Alexander DeMarco

SNHU: CS-320 Module 7

Professor Robert Tuft

12/20/2024

**1. Summary**

In developing and testing the **Task**, **Contact**, and **Appointment** classes in the final project, I implemented a structured unit testing approach to ensure each feature met the software requirements. Early in the course, I struggled to fully understand how to interpret and improve coverage percentages, initially achieving only partial results. However, Professor Tuft suggested an article in his feedback that clarified how to use coverage metrics effectively. This guidance helped me refine my testing strategy, leading to a 92% coverage across the final project—exceeding the requirement of a minimum 80% coverage—by focusing on edge cases and under-tested pathways.

The unit testing process involved verifying that the **Task** class handled both valid and invalid inputs, such as null descriptions or overly long task IDs, ensuring robust task creation and management. For the **Contact** class, tests focused on validating proper phone number formatting, unique IDs, and non-null names, aligning with software requirements. Similarly, for the **Appointment** class, tests ensured that appointments were created with valid dates, IDs, and descriptions, while invalid scenarios, such as null or past dates, were appropriately rejected. For instance, tests like assertThrows(IllegalArgumentException.class, () -> taskService.createTask(null, "Description", "ValidID")); demonstrate effective input validation.

I improved test efficiency and maintainability by using parameterized tests and mocking techniques. Parameterized tests allowed me to validate multiple scenarios in a concise manner, while mocking ensured isolated and reliable tests without redundancy. For example, using verify(mockService, times(1)).addAppointment(any(Appointment.class)); helped confirm that operations were executed only once.

Overall, the JUnit tests were highly effective, as evidenced by the high coverage percentage and their alignment with software requirements. By exceeding the minimum requirement and focusing on edge cases and critical pathways, I ensured the tests provided confidence in the functionality and reliability of the Task, Contact, and Appointment classes. My growth in understanding testing metrics and applying them effectively throughout the project was instrumental in achieving this outcome.

**2. Testing Techniques**

For this project, I primarily used **unit testing** and **boundary testing** techniques. Unit testing involved testing individual components of the software in isolation to validate their functionality. For example, I implemented unit tests for the **Task**, **Contact**, and **Appointment** classes to ensure each feature worked independently. These tests included scenarios such as checking character limits for task descriptions or ensuring appointment dates were not set in the past. A specific example is the test assertThrows(IllegalArgumentException.class, () -> taskService.createTask(null, "Description", "ValidID"));, which confirmed that invalid inputs were handled appropriately. Unit testing is practical for catching bugs early in development and ensuring specific methods or functions work as intended (Hambling et al., 2019).

Boundary testing focuses on testing the limits of input values to ensure that edge cases are handled correctly. For instance, I tested the minimum and maximum allowed lengths for task descriptions and appointment IDs. A boundary test validated that a task description of exactly 50 characters was accepted, while one with 51 characters threw an exception. Boundary testing is essential for input validation, particularly in scenarios involving constraints such as database limits or form submissions (Hambling et al., 2019).

Techniques not used in this project included **integration testing**, **system testing**, and **acceptance testing**. Integration testing ensures that different modules of the software work together cohesively. However, since the components in this project were independent, integration testing was unnecessary. System testing validates that the entire system meets specified requirements by simulating end-to-end workflows, such as creating tasks, associating contacts, and scheduling appointments. This approach is more suited for testing user-facing applications or back-end workflows. Acceptance testing evaluates the system against user requirements to confirm it meets client expectations. This technique typically involves stakeholder review in later stages to ensure all functional and non-functional requirements are satisfied (Hambling et al., 2019).

Unit testing and boundary testing were the most effective techniques for this project, particularly during the early development stages. These methods allowed me to catch bugs, ensure individual components worked as expected, and debug smaller sections of code efficiently. They are ideal for projects with well-defined requirements, as they ensure reliable and maintainable code while optimizing the development process.

**3. Mindset**

While working on this project, I adopted a mindset of attention to detail. I approached each test by first ensuring the core functionality was working as expected, then expanded into edge cases and unexpected inputs. Caution was vital because even minor issues, such as invalid input handling or overlooked constraints, could lead to significant system failures. For example, while testing the **Appointment** class, I included edge cases such as null dates and past dates, using assertions like assertThrows(IllegalArgumentException.class, () -> appointmentService.scheduleAppointment("ID", null));. Appreciating the interrelationships of the code was crucial to prevent cascading failures and to ensure that each component interacted seamlessly as intended.

To limit bias, I consciously separated my role as a software developer and stepped into the mindset of a software tester. By approaching the code as if it had been written by someone else, I was able to critique it more objectively. This mindset allowed me to focus on attacking the edge cases and challenging the assumptions made during development. Essentially, I adopted a "thinking outside the box" approach to uncover hidden vulnerabilities. For example, while testing the **Task** class, I deliberately tested scenarios I initially assumed were unlikely, such as excessively long descriptions or invalid IDs, which revealed gaps in the initial logic.

As a software engineering professional, maintaining a disciplined commitment to quality is essential. Cutting corners when writing or testing code can lead to technical debt, which increases maintenance costs and risks over time. For instance, by thoroughly testing input constraints in the **Contact** class, I avoided potential issues like invalid phone numbers being saved, which could have complicated future enhancements. To avoid technical debt in the future, I plan to adhere to coding standards, write comprehensive tests, and integrate automated testing and code analysis tools into my workflow. This proactive approach ensures that quality remains a priority, reduces the risk of defects, and delivers reliable, maintainable software solutions.

Reference  
Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.