**CS-320 Project Two: Summary and Reflections Report**

When approaching how to fulfill software requirements for all three features, since there was no direct contact with the client, I read each of the milestone prompts carefully to ensure I interpreted the customer’s needs correctly. The prompts clearly delineate what each object must contain. For example, a contact ID string that is no longer than 10 characters, not null, and not updatable. This information in conjunction with the rest of the variables were clear, concise, and enough to create a Contact Class with the required attributes and create a test to ensure the correct data is input.

The quality of the JUnit tests was enough to fulfill 80.1% coverage for the whole project. However, there is room for improvement since the testService class only reached 78.3 coverage. This lack of coverage can be attributed to functions in each class that were not tested because they were not part of the requirements for the assignment. However, in a real-world scenario, I imagine there is more robust and stringent testing conducted on programs.

The incorporation of feedback from our instructor was critical to eliminating some of the bias and conflict of interest from acting as both the developer and the tester. It is the job of the tester to find flaws in the code, however, as the programmer I leaned toward the idea that my code fulfilled all the requirements fully from the start. It was very revealing when the feedback pointed out errors or omissions but also a very humbling learning experience. I found that working with another person can bring to light unforeseen issues and make the software more robust and technically sound. For example, a simple omitted test on my feedback in the Task Class was the testDeleteTaskId() function depicted below:

// Test deleting a task

*@Test*

void testDeleteTaskId( ) {

TaskService taskService = new TaskService();

Task task1 = new Task("2", "homework", "Java programming");

taskService.deleteTaskId(task1.getTaskId());

}

One crucial concept I applied in my code to be used efficiently was reusability. Most of the code required for all three features was identical, with the exception of the naming conventions for functions, parameters, classes, and objects. A simple search for Contact Class objects then renaming them as Task and Appointment Class objects greatly reduced the work required to make two additional classes from scratch. Reading the entirety of the requirements for the project was vital because that gave me some foresight into the idea of reusing the code from the first class on the next ones. For example, the ContactService instantiation and addition of a contact object was pretty much identical for the Task and Appointment classes, with the major difference being the number and type of parameters being passed to each individual class constructor.

// Test that a contact can be added to contactStorage

*@Test*

void testContactService() {

ContactService contactService = new ContactService();

Contact contact1 = new Contact("1", "juan", "pablo", "7021112222", "street");

Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

contactService.addContact(contact1);

});

// Clear contact storage for next test.

contactService.deleteAllContacts();

}

Fault injection was the most common form of software testing technique used in my project. I would regularly give the program erroneous input and see how it dealt with it, when it came back with errors, I would make the necessary adjustments to validate the input or use a different approach. For example, the most issues I encountered were with the Date class because the date objects return a lot of information formatted in a very specific way. Only after researching the Data class methods, was I able to trigger an “illegal argument exception” error when inserting an improper past date for a future appointment (n.d., 2022).

Mutation testing was not utilized during my project because there is not a wide range of values and scenarios needed to test for input validation when it comes to a small program with straight forward requirements. However, I can see the potential of mutation testing in applications that require to perform calculations and many values are needed to be tested prior to deeming the program fully functional.

Every technique is practical and useful in the right context. Fault injection testing is most effective when vulnerabilities are known and are targeted directly during testing, similar to blacklisting input. This is because, each of the test seek to trigger a very specific type of fault, such as an “illegal argument exception”. However, whitelisting is more effective because restrictions are set to only receive the input that is wanted and block everything else. Mutation testing is most effective when attempting to find problems in scenarios with a wide range of values and try to break a program with unforeseen values out of specifications. However, both add benefits to the overall coverage of a program and can be used in tandem and other best practices like input validation.

A cautious mindset was definitely taken when creating the program because I knew that I would have to test it. Making the program features as simple as possible while still fulfilling the requirements was key to creating simple tests that would pinpoint the milestones requested by the client. For example, all three features required access to a data structure storage that held all contacts, tasks, and appointments in 3 separate containers, so I decided to make the storage data structure a centralized static object that all children objects would be able to access and would be easy to test.

There was a definite bias when creating tests for my program, but I believe the more cases I tested, the more problems I discovered, and the more fixes I came up with. For example, at first glance, my program seemed complete, and I was about to submit my project for grading but running it one last time revealed that the phone number requirement only worked for number less than 10, but not greater than 10. After a quick logic fix, I was able to submit the completed project, but I also became more aware of a blind spot because I was overconfident in my work.

Finally, being disciplined and honest about the program and testing of it should always be a priority for developers. According to the Consortium of Information & Software Quality (CISQ), poor quality software in legacy systems, unsuccessful software projects, and operational software failures cost the U.S. an estimated $2.08 trillion dollars in 2020. Therefore, it is of great importance for development teams and organizations as a whole to implement quality software practices from the inception of a project to avoid being part of a costly statistic (Krasner, 2022).

References

García, Boni. (2017). Mastering software testing with JUnit 5. Packt Publishing.

*Java.util.date - javatpoint*. www.javatpoint.com. (n.d.). Retrieved November 23, 2022, from https://www.javatpoint.com/java-util-date

Krasner, H. (2022, October 4). *The Cost of Poor Software Quality in the US: A 2020 Report*. CISQ. Retrieved December 11, 2022, from https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report/