7-2 Project Two: Summary and Reflections Report

Joshua Pugh

CS-320

02/25/2024

Understanding the requirements of the given software to be tested is important in ensuring the software will function as intended. My approach for testing the contact, task, and appointment service deliverables was to ensure each one met the minimum requirements first. For example, the deliverables required validation for the class attributes. For the contact class, the name was not allowed to be null and could not be greater than 10 characters. After coding the logic, I made a few test cases to ensure that the class would only accept a valid first and last name with the given restrictions. I then thought about other inputs that could be given and thought about the fact that a person's name should not be allowed to contain numbers. I also considered that the first and last name fields should not be allowed to be empty.

I then created some more methods within the class to handle these exceptions and then made some more test cases to ensure the class would only accept valid input with these exceptions now in mind. The three methods I created to handle these exceptions were, void verifyFirstName(String fName, void verifyLastName(String lName), boolean isAlpha(String string). These methods check to make sure that the names only contain letters, are not null or empty, and are at max only 10 characters in length. I then followed the same approach for each other class implementation.

My JUnit tests are effective based on the coverage percentage. When making my test cases I ensured that I achieved 100% code coverage within each class. This ensured that all the code was exercised and validated to ensure proper functionality. For example, when creating the test cases, I created a few tests first and then ran the tests to ensure it was providing the desired result. If it did not pass, I went back and changed things around while looking over the code to find the reason for the unexpected outcome. I also verified the test case was targeting the correct lines of codes and methods. Once most of the test cases were coded for each class, I then began to run all the tests at once with coverage to see how many lines of code were being exercised. This gave me a good indication of how many more test cases needed to be created to ensure 100% coverage for each class’s code and methods. Since each class has 100% code coverage, I know that all lines of code within each class have been tested to ensure proper functionality and that each class meets the requirements set forth.

To ensure that my code was technically sound, I made some test cases with invalid inputs to make sure the class would reject the input such as, when deleting a task or contact from the list, the list should not be empty, and if it is the class should throw an exception as seen in this line of code:

//Test with empty list

assertThrows(IllegalArgumentException.class, ()-> services.deleteTask(UniqueID));

The class indeed throws an exception if the list is empty. My thoughts behind this logic were simple, if the list is empty there is no need to search to remove something that does not exist. In the same way, I ensured the code was efficient by keeping things as simple as possible and not overcomplicating things. I made sure my function names were appropriate and only handled one task. For example, void test\_deleteContact\_With\_Empty\_list(). The purpose of this function is clear from the name. It tests the delete contact method with an empty list to ensure it will throw an exception. The same is true for the rest of my testing functions.

For each of the milestones, the software testing techniques that I employed were unit testing, functional testing, and code coverage analysis which are forms of white-box testing. Unit testing is a type of software testing that focuses on individual units or components of a software system. A unit is the smallest possible source code that can be tested, such as a function or a procedure. The purpose of this type of testing is to validate that each unit of the software works as intended and meets the requirements. Functional testing validates a software system against its requirements and specifications. It ensures that the application behaves as expected by providing appropriate input and verifying the output.

Code coverage analysis is a method that helps one to understand how much of the source code is being tested. Some other software testing techniques that I did not use for the milestones are non-functional testing, integration testing, and regression testing. Non-functional testing tests the non-functional requirements such as performance, security, and usability and focuses on aspects beyond functionality. Integration testing tests the integration of different components and verifies that they work together seamlessly. Regression testing tests the software after changes have been made and ensures that the new changes did not introduce any new defects.

There are many uses for each of these techniques. Unit testing can be used within any project to verify that each function or unit of software works as intended and can also help to catch bugs early. Functional testing can be used to ensure the software that is being developed meets the requirements and expectations of the end users or stakeholders. Non-functional testing can be used to check aspects beyond functionality such as the security, and usability of the system or software. This can help to ensure that the software is safe and secure to use. Integration testing can be used to verify the interactions and data exchange between different components or modules of a software application. Regression testing can be used to ensure no new defects have been introduced into the software after changes or updates have been made. All these techniques can be used in conjunction to ensure that the final product meets the expectations and requirements of the end users. They also help to provide a high-end quality product to the market.

My approach to testing was to ensure that the finished product met the requirements and functioned as intended. I also looked for ways to improve the code using the test cases which only made the final product stronger and more reliable. When writing my test cases, I was not merely trying to validate inputs or outputs but rather trying to make the software fail to find the weak links within the code. This allowed me to find ways to improve the overall quality and reliability of the code. I used caution to ensure I was keeping my test cases simple and to the point, only testing a few invalid inputs and valid inputs within each class. The complexity of the code can lead to over-testing if you are not careful. For example, if you are writing test cases for a complex algorithm that calculates current home values, trying to test every possible input could lead to overlooking other defects within the code due to over-testing.

Appreciating the complexity and interrelationships of the code you are testing allows you to identify potential issues, optimize performance, and ensure robustness. This is because, as you gain deeper insights into the code, you will become better equipped to write, maintain, and test high-quality software. Bias is something that one cannot merely remove from the equation when performing a task or doing a job but rather something one must recognize is present within the given situation. For instance, a carpenter who builds cabinets and thinks he has no bias toward his work will most likely think his cabinets are the best ones on the market. However, when his cabinets do not sell as well as the other carpenter down the road who recognizes his own bias toward his work, he might question what makes his cabinets better than mine. Knowing you have a bias toward something allows you to look at things through a different lens and from a different perspective. This allows you to look past your bias and ensure you are creating high-quality products or doing the best possible job you can do.

Being disciplined is an important aspect of any job. When it comes to software engineering, it is even more important to stay disciplined when developing software. Trying to cut corners to save a buck or a few minutes, could lead to a significant loss of time and money down the road. It is very possible and even likely that you will introduce defects by cutting corners that could lead to failures of the software. Depending on the type of software you are working on, this could put people's lives in danger. From my experience, the payoff for cutting corners is never worth it in the end as I have seen the damage that can be done when one is careless and not disciplined. It is wise to always strive to do things the right way the first time around.