This helps by removing some of the assumptions that the code “just worked” and making sure that it did what was expected. As a developer writing their own tests, bias is a concern because you tend to assume that the code worked correctly and might forget to test all the possible inputs. Also testing your own code can lead to repeatedly taking the same steps to see if something is working correctly, while another person might do the steps slightly differently, causing the program to fail. In my review of my tests, I found that I made mistakes in assuming that the program deleted an appointment. I wrote a test case that made sure the delete call did not throw an exception, but I never checked through the list of appointments to verify that it was removed.

It is important not to cut corners when it comes to writing or testing code because it could lead to longer development time and higher development costs if issues are found later that could have been resolved early in the process. Making sure that the code you write is of good quality by using your own tests during development, even if you are not the one testing, will lead to easier fixing of issues later. For example, if you correct an error early in the process it could prevent you from having additional errors that built on the original one. I plan on avoiding technical debt by making sure I test my own code often. I also plan to write tests by creating shells of the program as placeholders and then writing the tests using the shells. This will help remove some of the bias of creating the code and tests by making sure not to assume the program did it correctly.

References

Sharma, L. (2022a, October 2). *Equivalence Partitioning - A Black Box Testing Technique*. TOOLSQA. Retrieved October 2, 2022, from https://www.toolsqa.com/software-testing/istqb/equivalence-partitioning/

Sharma, L. (2022b, October 2). *What is Boundary Value Analysis (BVA) of Black Box Testing Technique?* TOOLSQA. Retrieved October 2, 2022, from https://www.toolsqa.com/software-testing/istqb/boundary-value-analysis/