Fisal Ikhmayes

CS-320

Project 2

04/16/23

**Summary: Alignment to Requirements**

I used a unit testing strategy for creating the contact, task, and appointment services for the mobile application, writing JUnit tests for each feature. Because I carefully read the requirements paper and built tests that fully covered the functionality described, my approach was in line with the software requirements. For the contact service, as an illustration, I created tests to make sure that new contacts could be added, current contacts could be changed, and the system would accurately respond to a query for contact information. Running the tests and confirming that they all passed allowed me to confirm that my strategy was in line with the criteria.

**Summary: Effective Tests**

My JUnit tests were successful in that they offered extensive test coverage for each functionality. I made sure my tests included a range of situations, including border cases and edge cases. As an illustration, when I tested the task service, I set tests to make sure that tasks were created, updated, and deleted correctly, and that the system provided the appropriate task information when asked. I measured the proportion of code covered by my tests using a coverage tool to confirm their efficacy. I was successful in obtaining 100% coverage, which made me feel assured about the caliber of my testing.

**Summary: Technically Sound Code**

By adhering to the recommended practices for writing tests, I made sure my code was technically sound when I wrote my JUnit tests. For instance, to make my tests simple to comprehend, I utilized descriptive test method names and comments. I also created tests that were independent of one another and did not depend on the results of other tests. The test method for the appointment service is an illustration of technically sound code:

*@Test*

void testGetAppointment() {

// Create an instance of appointmentService

appointmentService appointmentServiceTest = new appointmentService();

// Create some appointments

appointment appointment1 = new appointment("1", new Date(), "Description 1");

appointment appointment2 = new appointment("2", new Date(), "Description 2");

appointment appointment3 = new appointment("3", new Date(), "Description 3");

// Add the appointments to the scheduler

appointmentServiceTest.addAppointment(appointment1);

appointmentServiceTest.addAppointment(appointment2);

appointmentServiceTest.addAppointment(appointment3);

// Get the appointments from the scheduler

HashMap<String, appointment> appointments = appointmentServiceTest.getAppointments();

// Verify that the appointments match the ones that were added

*assertEquals*(3, appointments.size());

*assertTrue*(appointments.containsValue(appointment1));

*assertTrue*(appointments.containsValue(appointment2));

*assertTrue*(appointments.containsValue(appointment3));

}

This code creates three appointments for a new date, adds them to the appointment service, and then verifies that the correct appointments are returned when queried. This test is technically sound because it is independent, has a clear purpose, and is easy to understand.

**Summary: Efficient Code**

By reducing the amount of setup needed for each test when developing my JUnit tests, I made sure that my code was effective. For instance, I put up the appropriate test data using test fixtures so that each test could be run fast and independently. The task service test method below is an example of efficient code:

*@Test*

void testDeleteTask() {

// ccreate a new Contact object

Task task = new Task("123456", "Fisal", "This is for xxx");

// add the task object to the taskService's HashMap

taskService.addTask(task);

// test deleting a valid contact ID

taskService.deleteTask("123456");

// check that the taskServices HashMap no longer contains the removed contact

HashMap<String, Task> tasks = taskService.getTasks();

*assertFalse*(tasks.containsKey("123456"));

// test deleting an invalid contact ID

Exception exception = *assertThrows*(IllegalArgumentException.class, () -> {

taskService.deleteTask("invalid Task id");

});

//check that the correct exception is thrown

*assertEquals*("Id does not exist", exception.getMessage());

}

This code Creates a new task and adds that task. To delete the task, it checks the Task ID. The program checks if the ID exists then deletes it and if no ID exists it will throw an error that the ID does not exist.

**Reflection: Techniques Employed**

I used a variety of software testing strategies in this project, including unit testing, integration testing, and regression testing.

My main testing strategy comprised unit testing, which required checking the functionality of each individual unit or module of code. Each method of the classes I created for the contact, task, and appointment services had tests written for it using JUnit.

To make sure that different modules interacted properly, integration testing examined their interactions. I used integration testing in this project to make sure the contact, task, and appointment services could exchange information.

In this project, regression testing was also crucial. After making changes to the code, I ran regression tests to ensure that the new changes did not break any existing functionality.

**Reflection: Other Techniques**

There are other software testing methodologies, such as acceptance testing, usability testing, and performance testing, though I did not utilize them in my project.

The purpose of acceptance testing is to determine whether the program satisfies the requirements set out by the client. It guarantees that the software carries out the tasks that are expected of it and that it lives up to user expectations (Software Testing Fundamentals).

Testing for usability entails determining how user-friendly the program is. It makes sure that the software is simple for end customers to use and comprehend.

Performance testing examines how the software functions under various circumstances. It guarantees the software's ability to manage massive amounts of data and its ability to react promptly to user requests (Software Testing Fundamentals).

**Reflection: Uses and Implications of Techniques**

In this project, I used unit testing, integration testing, and system testing as my software testing methodologies. Each feature was tested independently through unit testing to make sure it functioned as intended. The features' expected interoperability was checked via integration testing. To guarantee that all features of the program were functioning properly, system testing was performed to test the application as a whole.

Acceptance testing, performance testing, and security testing are additional software testing methods that were not employed in this project. To make sure that the system complies with the needs and specifications of the client, acceptance testing is used. The speed, scalability, and stability of the application are assessed through performance testing. To find and fix potential security flaws in the application, security testing is utilized (Software Testing Fundamentals).

Depending on the scenario and type of software development project, different testing techniques have different practical applications and ramifications. For projects where the client has specific demands that must be fulfilled, acceptance testing may be essential. Applications that are supposed to handle a lot of traffic might need to undergo performance testing. Applications that handle sensitive data or have strict security needs should undergo security testing.

**Reflection: Caution**

As a software tester, I exercised caution by carefully going over and testing the code to make sure it complied with the software's specs. To make sure that all potential problems were found and fixed, it was crucial to understand the intricacy and interconnections of the code I was testing. For instance, I scrutinized the code before testing the contact service to make sure that all contact information was being stored correctly and that the user interface was showing the data as it should.

**Reflection: Bias**

I made an effort to approach the testing procedure with an open mind and without any preconceived preconceptions about how the code should function in order to reduce bias in my review of the code. In order to direct my testing process and uncover unbiased proof to back up my conclusions, I concentrated on the program needs and specifications. Bias may be an issue if I was in charge of testing my own code since I might have preconceived notions about how the code ought to function and might fail to notice potential problems or flaws.

**Reflection: Discipline**

As a software engineering professional, it is crucial for me to maintain discipline in my dedication to quality in order to make sure that the code I write and test adheres to the highest standards. Cutting corners when writing or testing code can result in technical debt, which can be expensive to resolve and have an effect on the application's overall operation and quality (Software Testing). I intend to continuously evaluate and test my code to make sure it complies with the software requirements and specifications in order to prevent technical debt as a practitioner in the area. For instance, I carefully examined the code before testing the appointment service to make sure that all appointments were being scheduled and shown appropriately and that any conflicts or problems had been found and fixed.

Sources:

Software Testing Fundamentals. (n.d.). Software Testing Fundamentals. Retrieved April 12, 2023, from <https://softwaretestingfundamentals.com/>

Software Testing: An ISTQB-BCS Certified Tester Foundation Guide: <https://ebookcentral-proquest-com.ezproxy.snhu.edu/lib/snhu-ebooks/detail.action?docID=5837074>