Jason Marcil

CS-320 Project Two

22EW3

1. **Summary**
   1. Describe your unit testing approach for each of the three features.
      1. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.

This project for creating a mobile application for the client consisted of three separate milestones which combined into the overall application. Each milestone was designed to allow the client to create contacts, tasks, and appointments with specific needs that were outlined in the software requirements for each type. Each piece of the project was created within their own service classes in the forms of “Contact Service, Task Service, and Appointment Service.” Within each of these packages, there was a class for the type of entry itself, a service class that executed the state of that type, and classes that ran Junit tests for each type and service class. Creating the Contact Service file was the first task, which required a contact.java class where a contact must have a unique ID, a first name, a last name, a phone number, and an address field, each with character limits. The file also required a contactservice.java class that allows a user to add, delete, and update those variables. After completing these requirements, I found that the “Task” and “Appointment” requirements were very similar, so I was able to simply modify my “Contact” code to fit the requirements for the other two files.

* + 1. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were **effective** based on the coverage percentage?

I believe that my Junit tests were created with high quality because I ran tests for every scenario that the software requirements outlined. My JUnit tests were also built in a way where I could run several instances for a particular test which helped to verify that the code coverage of my tests was thorough. To further ensure that my tests were working correctly, I ran false negative instances to make sure that the tests would fail, which they did. Running this type of combination to verify that my code coverage was complete and accurate ensured that my tests were effective.

* 1. Describe your experience writing the JUnit tests.
     1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate.

My code was technically sound because I approached the design by breaking the requirements down into smaller pieces. This is usually how software programmers write code stemming from general requirements. For example, to add a contact, task, or appointment, we need to verify that an instance of any of these classes does not already exist. By this logic, to delete an instance of these classes, we need to verify that said instance exists. Writing this thought process down makes it easier to design the overall program. Creating comments throughout the program also helped to visualize the program before writing any code. These lines of code for adding a task demonstrate this approach:

Text

Description automatically generated

* + 1. How did you ensure that your code was **efficient**? Cite specific lines of code from your tests to illustrate.

My code was also written in an efficient manner by way of simplicity and the ease of which it is to read. Even without comments, it should not be too difficult for a novice programmer to understand. By using “for” loops and “if” statements, the program can run efficiently without more logic than is needed. The following screenshot illustrates an example of efficiency in what constitutes and appointment:

Graphical user interface, text, application, email

Description automatically generated

1. **Reflection**
   1. Testing Techniques
      1. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details.

I used several software testing techniques throughout this project. At the beginning, I was heavily reliant on static testing because I could not run my code to test that it was working, so most of my testing was done simply by looking at the flow of my code to make sure that it was written as logically as possible. When I would finish coding the “contact, task, and appointment” classes, I utilized the white-box testing technique by writing test cases that used the decision logic within my classes to determine if the information I was passing through the program would validate. This ultimately led to using a dynamic testing approach by running my code against these test cases to ensure that tests would pass when they were supposed to and fail when I passed false negatives into the JUnit tests.

* + 1. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details.

Since I was acting as both the developer and the tester in these milestones, I did not use black-box testing techniques. Black box testing is a method of testing where the tester does not know the inner workings and structure of the program is not known to the tester. For these milestones, I coded the classes as well as their tests, and this is what white box testing is all about. If I was given pre-built classes and was required to just run tests on those classes, then I would be using black box testing methods.

* + 1. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.

Each testing technique that I used throughout these milestones have practical uses and implications for different software development projects and situations. Since these milestones involved coding everything from scratch, it’s easy to see clear examples of this. When you are coding from scratch, you are more likely to utilize static testing methods because it’s easier to catch errors in real time especially if your IDE is pointing them out as you code line by line. This makes dynamic testing much easier because you will have likely caught most errors before running your code. White box testing was clearly a method used so far for this project because I was familiar with the development of the code, but black box testing would be more applicable if this were not the case. Learning testing techniques makes more sense from a white box perspective because I’m able to see the relationship between JUnit tests and the programs themselves.

* 1. Mindset
     1. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.

My mindset throughout this project was a mix of both programmer and tester. When I began working on the project, it had been almost a year since I had done any projects with the Java language, so most of my time working on the “Contact” milestone was spent reviewing and relearning the language. After getting over this hurdle, my focus shifted to the mindset of a tester and how to run JUnit tests. Both the programming and testing aspects of this project forced me to employ caution because I was not as confident in my abilities early in the process. I spent a lot more time analyzing my code and my tests during the first week of working on this project to ensure that I was doing everything correctly. Things started to make more sense to me as I followed the requirements and saw how the classes related to each other. As I progressed through the project, I was able to appreciate the complexity and interrelationships between the classes I was testing, especially since each milestone appeared to only require slight modifications rather than needing to code from scratch for each. It was a good experience in working smarter, not harder. This screen shot illustrates the similarity between the “Contact” and “Task” code:

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

* + 1. Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.

My approach to limit bias in the review of my code was to simply follow the requirements to the best of my ability and run tests that would cover any possible scenario of a user’s input to the program. Bias was in the back of my mind when I was creating my first JUnit tests because I just wanted the tests to pass, but I knew that this would not be acceptable to the client. There can certainly be advantages to having a developer also act as a tester, but bias is something to watch out for. One of the main advantages of keeping these two roles separated is that a tester is more likely to try to find vulnerabilities or errors in the program, especially if they did not write the program. When a software developer also acts as the tester, it can make debugging the program easier, but a programmer also needs to be thinking about every possible scenario that could be passed into the program.

* + 1. Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

It is important to be disciplined in your commitment to quality as a software engineering professional because the end users of your software will expect that it works as it should. A finished product should have the quality of a finished product, especially in terms of its functionality. It is not the end user’s job to determine that the software that are using is functional, it is the software developer and the tester. Cutting corners when coding and testing will undoubtedly lead to faulty software, so it’s crucial to be thorough, especially when testing software. While reworking existing code for new functionality can be an efficient way to create a larger application, it can create technical debt if that method leads to additional reworking of the code later in exchange for an easy solution in the short term. To avoid this, testing is the key to ensuring that technical debt is avoided even if you are going with an easier method of coding. In this project, coding the “task” and “appointment” files was easier because I was able to reuse code that I created in the “contact” file. Running thorough tests for each file made me more confident that I would not face technical debt in the future, even though I felt like I was taking a easier approach to coding instead of coding each file from scratch.