The way that I ensured my testing approach aligned with the software requirements was to make sure I had methods that tested required methods. When we look that one of the requirements for the appointment class, we couldn’t have an appointment description that was null. With that in mind I had a JUnit that had a test to see if we didn’t meet that requirement. The test I had for it was if void test03\_checkDescription\_null() {

try {

app = new Appointment("Ap01", date, null);

} catch (IllegalArgumentException ex) {

Assert.assertEquals("Description field can't be blank.", ex.getMessage());

} to verify it wasn’t empty. This method threw an error if the appointment description was null and let me know that my method worked properly. I would go through and ensure I had other methods that met the other software requirements.

When I look at my early JUnit tests, I noticed that it was a lot more difficult for me to understand how to implement them. I was able to improve throughout time from the different resources provided as well as looking through more examples. This learning helped me understand how to continue to have a good coverage percentage. When I looked at my coverage it being positive and the tests passing showed me that I had effective JUnit tests.

When I was writing my code, I wanted to make the assert tests more simple so that I can have an understanding of it, but also so that the code could be readable. When doing that it allowed me to focus on the code being technically sound. If I was able to use simple operations when doing my evaluations, it would help create a less overall complicated JUnit test. An example of this is public void addTaskTest2() {

boolean result = TaskService.addTask("9799999999","New task", "Test new task");

assertTrue(result);

} . This test was a simple way to see if I was able to add my task. Inside the Boolean result I completed the line that should add the task and it should then pass the Boolean a value of true. Once that occurred in the assertTrue field when we test to see what it was it showing true would mean that we pass the test. This simple way to verify if it was true not only helped reduce code instead of doing separate lines for creating the Boolean, the task, etc but also helped further my understanding of using assertions and helped me make more technically sound code.

The way that I made sure my code was efficient was to limit unnecessary lines of code. I would try to get things done in a few lines without making it to compact and unreadable. A good example of this is String tName = "Test";

t.setName(tName);

assertEquals(tName, t.getName());. With this I set a string to something used it to set the task and then verified if it was set properly. I made sure to have the different operations on new lines to increase readability, while also making sure to use simple options when creating it to make the code efficient. Throughout the JUnit tests and other code, I tried to reduce the complexity of things while also ensuring that they met the requirements. Doing this helped me make sure that my code was efficient.

When I look at the testing that I did for my code I would say a lot of it was Dynamic testing. Being able to test what I expect the inputs to be as well as different variations to see if a certain input would cause an error helped me think and try new use cases in testing. With dynamic testing we are “…Working with the actual system by providing an input and comparing the actual behavior of the application to the expected behavior. In other words, working with the system with the intent of finding errors.” according to Hamilton (2022) (para. 6) Which was a great way to put it. When I was looking at ways to do the testing, I wanted to look at what I think errors would be and test them and see the output from our JUnit test if it passed or failed.

I would say that something I didn’t use enough or at all was Static testing. I noticed in the end of the project when I was encountering some roadblocks in how I should proceed in the JUnit tests that this wouldn’t have been as much of an issue if I had a set test plans. There were a lot of things in the static testing that I could have used as well like better design documents and better writing of my requirement specifications.

When I look at Dynamic testing, I can say that there are a lot of situations that we can use it. If we are trying to test different type of entry fields that we have a user inputting data dynamic testing would shine. Wherever we have things that we are expecting some type of entry with a various range of options. With using Dynamic testing, we can verify that our testing meets the requirements of the software as well as it helps us see what errors could arise from a user entering data. When I look at static testing, I see a lot broader of a use option for it. It allows you to plan out the code effectively before creating the code. You can see the different errors that may arrive in the code with this static testing from analyzing your plans and making sure that you meet the requirements. Speaking on requirements static testing will also shine in making sure you understand how to meet those requirements and plan on ways to achieve them. If I did proper static testing I wouldn’t have run into a block near the end and had a better understanding of how I could meet my requirements.

When I worked on this project, I went in with a mindset of just learning. This would be my first time working on JUnit testing and I never really heard of it prior to this class. I went in understanding that I wouldn’t know the best ways to implement them as well as not understand how to use them and make sure I got the correct coverage percentage. While going in I expected to run into errors but didn’t want to get discouraged and rather just use this to continue to grow. I would also apply caution when working on it. I would look at the different options when using JUnit tests from the supplied material as well as videos speaking on them. When looking at the complexity of this project it was important for me to understand the interrelationships. A good example of this was when I was first starting my JUnit tests. It was difficult for me to understand how to get a 100% coverage of the tests. I was able to improve this by creating more test cases and understanding that some of the test cases can be reused across the different test cases. A good example of this was Task t = taskService.deleteTask("9799999999"); assertTrue(t != null); . I was able to reuse a variant of this throughout the different tests and it helped me understand how these tests would work together and how much of it is intertwining with each other.

When I reviewed my code, I went in with an understanding that I may be biased to my own code. Instead of trusting my own assumptions on what I felt was code that would work and what I felt like was code that may have issues I instead tested all the parts and continued to review them. By going in with the understanding that I may be biased to my own code and instead just testing things more than I would other code it helped me limit and mitigate my bias. I can easily see how this could be an area of concern on the software developer side if I was to test my own code. I can see people having an over confidence in code that they may have created, and I saw that before when I first started a project last class. I realized that my over confidence in my code could lead to errors and when we look at how others may be testing their own code for important projects it could be worrying. It shows that if we are to test our own code, we need to make sure that we try to limit our biases but having a fresh pair of eyes review and test the code seems like a better solution.

I remember about hearing about technical debt a lot especially when looking at some products. From that understanding I knew that it was something that I wanted to make sure I avoid because of the problems it can cause in the future. Some of the ways that I want to avoid falling into that pitfall was by ensuring that I have adequate time to complete a project and use the correct tools. If it takes a week or more to migrate to a new toolset, but that toolset helps mitigate or remove some areas where I can see technical debt being a problem in the future, I will push to move to that. It would cost more in the present, but it would help us in the future. When I look at good examples of technical debt, I feel like the game industry is a great example. Some games push their release date with older engines that have mountains of technical debt not wanting to migrate or moving to a new engine because of the time and resource commitment. I want to focus on quality as a software engineer and that to me means not cutting corners in writing or testing or even time. I feel that it is important not to cut those corners because it can lead to errors, and we saw ways in this course that small errors had costly and even deadly results. Another way besides just ensure I have the time commitment to not cut corners is to make sure that I trust ethically that my code doesn’t take shortcuts. If I feel like a solution I am looking at is morally or ethically compromising it should help show me that I should continue to look at a new solution.

Sources:

Hamilton, T. (2022, April 30). What is dynamic testing? types, Techniques &amp; Example. What is Dynamic Testing? Types, Techniques &amp; Example. Retrieved June 21, 2022, from https://www.guru99.com/dynamic-testing.html