//SNHU

//CS-320: Software Testing

//MOD 7: Project Two

//Student name: THOMAS SEIBERT

Within the context of this mobile application, the three features in question are Contact, Task, and Appointment. Classes were built for each feature, with specific methods and unit tests built for each object. Specifically, the JUnit testing approach was utilized for each of the three features.

For the Contact feature, contact and contactService classes were built. Then, contactTest and contactServiceTest classes were built to validate the functionalities therein which included methods to update, add and delete contacts. Assertions were coded to verify the expected outcomes against their real-time results. This ensured that each unit test correctly corresponded to each specific requirement outlined in the rubric.

For the Task feature, task and taskService classes were built. Also, taskTest and taskServiceTest classes were built to test the creation and deletion of tasks. The taskService class utilized an array for a list of tasks, as well as methods to add a task, delete a task, and update a task. The taskServiceTest class tested these functionalities using JUnit tests.

For the Appointment feature, appointment and appointmentService classes were built to ensure that appointments could be made or deleted. Also built were the appointmentTest and appointmentServiceTest classes which utilized JUnit testing to ensure the functionality thereof.

To the extent that my approach was aligned with the software requirements, I believe that I did meet the rubric requirements to the best of my current ability cross-referenced with the knowledge that I have learned over the course of seven weeks. For example, the requirement to successfully add a contact to the list was validated through a test. Specific evidence:

In the contactServiceTest class, Line 25 states:

assertEquals(1, service.contacts.size());

This code verifies that the method to add a contact works correctly by ensuring that one contact has been added to the list. If the size is not 1, the test will fail. This ensured that the code was operating as required by the customer (rubric).

The overall quality of the JUnit tests may be defended by the coverage percentage which was achieved. Initial coverage test results indicated approximately 70.5% coverage across all classes. The reason for this not being 100% initially, is that I mistakenly mistyped the name of a function in a class. After debugging and correcting this error, improved coverage results suggested that most of the code paths were tested and were indicative of sufficient reliability.

To ensure that the code was technically sound, I adhered to the best practices that I've learned in college thus far such as clear naming conventions and modular test cases. For example, in the contactServiceTest class, I ensured that the method to delete a contact was tested thoroughly. The code is as follows:

@Test

public void testDeleteContact() {

contactService service = new contactService();

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

service.addContact(contact);

service.deleteContact("1234567890");

assertEquals(0, service.contacts.size());

}

As for the efficiency of the code, this is demonstrated by the testing the performance of certain methods. Specifically, the use of direct assertions. For example, in the contactTest class, the use of assertEquals directly compares expected values with actual values. This approach is efficient for testing because it provides immediate feedback on the correctness of the contact object's state. Code:

@Test

public void testValidContactCreation() {

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

assertEquals("John", contact.getFirstName());

assertEquals("Doe", contact.getLastName());

assertEquals("1234567890", contact.getPhone());

assertEquals("123 Main St", contact.getAddress());

}

The software testing technique which I employed during this project was unit testing, specifically JUnit in Eclipse, as I detailed earlier in this report. The primary characteristic of unit testing focuses on testing individual components and methods in isolation. Every test case was designed to validate a specific method or functionality within the code.

Specifically, every service (contactService, taskService, appointmentService) had a corresponding test class which verified the correctness of their methods. Every test case was structured to check for various scenarios that might occur, which ensured that every feature met their requirements.

The other software testing techniques which were not utilized were integration testing and system testing.

Integration testing, tests the interaction between integrated components to ensure that they work together correctly as expected. Because I focused only on unit testing within the confines of this project, integration testing was not utilized.

System testing, is a form of high-level testing which evaluates the complete, integrated software product. If this had been done as part of the project, it would have involved cross-checking the application's compliance with requirements in a "real world", business type environment.

Now, to discuss the practical uses and implications of testing techniques for different software development projects.

1. Unit Testing.

Practical: unit testing validates individual components which ensures that they function correctly.

Implications: necessary for maintaining code quality.

2) Integration Testing.

Practical: integration testing ensures that components work together as intended.

Implications: necessary for identifying issues in component interactions.

3) System Testing.

Practical: system testing validates the complete system against requirements.

Implications: necessary for ensuring that the software meets user needs in the "real world".

In terms of mindset, I was very cautious in writing the code because I did not want to make any mistakes, although since my coverage results were less than 100%, apparently, I was not cautious enough. When I write code, I take an incremental approach and run frequently throughout the process before the program is complete.

It was paramount to understand the complexity and interrelationships of this code to identify issues early on. An example of why this is, can be found in the parameters of a method in the contact class. Specifically, the parameters for the first name, and how those expected results are tested against the actual results using the contactTest class. If the code and their results meet the requirements, then this is considered a success. Example, this method in the contact class:

public void setFirstName(String firstName) {

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

throw new IllegalArgumentException("First name can't be null and must be up to 10 characters.");

}

this.firstName = firstName;

}

Against the test in contactTest class:

@Test

public void testInvalidFirstName() {

assertThrows(IllegalArgumentException.class,

() -> new contact("1234567890", null, "Doe", "1234567890", "123 Main St"));

}

In terms of bias, I purposely try to have as little as possible when building my code from the beginning. If anything, a positive, lenient bias when reviewing one's own code will probably lead to the creation of errors and bugs. I expect mistakes and errors to be made, which might be interpreted by others as a negative bias, but I view it as an intense scrutiny. In this project, the utilization of unit testing of every requirement, plus test coverage analysis revealed a result of less than 100%, which translated to a fair and unbiased review of the code.

Being disciplined in my commitment to building quality code as a software engineering professional is paramount and key. There can be no compromise in this area, likewise there can be no compromise in cutting corners when it comes to writing and/or testing code. Cutting corners leads to increased technical debt which very often requires more time to fix later, leading to a backlog of issues.

To avoid technical debt, I plan to regularly test code in an incremental manner, in the same way that I build code. Continuous testing and continuous integration throughout the process of building an application. Another good strategy is to refactor regularly to improve the quality of the code. An area that I can improve upon is that of documentation, or comments throughout the code that clearly explain what is going on.

References:

Garcia, B. (2017). *Mastering Software Testing with JUnit 5*. Packt Publishing.

Hambling, B. (2019). *Software Testing*. BCS Learning & Development Ltd.