**Summary and Reflection Report**

Summary:

For Project One, I implemented unit tests using JUnit for the three primary features i.e. contact service, task service and appointment service. My testing approach followed a structured methodology.

Contact service: This test focused on these constraints.

* Ensure valid contact was created
* Ensure contact is unique and not null
* Ensure contact id, name and phone number follow the requirements such as length, not null, not more than certain number of characters etc.
* Ensure user is able to add, delete, and update contact information
* Ensure boundary cases are passing

Task service: This test focused on these constraints.

* Ensure valid task was created
* Ensure task id is unique and not null
* Ensure task id, name and description follow the requirements such as length, not null, number of character limit etc.
* Ensure boundary cases are passing
* Ensure user is able to add, delete and update task information

Appointment service: This test focused on these constraints.

* Ensure valid appointment was created
* Ensure appointment id is unique and not null
* Ensure appointment details follows the requirements such as length of characters, not null, unique id etc.
* Ensure user is able to add and delete appointments
* Ensure user is not able to add duplicate appointment

This approach was well aligned to the software requirements. Each test was directly testing the given requirements. To support my claim with specific evidence, you can see that in TaskTest, the task id cannot be more than 10 characters and I tested it by giving it 11 characters which throws an error.

The overall quality of my JUnit testing was demonstrated by the text coverage metrics. The given requirement was to have more than 80% of the test coverage and I have 100% coverage in all my tests. I’ve also tested the edge cases to make sure the software does what it is supposed to do.

I had a lot of fun writing JUnit tests. Here’s an example of my JUnit code.

@Test  
void testValidContact(){  
 Contact contact = new Contact("12345", "Jane", "Doe", "1234567890", "111 Jane Doe Street");  
 *assertEquals*("12345", contact.getContactId());  
 *assertEquals*("Jane", contact.getFirstName());  
 *assertEquals*("Doe", contact.getLastName());  
 *assertEquals*("1234567890", contact.getPhoneNumber());  
 *assertEquals*("111 Jane Doe Street", contact.getAddress());  
}

Here, I tested the required fields carefully to make sure the input validations are correct. This made sure everything was correctly instantiated.

We can look at another code snippet below where I ensured my code was efficient.

@BeforeEach  
void setUp(){  
 service = new ContactService();  
}

This code eliminated redundancy in my test and the common setups were defined in the @BeforeEach method. It helped get rid of redundant object creation and improved performance.

Reflection:

Some of the testing techniques that were employed in this project are black box texting, testing edge cases and negative testing. I was able to verify that inputs and outputs were correct without analyzing internal logic. I also tested the edge case to make sure the values pass and fail depending on the minimum and maximum allowable values for the fields. I introduced error when fail conditions were met.

Since the project was focused on JUnit testing, I didn’t test other tests such as load testing or integration testing. Integration test would have focused on integration between different components and load testing would have focused on performance when the software is under stress. These tests are vital in real life software development since we never know how the user base will increase and the software should be tested to make sure it can handle the pressure and it does what it’s supposed to do by integrating with other components.

The mindset that I adopted while working on this project was like a detective and have a detail-oriented mindset. Employing caution was very important because there are so many tiny errors that we forget to test and as a result, the software might not be doing what it is supposed to do. For example, nobody should be able to modify id since it’s unique and should be not updatable.

I tried to limit bias in my code review by switching my mindset to detail-oriented and acting as a detective. I was thinking how I can break this software and let my software developer side know about the error. Some ways to mitigate these biases would be having peer review my tests and give feedback and also follow test-driven development where we write tests before development. If I were testing my own code, then I’m sure I’d be biased and would have missed the edge cases, regex values etc. For example, for phone numbers, I know it needs to be 10 digits. Maybe I’d have forgotten about testing with alphanumeric characters or different number of digits.

Being disciplined in quality assurance is a must for software testers. This helps ensure developing a good habit of software testing and ensures long-term quality software. Cutting corners will pass the tests in short term, but can lead to huge technical debts in the future and can cost a lot of fortune for the company. I’m not too much familiar with CI/CD pipeline but I think if I learn and use the automated CI/CD pipeline to enforce testing standards, that’ll help a lot for the long run. Also, documentation is important in everything we do in software industry. I’ll also ensure my codes are readable for future engineers.

Reference:

GeeksforGeeks. (2024, January 10). *Requirement-based testing in software development*. <https://www.geeksforgeeks.org/requirement-based-testing-in-software-development/#>