Alex Crosswhite

12/05/23

SNHU

CS 320

**Summary and Reflections Report**

In an attempt to properly meet the requirements of the software I created a test for each class and function. By testing all parts of the code’s functionality, I was able to ensure that each performed the way they were expected to.

For Contact.Java, I created a test for the results of the constructor and a separate test for the setters. I used Junit testing to ensure the constructor properly created the initial contact. I created a second test to prove that the getters reflected the given setters. For ContactServices.Java tests were created for adding, deleting, and updating contacts to validate that each function performed as expected.

For TaskServices.Java I created validation tests for ID, Name, and Description. The software requirements dictate that an ID cannot exceed 10 characters, a Name cannot exceed 20 characters, and a Description cannot exceed 50 characters. Additional tests were created to test the update and delete functionalities. The test for updating tasks checked to see that attempting to update a task with the incorrect ID would fail and a matching ID would properly update the Name and Description. For Task.Java I created a test to verify the creation of a new task. This test could only be passed if the tests for input validation for ID, Name, and Description were all passed first.

For Appointment.Java I created a test for the setters and getters to verify that an appointments ID number, date, and description could be properly created and accessed.

While each class and function are tested for the requirements of the software, it cannot be ignored that significantly more input validation and testing should be conducted to properly develop more precise input requirements and more secure code. Even though the code has a 100% Junit coverage, the code was not designed with input validations beyond the scope of the given requirements for the software, even ignoring some of the non-functional requirements that would be present in any actual software that would provide error handling.

In ContactServiceTest.Java line 26, three contacts are created, added to the cs ContactService, and a test is run to delete contact “75725”. Another test is run to delete a contact ID that was never created, and a final test is run to delete “75725” after it has already been deleted. These tests work to demonstrate that the code is technically sound and meets the software requirement to properly delete a contact.

A screen shot of a computer program

Description automatically generated

Perhaps my favorite function to work with in these projects was the addTask() function. This function exemplifies efficiency by calling on a series of functions to validate each element of a given task’s length and if all qualifications are met, adds the task to the ‘tasks’ TaskService. The test for this function uses the addTask() function to create a task, then each element of the new task is tested for the expected outcome.

A screen shot of a computer program

Description automatically generated

The techniques I chose to use for the various tests in each milestone were dependent on the purpose each was expected to serve. Some tests were designed to test potential user input to ensure that improper lengths would not create vulnerabilities. These tests are considered to be white box techniques; techniques that test the code through the use of test cases and access to the code itself.

Equivalence partitioning tests were used for both valid and invalid inputs to be sure the code is capable of handling both. Decision tables were used to test conditions. State transition tests were created to test changes in outputs. Boundary values were used to test limitations like maximum character counts for user input. The intention for these tests is not only to verify that the code produces the expected outcomes, but also to ensure that the code meets the requirements of the client. Statement, path, and branch coverage is determined by how much of the code is tested through use cases, input validation, and class and function output. Coverage is an important consideration when testing your code. The more testing that can be applied to a system, the more secure a system becomes and the greater the quality of the system overall.

Unfortunately, these milestones do not allow for the opportunity to work with experience-based techniques. These techniques draw on the experience of testers and users to pinpoint which areas require the most attention for additional testing. These areas are most commonly noticed while performing black box tests that are completed through interactions with system rather than by reading the code itself. Since these milestones were not accessed by any external users or testers, this was not an appropriate form of testing.

Error guessing is another technique that I did not use as it draws on past experiences to make educated determinations about which areas of the code may need the most focused testing. I could have used exploratory testing to further examine user inputs for my code to shore up additional vulnerabilities that exist. I chose not to do this as securing user inputs was outside the scope of the assignment past ensuring the length of the inputs and the validity of a future appointment date. Additional testing could have been implemented to limit character types and further restrict appropriate input types.

When developing code caution must be adhered to with each line of code. While it is not possible to predict the full impact of every line we write, it is still the duty of the author to put thought into how that line will interact with the rest of the system and the actors who may access it. In this project, caution was necessary when considering input validation, class creation, and test construction. Each were important in making the system align with the requirements of the client. Being cautious and thoughtful when choosing which tests to use helps to confirm that the code will behave as intended. The interactions that occur between functions can be a source of unintended outcomes if the code isn’t developed with all potential outcomes in mind, even if these outcomes only occur in outlier cases.

Bias is one of the most difficult obstacles to overcome due to its innate difficulty for an individual to detect in their own views. In programming these biases can be particularly dangerous when it comes to testing code. Whether its due to the fatigue of repeatedly looking through your own code, the limitations of the abilities of the programmer, or simply not having the necessary perspective to overcome the challenge at hand, bias can be the source of incomplete system testing. Limiting bias is no easy task and requires the program to be continually asking questions as they go. Thoroughly considering which tests are necessary for not only the security of their program but also the functionality of the system requirements can help to identify blindness’s created by bias. When possible, another set of eyes can be an invaluable tool for detecting weaknesses that may exist in your code.

Being disciplined in the commitment to quality often boils down to diligence and resilience. Combing through each part of the code to be sure it is well written before testing and that when testing done, the tests are well crafted. Writing the code with security and functionality in mind from the start increases the overall quality of the system and reduces the technical debt that may be accumulated by the need to continuously reiterate on the system. Cutting corners increases the technical debt accrued through the need for further testing and remediations. Proper testing can be applied to ensure the quality is there and to identify vulnerabilities, but in order to obtain useful results, the tests must be designed to detect those occurrences. To get the right answers, the right questions must be asked.

**Citations:**

Cypress Data Defense. (2020, June 25). *White Box and Black Box Testing: Outlining the Difference Between Them*. https://www.cypressdatadefense.com/blog/whitebox-and-black-box-testing/#:~:text=White%20box%20testing%20is%20done,testing%2C%20security%20testing%2C%20etc.

GeeksforGeeks. (2023, November 24). *Software testing techniques*. GeeksforGeeks. https://www.geeksforgeeks.org/software-testing-techniques/