Freddy Case

CS 320

Robert Tuft

10/20/2024

Summary and Reflections Report

In the development of the mobile application for managing contacts, tasks, and appointments, my approach for unit testing ensured that each feature met the specified requirements and would function as expected. My focus was on validating individual components, ensuring there were constraints for user input, and confirming that business logic was applied correctly. This approach was applied to the Contact, Task, and Appointment functionalities using JUnit testing methods.

Unit testing for the three features was primarily focused on validating contact creation and modification. I wanted to make sure that my testing would verify the parameters being entered produced valid results if a field was entered correctly. I also wanted to make sure that if an invalid or null parameter was entered, an exception would be thrown. Test cases for this were very similar between each three features. Here is an example of code that illustrates when an exception is thrown due to a parameter exceeding the specified length:

@Test

void testContactPhoneTooLong() {

Assertions.assertThrows(IllegalArgumentException.class, () -> new Contact("001", "Freddy", "Case", "1234567890123"));}

The overall quality of the JUnit tests written for the three features are high because I ensured test coverage for the JUnit tests took a variety of scenarios into effect. This includes both valid and invalid inputs. The tests covered boundary cases, null inputs, and duplicate handling which ensured validation of the system’s functionality. These tests provided high code coverage, ensuring that all key methods and features were thoroughly validated. By testing for high test coverage, I was able to validate that the system would work within a wide range of inputs. One example of this is a test for `TaskServiceTest` that covered the core functionality of adding, updating, and deleting tasks:

@Test

void testDeleteTask() {

taskService.addTask(new Task("002", "TaskName", "Description"));

taskService.deleteTask("002");

Assertions.assertNull(taskService.getTask("002"));}

To ensure that the code was technically sound, I wrote test cases that targeted specific functionality. For example, I used assertions to ensure that invalid inputs, such as null values or overly long strings, were handled appropriately. Here is an example of one of my lines of code handling a null test case:

*assertThrows(IllegalArgumentException.class, () -> new Task(null, "Clean Room", "Pick up clothes"));*

Another example of technically sound code is that I was able to write code that could update existing Data. Here is an example of technically sound code that updates parameters from existing data:

*taskService.updateTaskName("123", "Do Laundry");*

*assertEquals("Do Laundry", taskService.getTask("123").getName());*

To ensure that my code was efficient, I avoided using redundant and repetitive code whenever possible. This approach ensured I wrote concise test cases that were efficient and avoided repetition. Here is an example of code where I was able to get the test cases to do their job without the need for excessive and repetitive code:

*@Test*

*void testDeleteTask() {*

*// Test deleting a task from the task service*

*TaskService taskService = new TaskService();*

*Task task = new Task("123", "Clean Room", "Pick up clothes");*

*taskService.addTask(task);*

*taskService.deleteTask("123");*

*assertNull(taskService.getTask("123"));}*

The software testing techniques I employed in the application was unit testing, which was utilized through JUnit. This was completed by using @Test blocks which validated that objects were created, and that null checks were completed for each test case. Unit testing is essential for software development projects because it allows developers to test a multitude of scenarios and fix issues before detection of faults becomes more expensive. Within the project, I also utilize integration testing as a testing technique. This is when I combined the three separate features, along with their test cases. For the project, I ensured that all three of the features communicated with each other seamlessly.

The software techniques that were not employed are system testing and acceptance testing. System testing ensures that the product meets the client’s needs by checking performance, security, and usability. This is when we make sure the program works before we deliver it to the client. Lastly, acceptance testing is testing the finished product with the clients to ensure it meets their needs. This usually entails clients being hands on with the product to make sure it will work!

The mindset I adopted working on this project was a persistent mindset. For the Contact feature, I originally had trouble with setting up and writing test cases as it was my first time doing so. I dealt with a lot of problem solving to initially get the hang of how JUnit works. But once I was able to figure it out, writing the Task and Appointment test cases proved to be much easier as I understood how test cases worked. Despite this progress, I remained cautious because I understood that great test coverage is necessary for developing sound code. Understanding the complexity and interrelationships within the code was crucial, especially when validating how one change could impact multiple areas. For example, in the TaskService class, adding a new task seemed straightforward, but I had to be cautious about handling duplicate IDs and ensuring tasks were properly updated without affecting existing ones. This required attention to how different methods interacted and how potential edge cases could cause unexpected behavior.

As a developer, it can be easy to assume that code will work as expected if code runs with no errors. But to limit bias, I was able to develop test cases that applied a high amount of test coverage so that obscure faults wouldn’t break the code. Also, as a developer, it is important that you have the discipline to not cut corners when developing code. Programming is a very detailed and sometimes tedious skill that warrants a high amount of attention. What you write in your code is what the compiler will make the computer do. Any lazy writing could cause a program to not work as intended and it could be unsafe in terms of security. Writing comprehensive and accurate tests is time-consuming, but it prevents technical debt and ensures long-term code maintainability.

All in all, learning how to implement unit testing was essential for my future in developing quality code. Implementing JUnit testing for the Service, Task, and Appointment features within the application taught me that testing coverage is necessary for programs to work as intended and to ensure that programs I help develop are reliable.