Rich Knoll  
CS-320  
Project 2  
6/22/2025

I believe that my testing strategy was very well aligned with the customer requirements. I made sure to review these requirements and make sure that I had a test for each of them. For example, there were several instances where a parameter could not exceed a specified length. For each of these instances, I made sure to have a test that would check this functionality. There was also the example of an appointment that could not be made in the past. I also had a test for this.

While I felt strongly that my testing did meet the customer requirements, I was actually surprised that I only achieved 83.4% total coverage when I ran them. I definitely would like to know what I would need to do in order to make up the remaining 16%, but am unsure on how to do this. I understand that it is not always possible to achieve 100% coverage, but knowing where the gap exists I think is valuable information to have.

In order to make sure that my code was technically sound I sometimes would actually test the tests. I would do this by writing out to the IDE console so that I knew what was going on. An example of this would be when I was adding items to any of the service classes. For instance, when I added a task to the task service class, I would actually write this out to the console with the displayTaskList() function that I created specifically for this purpose. As far as efficiency is concerned, I tried to make sure that I didn’t have any redundant code and consolidated code whenever possible.

I mainly used white box testing techniques for my unit testing, as I was testing the functionality of my code at the unit level. The next step after this would be to test how well these units worked together. I also used some black box techniques as I implemented boundary value analysis by testing values close to the thresholds set by the customer requirements. This can be seen in all of the functions that had parameters that could only be a certain length. This is important because most errors in code take place at or near these boundaries, so it is very important to test these cases.

I believe that mindset is probably the most important factor when trying to develop good testing for your software. You need to make sure that you fully understand the customer requirements on the one hand. And on the other, you need to make sure that you know the code well enough to know where it might not work. Or at least have enough experience in developing code to know where some of the problem areas might exist. It is not good enough to test the code in the perfect ideal scenarios. It is important to get int the head of a future user and think about the ways that they might use the code that may not have been anticipated during development. It is usually these scenarios where the errors occur, so this is where a great deal of our energy must be spent in testing.

Bias is also something that must be addressed. If you are testing the code that you have written this can have positive and negative effects. On the positive side, no one knows the code better than the person who wrote it. So this could lead to a deeper level of testing than would otherwise be possible if someone without this deep level of understanding were conducting the tests. On the other hand, it is human nature to want something that you created to work. This goes against the main mindset of a tester, which is to do whatever you can to break the code. If you don’t go into testing with this mindset, you may not be giving your code the true test that it really needs to uncover all possible bugs. Because of this, there is usually a good mixture of allowing the developer to be involved in the testing to some extent, but also requiring an unbiased third party tester to really put the code through some exercises that the developer might not have thought of.

Finally, discipline in testing is of upmost importance. Testing is often something that is overlooked, as a lot of developers and companies are eager to get things into production. Many companies think that they can “fix it on the fly” after putting it to the real world test. The reality, however, is that this usually never happens despite the best intentions of the company or developers early on. The reason for this is that when a problem does come up once code has been released into a production environment, the development team is usually already off to the next task, meaning that they no longer have the time to dedicate to the previously released software. At best, the software will receive some patches that may make it functional in the moment, but not robust over the long term. In my experience, this is the number one way companies begin to build technical debt, which is like a snowball rolling down a hill, only getting bigger the longer it goes. Because of this, it is very important to have the discipline as a developer and as a company to know the value of a solid testing process and to truly understand the “definition of done.”