**Project Two**

Micheal White

Southern New Hampshire University

CS-320-T5519

Jonathan Norman

June 18, 2023

My testing approach starts with knowing the software requirements: what the software is intended to do and what the requirements are to perform those tasks. When working on the first two assignments, Contacts and Tasks, I first re-read the scenarios and made a list of what functions the code needed to perform like add, delete, and update a contact or task. Then I had to define the variables that would be used. For example, in the Contact class, we needed a contact ID, firstName, lastName, Number, and Address variables to perform the operations. Additionally, I needed to know the valid length of each variable and that none of them could be null.

A screenshot of a computer

Description automatically generated

I wanted to be as efficient as possible with the constructors. If any variable came back as null or went over the length requirement, an exception would be thrown. This validation check, if passed, would allow the variable to be updated. Tests could then be built to test each variation. For example, in the Contact assignment, I created JUnit tests to ensure that the code would not accept values longer than their requirements outline and that were not null. Below is a screenshot of checking whether the contact would update with a contact ID value that was too long and another test whether it would update with a null value.

A picture containing text, screenshot, font, line

Description automatically generated

In each test case, we are only interested in the specific value we are testing and control all other variables. This simple yet solid approach allows us to modify the test to check other variables making our testing more efficient.

In the Appointment assignment, I ran into challenges to determine whether a date came before or after the current date. I have struggled working with dates in the past. First, I researched how to work with dates in Java. The requirements suggested working with util.Date which had some straightforward commands, but was out of date and not as dynamic as newer libraries like util.Calendar. After reviewing numerous examples of similar problems on Stack Overflow, I developed an approach that used both tools: Date to keep things simple while incorporating Calendar to check my current local date for the application. As we can see in my Appointment Services Test code screenshot, the JUnit tests were 90.6% successful. My goal was to consider as many combinations of invalid and null inputs as possible for coverage. The first block tests dates for null, the second block tests for valid date with other variables null, and the third block for valid combinations of all three variables. But we can see that I should have written tests to check for Past Date and Too Long Descriptions to get full coverage.

A screenshot of a computer

Description automatically generated with medium confidence

I worked to incorporate a variety of software testing techniques by designing tests based on software requirements. This strategy is a form of black box testing since at this stage I did not fully understand how the code worked internally. As I developed each project, I gained a better understanding of what the code was expected to do and used that knowledge to refine my tests to improve their accuracy and coverage.

I used what I learned from the Contacts project and applied that when writing the source code and tests for the Tasks and Appointment projects. I used boundary value analysis to check that each variable did not exceed their defined length or, in the case of dates for the Appointment milestone, that appointment dates could not be set prior to the current date. I used a form of state transition testing to check whether variables were not null and to throw an exception if it returned a null value. Statement testing for the Services classes in each project allowed me to check whether the program would accept correct and incorrect inputs. Then I could determine which statements were tested versus those that were not to ensure coverage to meet project goals.

I did not use decision testing since everything that was being tested were discrete values such as variable length, and statement testing satisfied my needs. I also did not use much white box testing. However, having a better understanding of how to work with dates and the libraries available to me would have made the Appointments project more manageable. I did not write out pseudocode or create flowcharts to help me as I have for more complex projects though this could have been useful for the Appointment assignment to determine how to deal with the date branches. Boundary value analysis did not serve any purpose for this project in my opinion since we were only testing whether variables were or were not within their expected lengths. Overall, the techniques I selected to use in these projects ensured that I created quality code that met the requirements stated in each milestone.

One important lesson I learned in this class is to be more cautious when working on different components of a larger project. Initially, I approached the Contacts assignment, the first, slowly and thoughtfully. I tend to be an overthinker and so I research a dozen variations and read resources to make sure I am on the right path. Once I completed Contacts, I had much more confidence heading into the other two. The Tasks file was like Contacts and had fewer variables to check. This further increased my confidence and made the curve ball of Appointments hit much harder. I assumed the same practices I had used could be repeated. However, figuring out how to use dates efficiently presented a new challenge and thus new challenges for writing JUnit tests. I went into the Appointments project with the bias that this would be easy. As a result, I did not allot myself extra time and wound up working on it up until 8 hours left on it. This cost me time with my family as a result.

I also realize how bias can impact developing JUnit tests for your own code. I would run test after test on my Appointment Test and Appointment Services Test and get different results each time. I would go back and make minor adjustments and re-run. I am sure I made the same changes multiple times wasting valuable time. But I was so convinced that my code was solid and should work. A fresh pair of eyes and some honest feedback could have saved me hours that I could have spent with family and friends. Every good developer must design a framework to work within that includes security and coverage testing. Frameworks organize our work habits, allow us to set timebound goals, and keep us disciplined to be more objective in our work. Had I reviewed the requirements in the Appointment project earlier than mid-week of the project I likely would have had enough time to research solutions for working with dates. This would have, in turn, given me more time to develop more JUnit tests to achieve full coverage of the project rather than 80%-90%. As I look to improve my work habits in the future, I need to plan time to review requirements and goals, conduct research as to best practices to achieve those goals, ask for peer feedback to get fresh perspective and suggestions, and strive for full coverage testing to reduce as many defects as possible in the finished product.

References:

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Software testing: An istqb-bcs certified tester foundation guide - 4th edition. BCS Learning & Development Limited.

White, Micheal, (2023) CS 320 Module Four Journal [Unpublished Paper]. Computer Science Department, Southern New Hampshire University.

White, Micheal, (2023) CS 320 Module Five Journal [Unpublished Paper]. Computer Science Department, Southern New Hampshire University.