Lukas Pentowski

CS320 – Software Testing & QA

Project Two – Summary and Reflections Report

Ensuring that the testing approach undertaken for this project aligned with the project requirements was an important part of the testing process. In order to ensure that my approach did align with the project requirements I had to verify what the project requirements were. Each module of the mobile application had specific requirements for the attribute fields that must be met, such as character limitations and field length, as well as functional requirements of which attributes could be modified.

Prior to writing my code I went through and read the requirements, making note of the cases that should cause a failure. After each class was complete, I created the JUnit test class for it. JUnit is a framework that was developed for use in Java unit testing. It assists in test-driven development by allowing pieces of code to be tested without full execution of the application. In order to complete the test class I went through each function of its corresponding class and wrote tests that best fit the requirements. An example of this would be the name field for the Task class. The requirements state that the name cannot be null or greater than 20 characters. In writing the constructor for the class I applied input validation to ensure that the field is neither null nor the length was greater than 20. In the corresponding test class, TaskTest, I wrote tests using JUnit to asserting that an error and exception was thrown if either condition were true, as well as verifying that a properly formatted name would be accepted.

When using coverage percentage to validate the effectiveness of my JUnit tests I feel that my tests more than adequately meet the requirements to be considered effective. My overall coverage for my JUnit tests was 81%. This includes both the Test units and the three required units of Contact at 95%, Appointment at 87%, and Task at 86%. Coverage percentage informs the tester what percentage of the code was executed through the tests written. In his article for Atlassian’s Blog, Sten Pettit comments that though there are no agreed upon values for coverage it is a good idea to aim for 80% coverage. (Pettit, n.d) The percentage of code that is not covered by my testing are branches of loops that do not execute. Additional tests could be written, but that will take additional time and resources and potentially may not be worthwhile. One of the foundations of testing is that testing cannot be totally exhaustive.

In order to ensure my code was technically sound I made sure that my code was well documented, followed accepted coding practices, and was testable. My code for each service has comments for the various functions explaining their purpose and the method in which they will achieve that purpose. In following accepted coding practices, I employed the DRY (Don’t Repeat Yourself) principle, proper white space, avoided deep nesting, and kept functions short and to the point. In each package I followed the DRY method by creating functions for both unique ID generation and to search the Array Lists created to store the items created. An example of my unique ID generator and its use in the addTask() function:

A screenshot of a computer program

Description automatically generated with medium confidence

Figure - Code for generating an unique ID

A screen shot of a computer program

Description automatically generated with low confidence

Figure - Code for addTask() utilizing the function to create an unique ID

Writing a function for the repetitive task of creating unique IDs or searching through Array Lists limits the chances of error, reduces lines of code required to be written, and improves readability of the code. I used input validation with both the Task class constructor and its Service classes for potential future security concerns with error logging.

The same methods to ensure my code was technically sound also ensure that my code is efficient. The use of stand-alone functions for repetitive tasks reduces the lines of code required. Writing the classes to the requirements document prevents any bloat of unnecessary functions. I used only the required variables for each class.

With software testing there are two main categories: static and dynamic. Static Testing, such as a manual code review, is performed without executing the application’s code. Dynamic Testing, such as Use Case Testing, executes the code and uses various types of input to test it. (GeeksForGeeks, 2023) In order to test the code written for this project I used a variety of testing methods from both categories.

Some of the Static Testing methods that I used to test my code were manual code review, code flow, Eclipse’s Compiler, and a code walkthrough with a classmate. With my manual code reviews, I would verify that conditionals met the requirements described by the rubric, that dynamic JUnit Tests were written correctly for the areas I wanted to test, and that code was simple and easy to read. Using Eclipse’s Compiler as a Static Testing method helped identify dead code, such as imported libraries that were not needed or variables that were unused and made a code flow analysis easier to complete. Lastly, doing a code walkthrough with a classmate helped me with verifying the logic of my expressions.

JUnit tests are a testing method that allows for both unit and integration testing. Unit and integration testing are two types of Dynamic Testing. When writing JUnit tests I tested the different methods of each class, with both positive and negative inputs. This allowed me to see if the code met the requirements stated in the rubric. I also tested to see if the Service classes calling on the corresponding base class worked and was able to set and get attributes as needed.

Unit tests, integration testing, code reviews, walkthroughs, compilers and code flow are the different testing methods that I used to perform both Static and Dynamic testing. Due to the size of the code written I felt that these were sufficient to ensure the code provided met the requirements listed in their rubrics. As the code progresses in complexity it may become necessary to use additional methods of testing

The mindset that I adopted while working on this project was one that embraced test-driven development. I reviewed requirements, determined what tests I would implement to verify those requirements, and wrote my code with those tests in mind. I would run the tests and coverage verification after any change was required, adding additional tests as necessary.

In order to reduce bias in my testing process I utilized code walkthroughs and automatic code reviews. Stepping back and receiving outside input allowed me to distance myself from the code I wrote and be open to corrections. It’s important to be unbiased in the testing process to ensure that industry best practices can be followed and that project requirements are met. There may be more efficient ways to write the software and complete the testing, bias may prevent you as the developer from achieving those ways and prevent professional growth as well.

Discipline is important as a developer when writing programs and tests to ensure quality of the product, as well as ethical commitments. Software development and testing is a continuously changing field with new languages and frameworks being introduced. It takes discipline as a developer to continue to stay educated on trends and topics in the field that may be important for project completion. A developer also requires discipline to have the integrity to make important ethical decisions with project development. It may be easier to produce a flawed product that does not meet requirements, while saying that it does, but that is not the right thing to do.

References

GeeksforGeeks. (2023, February 6). *Software testing techniques*. GeeksforGeeks. https://www.geeksforgeeks.org/software-testing-techniques/

Pittet, S. (n.d.). What is code coverage?. Atlassian. https://www.atlassian.com/continuous-delivery/software-testing/code-coverage