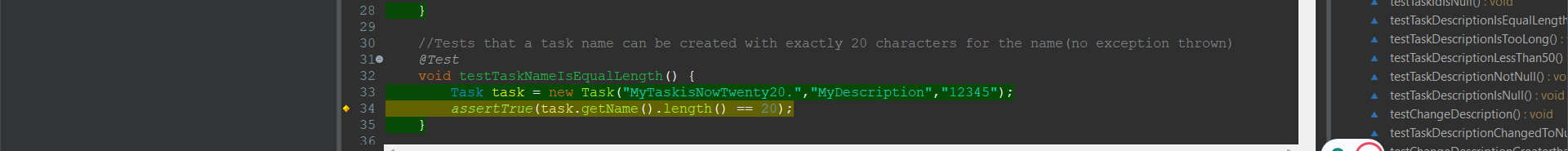
There was a lot to think about when it came to the testing approaches. Not only was I thinking about requirements, but also how to validate each method. Requirements were a big part of creating the overall code. First was reading the requirements to figure out what each constructor needed. Whether it was a name, an ID, or a description. This had to be figured out before all else so the methods for the class could be created.

While looking at the requirements I could utilize the information given to also decide what methods were needed. If a requirement was to allow something to be updateable then it would need a set method. If something was not updateable then I would avoid a set method, to prevent that item from being changed.

After creating the JUnit tests, I could use the coverage to validate that each item I created was being tested. The coverage would show me a percentage that would give me an idea of what still needed to be tested. 80 percent was considered good, but I would continue to look at it if it was below 100 percent. Even though 80 percent is good, I know that within that 20 percent something could go extensively wrong. The first image below shows the percentage coverage. The “Task.java” file has 100 percent coverage. The second image shows a highlighted version of the code in “Task.java”. The green highlighted areas show what was tested, and the yellow shows that the test will hit that section after meeting some conditions. If the area of a file was not being touched by some test, then the file would be highlighted in red.

A black rectangular object with a white border

Description automatically generated



There are many kinds of techniques that could have been used for testing software. Specifically in the project I used unit testing to test different aspects of a class. These tests also allowed me to test each area of the code under different conditions. Using JUnit 5 I was able to use assertions like “assertTrue”, to validate if certain conditions would cause the code to be true. There were multiple types of assertions I utilized for my tests including ones that validated a false statement and if it throws an exception.

One technique that was not used to a full extent was manual testing. Though I could eyeball what would happen under certain conditions, I did not physically execute the code and test using input. I did not have a main method to execute for manual testing to be possible. Another technique that was not implemented here was security testing. Although security is one of the top priorities while coding, we were impending a functionality of a unit test, and security could be implemented later.

Unit testing allowed us to test each method individually in a class. The unit tests also allowed us to test each type of circumstance based on certain criteria. Using this technique helps us find any particular problem precisely since some issues could arise that go through multiple methods. For example, if the problem resided in the “Appointment” classes set function. Then When we would call the “AppointmentService” classes update would not work as intended. Manual tests are great for debugging. It allows you to watch the value of each variable to see what happens and when. This helps to identify what needs to be done to fix any problematic programs.

While doing the project, I started to adopt a bit more caution. I used to think ahead and think, oh this is what I need it to do. Proceeding with caution allowed me to slow down and think not only about how to make something work the way I needed it to but also about what would be tested and how to proceed to pass the test. This allowed me to save some time from going back and add more code. Another side of caution that I considered was how each method worked together and how each class coincided. One example of caution is thinking ahead of possible circumstances that might arise. If a user inputs an incorrect number of characters. Knowing this could happen means I needed to have a test for the exact number of characters, a greater number of characters, and less than the exact amount. Another thing to test is to make sure that a user wouldn’t be able to provide a null response.

Developers all have their type of bias, whether they know it or not. The biggest is that they know what their code is supposed to do and feel that is all that the code is going to do. People can be very unpredictable and that is where things can go wrong. This fact is why it's very important to have an outside set of eyes looking at the code. They may be able to push the code to its limits and uncover faults that the developer didn’t realize. Sometimes even having more than one outside source can be better. Being disciplined in your software engineering techniques is highly important in someone's professionalism.

Developing your code is one thing but sometimes in research you find something you may want to implement but credit should still go to the original developer. Otherwise, it could be considered stealing and is morally and socially wrong. In this type of example, credit could be given via comments within the code.

**Citations**

Hamilton, D. (2023, August 18). *Junit tutorial with examples: Setting up, writing, and running Java Unit tests*. Parasoft. https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/

*Junit - Test Framework*. Online Tutorials, Courses, and eBooks Library. (n.d.). https://www.tutorialspoint.com/junit/junit\_test\_framework.htm

Morgan, P., & Hambling, B. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide*. BCS Learning and Development.