**7-2 Project Two: Summary and Reflection Report**  
Ido Tal

CS-320-T6630 Software Test Automation& QA 22EW6

August 14, 2022

**Summary:**

**Contact Feature:**

My approach was closely aligned to the software requirements. I made sure that each software requirement was given a test to make sure that my code complied with it. For the contact feature, the requirements included testing for a unique contact ID, a first name, last name, phone number, and address. For each these variables I had to test that they were no null and I also had to test that they did not exceed a certain character count. Additionally, testing had to be done to see if contacts could be added or deleted, with the user being able to update all variables except the unique contact ID. To test for these, I created tests that had filled in values for each variable, and that tested if the code let variables that exceed the character count pass, since my tests worked it did not accept inputs that exceeded the character counts, and it did not accept null inputs either. I also created tests that added and a unique contact and checked if it existed after being created, and after that test another test deleted the contact and check if it was gone.

My JUnit tests were effective based on coverage percent as they tested for all of the required features. I included the restrictions that were being tested in the main contact and contact service classes, while the test classes were created to test that those restrictions worked. This helped me make sure that I covered all of the restrictions like character count, unique contact ID, no null values, and users being able to update certain fields. I tested for all of these together and my tests all passed showing that they were all covered correctly. To guarantee that my code was technically sound, I ran the code and the test to check for errors, once I ran without errors, I knew that it was technically sound and works. My code for the testing could have been more efficient as it contained many repetitive elements. For example, I would regularly declare and define the same variable with the same value for each test, instead of doing it at the start of the class where I could have just used global variables which would mean that I would not have to declare the variable again each time.

**Task Feature:**

For the Task service feature, the requirements were similar to the contact features but only needed a unique task ID, a name, and description. There were still character limits for each variable. I approached this the same way that I did the contact class, even using the same coding techniques and features, as this feature was similar but with less variables. I created tests for the character count and checking for null values. I did this by creating specific values that were null or exceeded the character count, and then testing them against the code. I also tested that ability to add or delete by adding a task and using assert and assertEquals to check if the task was added and deleted.

To cover all of the tests for the task service feature, I needed to make sure that my code checked for the task ID, name, and description variables. Each of these could not be null and could not exceed a certain character count. Additionally, a user should be able to update the name and description along with adding or deleting a task. I was effective in covering for all of these features by testing for each of them individually until my code had successfully passed each of them. To guarantee that my code was technically sound, I ran the code and the test to check for errors, once I ran without errors, I knew that it was technically sound and works. My code for testing had some variables be declared and define in for each test despite being the same, instead of being global variables at the start of the test classes. This results in a less efficient code, which I should have improved upon.

**Appointment Feature:**

For the appointment feature, the requirements were a unique appointment ID and description, each of with had a character count limit. Additionally, there needed to be a variable for the appointment date, and the date could not be in the past. For the unique appointment ID and description, I checked that they had a value and were not null, while also checking for the character count. I did this by creating tests that inserted a null value or a value that exceeded the character count and checked if the code responded correctly to it. For the date, I created a date in the past in the test, and I then tried to input that as a value to see if my code recognized that it is in the past, and if it responded correctly to it. My approach was to follow the requirements to the word, and not add anything beyond what was requirement or any less than what was required.

To cover for all of these tests, I checked that each test passed by when it was given wrong inputs. This meant inputting null inputs, inputs that exceed character count, or that included a date in the past. This allowed me to cover all of the requirements. Additionally, I would run these tests and adjust my code in the main appointment and appointment service class until the tests passed, that way, after constantly testing, I would know that I had the most effective testing coverage by testing each required feature and passing each test as the requirement demanded. To guarantee that my code was technically sound, I ran the code and the test to check for errors, once I ran without errors, I knew that it was technically sound and works. An inefficiency in my appointment feature was the use of LocalDate as a means of finding the current date for my tests. This was inefficient since I had to use it to get the current date, then yesterday’s date, and then convert it to work with a Date variable since LocalDate variables cannot be set as equal to a Date variable without converting one variable to the other without using the Date.from() method. Otherwise, my code was efficient as it included only the basic features that it needed for the class requirements and testing.

**Reflection:**

**Testing Techniques:**

Most of my software testing techniques that I used for this project were done by running the code. I did start through static testing, by looking over my code to make sure that each requirement in the project was present and that each limitation that was included in the requirement was being tested for. Once I reviewed my code by checking for the presence of all of the requirements, I then ran my code to check for that my tests were testing the code and to make sure that there were no errors that I might have missed. For testing the code, I used assert most of the time, and sometimes I used assertEquals. These helped keep my code to minimal while checking for correct values that were within the limitations of the requirements. There are other features that I could have used for testing, for example I could have used assertNotNull, which could have simplified testing for non-null values. Additionally, there were features that I might have been able to use like asserting exceptions but I did not use them as I did not see how to best fit them in my tests, as I found other simpler means to testing.

The practical use of static testing is that it allows for quick review over a project’s current state of progress or the current state of the code to make sure that it has all of the requirements in it, and it allows for errors to be checked early before executing the code, allowing for mistakes to be picked up early and fixed. The implications of this are primarily in the time that it saves and for many projects it could also save money depending on how early mistakes are caught. When it comes to dynamic testing, the main goal is to run the code and see that it functions correctly. This means that there are no errors in executing the code and that the code outputs the intended outputs. Additionally, dynamic testing includes using features like JUnit testing which will allow different parts of the code to be tested. The practical use of dynamic testing is that it is the best way to make sure that the final product actually works while being free of bugs, as long as with was tested with as much coverage as possible. The implication of dynamic testing is that if it passes with a high coverage, it means that the code is either complete or at least functional in its current state.

**Mindset:**

When working on this project, I started writing the code based on the requirements. After writing my initial code, I then reviewed my code cautiously to make sure that I included all of the requirements for each class. I then started writing my tests, I had to be cautious of how I wrote my test since each test was meant to test for a specific segment of my code. This meant that I did not properly connect each test to the code that it was supposed to test, I would have run into errors. It was important to appreciate the complexity and interrelationship of the code and tests to make sure that, for example, the task description test, that checked that the description was not null or over 50 characters, was being testing the description getter and setter for the task description only, as this test could have had issues if it tested for the task ID instead.

Bias can be a concern when the coder and tester are the same person, as they might craft a test that fits to perfectly for their code that it would pass despite not actually testing for all of the required features. This bias could result in software being released that is not fully functional, as users could then try inputting things into the software that it was not designed for because the coder did not expect to test for it, but a tester who is not familiar with the code would have tested for. When it came to my bias, I tried to keep myself objective and limit my bias by sticking to the requirements. I made sure that the code only addressed what was required and that my test would try inputting information that the requirements expected me to test, like testing for character counts or null values for example.

Discipline is very important for software engineering. Keeping the quality of the code as high as possible would mean that the program will be dynamic and work despite any possible user input or interaction. This will make for a high-quality program that will succeed. Additionally, it is important for software engineering to try and become more efficient in their coding as to make the best possible software that they can, inefficient code could result in a piece of software to not run smoothly, or in a worst-case scenario, it could result in the software crashing if it runs into many issues due to low quality code. This also connects to making high quality tests, as not only does the software need to accept proper inputs and give the proper outputs, but it needs to do so effectively to avoid any unnecessary issues that could have been tested for. All of these can be achieved through becoming a more disciplined in software programming, engineering, and testing. The best way to avoid technical debt as a practitioner is to be constantly learning about new methods and techniques that other professionals in the field use. This means dedicating time for learning through books, lectures, videos, and through actually coding and testing.