CS-320 Michael Linsenbigler Project Two 4/13/21

The software requirements were the main target of my testing approach. Each class was written with the intention of proving the requirement functionality through a specific JUnit test function. For example, the Contact class required a phone number to be entered as exactly 10 digits and the field could not be left blank. The Junit test function was coded to test the phone number against the code using both more and less than 10 digits to ensure an error was thrown in either case. It also tested to verify an error was presented if the field was “null” or blank. One of the requirements of the Task class called for a name field that could not be longer than 20 characters and could not be blank. The Junit test function was created to make sure no error was thrown if the name field was 20 characters or less and that an error was presented with more than 20 characters or a null value in the name field. These testing approaches directly meet the software requirements set forth.

The quality of the Junit tests for the Contact Service, Task Service, and Appointment Service programs was of the highest caliber. This can be seen by the results of 100% coverage on the classes when their respective JUnit testing was performed. This means that 100% of the code was effectively evaluated and passed correctly through the testing functions for the specific class test. Also, no errors or failed tests were returned with those results and can be repeated with the same results consistently.

There were many ways that best practices, proper spacing and formatting were used in these classes to produce code that is technically sound. Also, implementing the JUnit tests in proper format and notation greatly improves the effectiveness and functionality of the testing functions. All six Test classes follow consistent and recommended naming conventions for variables to begin with “test” such as “testTooLongId” (ContactTest.java: lines 11, 23) and “testId” (TaskTest.java: lines 11, 16). Additionally, using JUnit annotations such as @Test (TaskTest.java: lines 24, 30, 36 etc.) and @BeforeEach (ContactTest.java: line 16) provides the correct structure to allow the Jupiter API to identify what test need executed and when.

In order to help with too much code redundancy when coding the required testing parameters, I utilized the setup() function (TaskServiceTest.java: line 18) to assign testing parameter variables. This method allowed for the test case variables to be called into the JUnit test function as opposed to being manually hard-coded into the fields. This was an efficient way to setup testing parameters once and then be able to recall them throughout multiple JUnit test functions throughout the test class. Another way that the code was created to perform efficiently was to keep each test separated modularly so it could be isolated in or out of the code base easily using comment lines. In this way, any bug isolating or troubleshooting of test failures could be narrowed down by test module quite easily.

The software testing techniques used for milestones three, four, and five all started with a static review of the milestone requirements documents. Reading each document provided the guidelines and system requirements that were going to be evaluated to determine a successful outcome. This step helped to drive the decisions of how to structure each class of the program and how best to test each requirement. While developing the code base, the Eclipse IDE continually ran static analysis tests on the code to identify any errors such as formatting, undefined variables, and any syntax or standard code violations. This was another testing technique that assisted with ensuring the program could compile and function upon completion. More specifically, each milestone set input requirement parameters for each field that would require a test case to verify the appropriate outcome. These input requirements included having fields with less than or equal to a designated character amount, cannot be left blank, or can allow for additions, deletions, or updates to existing records. The three main techniques used for these cases were equivalence partitioning, boundary values, and state transition testing. An example of the boundary value testing technique can be identified in module three’s test cases to verify that the phone number field of the contact class must be exactly 10 digits and could not be null. Test cases were created and exercised to for phone number inputs to be valid with exactly 10 digits and invalid with 9 and less or 11 or more digits. In milestone five, the equivalence partitioning technique was used to verify the requirement dictating the appointment id field of the appointment class could be no longer than 10 characters long and could not be null. Test cases were created and exercised to check for valid partitions of 1 to 10 character appointment id fields and also for invalid partitions of 0 or more than 10 characters. Milestone three displays an example of a state change testing technique being used for the requirement ability to update the first name, last name, phone number, and address of a specific contact id. The test case first ensures that the desired contact id exists, and then allows for the specified field to be updated. If the changes requested to these fields fail the existing field requirement tests, the update is rejected, and the transition is invalid.

A few techniques that were not implemented in these milestones include: error-guessing, static technical review, and decision table testing. The type of testing and the optimal technique used varies depending on the requirements, functionality, and scope of any program or project. There are many techniques that were not used here but they still carry value when used in the appropriate environment. Error-guessing falls under experience based techniques that relies on the skill of the tester to identify errors in logic or in systems design. This type of testing generally takes place with very experienced individuals or even with larger groups of testing teams. An example would be Google that has many people working on projects together and can all bring their own experiences to the table helping to eliminate errors that have been identified during development of previous systems. Static technical review is a more formal process that was not used in this case. This method involves a documented and well-defined process to review aspects of a project to ensure it is meeting the requirements. Technical reviews can be done in any environment that includes a team and usually one or more people that have experience and are responsible for the project’s deliverables. Agile teams are a great example of where technical review is done, in the form of scrums, to discuss, review, troubleshoot, and iterate various parts of the project. Using a decision table for testing could have been used for these projects but would not have been the most efficient method. A decision table provides a logical breakdown of all the possible input combinations and charts the appropriate response of the system. When a system has many different response abilities, the decision table can show what conditions are required to reach any of those end state results. This technique is effective when a system has to make many decisions in order to determine how to proceed. An ideal system to utilize a decision table is an ATM machine interface. Choices such as if the user enters a valid PIN, if the funds available for withdrawal, and if an eligible account exists for a transfer can all determine the machine’s response to a request.

The mindset I employed while working on this project was one of attention to detail and completeness. In order for the program to function correctly, there could be no conflicts between contacts, tasks, or appointments. The whole concept of the program is to organize and maintain accurate and detailed data that met specific criteria. Keeping this in mind, caution was used in how each parameter was measured and verified as accurate. Things like making sure character limits included the limit number such as appointment Id being greater than (>) 10 throws an error but 10 or less (< =) is a valid entry. Recognizing that the user would be able to add or delete appointments per appointment Id made it crucial to get the details of that parameter entry tested correctly to allow manipulation of the data based on that dataset. Seeing these patterns of code interacting with other areas of the project highlighted the importance of their accuracy.

Limiting bias was difficult here because I was testing the same code that I had written. Obviously, I felt the code I wrote met the requirements of the program, so I had to approach the testing portion with a fresh mindset. I tried to write the tests to make sure the code met the requirements set forth in the provided documentation. I did not write the tests to prove that the code did what it was written to do. If I had done the latter, then my bias toward my coding would have driven the way my tests read the code and provided results to verify what I had written was correct. The idea is to have the test run to prove the program will provide the required functionality. Specifically on the Task Service requirements, the user should be able to update the name and description per existing task Id. The unit test I wrote were based on that criteria and not on the code I had written to update those fields for a given task Id.

When it comes to quality as a software engineering professional, striving for the best should always be the goal even at the expense of delivering a product quickly. There will always be pressure from the customer or project manager to deliver a product as fast as possible while meeting the program requirements. However, during the SDLC there may be times that an overall issue will present itself that requires correcting or at least addressing for a longer-term solution to an existing problem. Writing a few more lines of code or implementing a workaround in a program is not the proper way to deliver a product. Speaking up and bringing issues to light, especially if they are not directly in the scope of your project, will not typically be a received very well by management. It will, however, bring much credibility to your work and reputation as a complete and disciplined software engineer. That is my future plan to avoid technical or code debt when working on software development. I will value my integrity and reputation as someone who will deliver a product of quality and not just deliver a product on time that includes short-term shortcut solutions. There may be some career drawbacks to that approach at first but having a long-term view and disciplined goals will help to overcome those beginning hurdles.