Janncy Mota

Prof. Toledo

My approach to each of the three features had been similar, although following each of the prompts since they were different in their own ways. I tried to align them to the best of my ability following the prompts that were given to me, for example if you take a look at the first assignment that we had to do I followed through with the tutorial that had been given to me by SNHU. The video served as something that gave me an idea on how to start programming the base of the code when it came to the Contact Service.

It did take me a while to get the hang of it when it came to the JUnit tests. I found it to be rather difficult, considering since it was the first time I ever did try to do something like this and it motivated me to do some research on it as well. At first, my tests weren’t effective at all and I wasn’t able to do it properly which was understandable but then I did my own research and thanks to the Professor’s guidance I was able to efficiently use it. I knew that my JUnit tests were effective based on the coverage percentage because I was consistently running and checking if the tests were going through. This way I could make sure to go over any of the failures that were evident, incase I needed to fix it myself personally but after having practiced and learned through the model I thankfully didn’t have to. Another way I knew that my JUnit tests were effective is because of Eclipse itself, it was telling me specifically if the tests that I had been running were going through properly. This told me exactly all it was that I needed to know and how to proceed with the assignment.

I checked that my code was technically sound by performing tests, making sure that the variables were all together. Not only that but one thing that really stuck out to me was the pathing and it was something that I had to learn to do when it came to my code and the rest of the files including the JUnit tests on Eclipse. I followed through with the prompts that were given to me, for example when it came to the Contact Class there was something that I had to specifically do which was make it under a certain amount of characters. The same can be said for the rest of the assignments, this gave me an idea of how it was that I was meant to construct the code.

**if** (firstName == **null** || firstName.length() > 10) {

**throw** **new** IllegalArgumentException("Invalid First name.");

This is an example with the name being something that could not go over ten characters. I read through with what was given to me and used the examples from the video to help create my constructors. I feel as though if it wasnt’ thanks to the resources that were given to me I likely would have had far more difficulty ensuring that my code was efficient. When it came to the JUnit tests I did my best to properly execute them by making sure to list the name of what it was that was happening in the specific test like Professor Toledo told me to. This way, I could showcase exactly what was going on in the JUnit.

**void** whenAddingValidContact\_thenSuccess() {

Contact contact = **new** Contact("1234567890", "John", "Doe", "9876543210", "123 Main St");

*assertDoesNotThrow*(() -> contactService.add(contact));

*assertTrue*(contactService.deleteContact("1234567890"));

This is an example that we see here with a valid contact being added and it being successfully added. After this was done, I would run it to make sure that everything was running smoothly and that there were no errors on my part when it came to the code itself. I had an enjoyable experience learning and figuring out exactly how it was that JUnit tests were done, especially because this was my first time trying it out in this class.

The technique that I used the most was definitely Unit testing with JUnit. This way I could focus on testing individual units of the source code and isolate the test components from the rest of the system. With the uses of annotations like @Test, @Before, and @After in order to define the test cases that I had set up and tear down the testing environments. Techniques that I didn’t use is Integration Testing because that is for components of the software and making sure that they’re working properly when they are integrated. The practical uses are this are that it’s ideal for verifying the interactions and interfaces between the different pieces of code that there are. It also detects issues that the unit tests might miss.

@Test

void timeoutNotExceededWithMethod() {

// The following assertion invokes a method reference and returns an object.

String actualGreeting = assertTimeout(ofMinutes(2), AssertionsDemo::greeting);

assertEquals("Hello, World!", actualGreeting);

}

In this project, I had to adopt a really cautious mindset because if I did any mistake and I would have to look back and find out where it was exactly that I did the mistake. But one thing that I will say is that when I set up the foundation and realized what it was with the JUnit tests especially, things began to come rather easily. For example, while using JUnit and missing an edge case in a single unit test it could allow a significant bug to go undetected which could later crash the system in a real-world scenario like we went over this week. Small errors could lead to huge failures, so I had to be really careful with what it was that I was doing on this project.

Bias is a genuine concern when it comes to software testing. By reviewing the code impartially, I personally aimed to limit bias. I’m fully aware that if developers test their own code, they might overlook certain things about it due to their familiarity or confidence when it comes to their own code potentially believing it to be error free. An example is how while I was writing JUnit tests, I could believe that to only test the scenarios which are most likely and neglect the edge cases. But a fresh set of eyes or another external programmer could approach the code differently, even uncovering overlooked bugs that I might have missed.

Committing to quality is a trait that every software engineer should have. There are many who take shortcuts and they are beneficial in the short term, but in the long term it will introduce technical debt which could result in way bigger challenges and increased costs. For example, overlooking a small bug that is identified during a JUnit test could potentially expose the system to broader vulnerabilities, data inconsistencies, and other costly repercussions that the whole team will have to face. In order to make sure to mitigate and manage technical debt, the software professional needs to prioritize their planning thoroughly through testing, regular code reviews from their fellow programming peers, and education from those that are more experienced. This is how I plan on personally avoiding these kinds of mistakes, by taking my time and sticking with this cautious mind set when it comes to the assignments. “Tests should always run at build time, especially unit tests. A build with any failing tests should be considered failed, regardless of whether the problem is in the production or the test code– this requires discipline from the team and a willingness to give highest priority to resolving failing tests, but it’s necessary to adhere to the spirit of automation.” Hayden, J. (2017, February 24). A guide to robust unit and integration tests with junit: Toptal®. Toptal Engineering Blog.

**References:**

1. Stefan Bechtold, S. B. (n.d.). JUnit 5 user guide. https://junit.org/junit5/docs/current/user-guide/
2. Hayden, J. (2017, February 24). *A guide to robust unit and integration tests with junit: Toptal®*. Toptal Engineering Blog. https://www.toptal.com/java/getting-started-with-junit