Courtney Warner  
08/18/2024  
CS 320 - Summary and Reflections Report

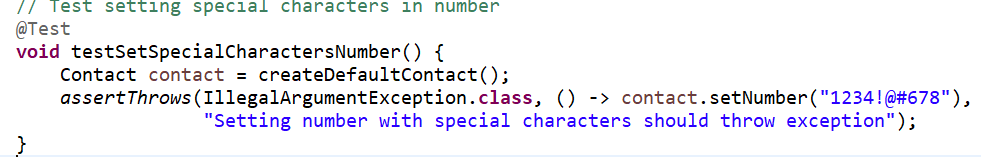
Summary - Unit Approach for Each Feature:  
In developing the Contact, Task, and Appointment Services, my unit testing approach was done in such a way as to ensure data integrity, consistency, and adherence to the specified requirements. Each of these services shared several foundational requirements, such as the immutability and uniqueness of IDs, the enforcement of non-null fields, and specific constraints on field lengths, to name a few. To help ensure these requirements were satisfied, I made the ID fields private and final in their respective classes, and I did not include setters for these fields, preventing any modifications post-creation. Additionally, I implemented validation logic to ensure that no field could be null, fields did not exceed their specified length of 10 characters, and the phone number field had exactly 10 digits. These tests ensured the services maintained consistent data integrity and met the defined business rules.  
Summary – Quality of JUnit Tests  
The overall coverage of my ‘.java’ files was at 80% or higher. I accomplished this coverage by running JUnit tests and JUnit Coverage tests to test the percentage and effectiveness of the file. JUnit testing allowed me to identify areas of the code that were not being exercised by the tests. By addressing these gaps, I ensured a high level of test coverage, which provided confidence that the core functionalities of the application were robust and reliable.

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

Summary – Experience Writing JUnit tests:  
Writing @Tests for JUnit testing was a valuable learning experience. It provided a practical approach to verifying that the code meets the required specifications and behaves as expected. Writing ‘@Test’ methods for JUnit was a new skill for me and I quickly learned the importance of systematically verifying code functionality. I began by writing tests to ensure the code functionalities for each Service ( Contact, Task, and Appointment) were correctly implemented. To ensure the tests were technically sound I focused on running tests iteratively, validating core functionalities, and testing edge cases. I execute the tests frequently and refine them until they pass with no failures or errors. I then focused on the percent coverage aiming for at least 80%. I used validation to test cases. For example, in ContactTest I wrote a test to verify that a Contact object with valid parameters was created correctly.   
A screen shot of a computer code

Description automatically generated  
In addition to verifying typical use cases, I included tests for edge cases, such as maximum field lengths and invalid inputs. Below is an example of testing for special characters in the phone number:   


To ensure my code was designed efficiently, I carefully considered handling various scenarios while minimizing redundancy. This is demonstrated when I make multiple checks in one method. An example of this from AppointmentTest is given below:

A computer screen shot of a program

Description automatically generated

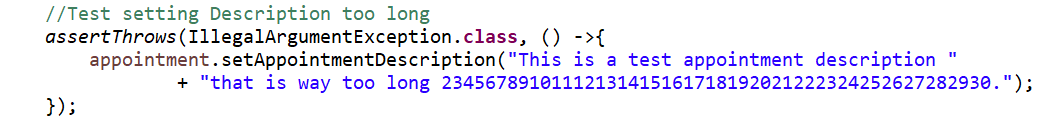
Reflection:

Testing Techniques Employed:

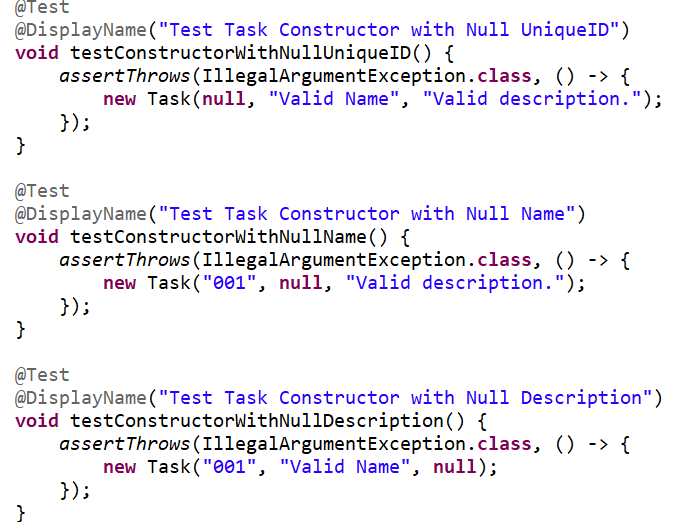
For the Service projects, I used unit testing. The unit testing involved testing components or methods in isolation to ensure they work as expected. It usually is automated and uses frameworks, the framework I employed was JUnit to run tests. This technique focused on verifying the correctness of small, and self-contained units of code. Below I have a unit test where I used a @Test method to verify core functionalities, and I am testing to make sure the task cannot be set to null.

A screenshot of a computer code

Description automatically generated

Additionally, I used boundary testing to check the behavior of the software at the edge of input ranges, such as maximum and minimum values. This ensured that the code could handle edge cases. Below is an example demonstrating an edge case for the length of the Appointment Description:   


Furthermore, I used negative testing to check how the system behaves with invalid input to ensure that the code handles errors and exceptions (GeeksForGeeks, 2023). Below is a demonstration of how I used this to check the handling of null for the unique ID, name, and description in TaskTest:



Reflection:

Testing Techniques Not Used:

Integration Testing is a technique that was not used in this project. It focuses on verifying the interactions between different components or systems (Carpenter, 2021). It checks how the units work together and whether they correctly communicate and share data (Carpenter, 2021). This would be functional in a situation where multiple components or services are integrated, such as in a larger system where tasks, appointments, and contacts interact. In addition, system testing was not utilized, which evaluates the entire application as a whole to ensure it meets specified requirements (GeeksForGeeks, July 2024). It involves testing the complete system in an environment that mimics how it would be used in the real world. This is useful in the later stages of development where the system is complete and needs to be validated for all-over functionality and compliance with requirements before release. Additionally, I did not use performance testing which assesses how the system performs under various conditions, including load and stress testing (GeeksForGeeks, May 2024). The practical use for this is important for applications under heavy load and high traffic (GeeksForGeeks, May 2024).

During this project, my primary focus was on ensuring comprehensive testing and achieving high code coverage. I aimed to maintain simplicity in my code while ensuring that all relevant functionality was tested thoroughly. This approach was intentional to make the code easier to understand, test, and maintain. Managing complexity is important for software development because it directly impacts the effectiveness and efficiency of the testing. Complex code can lead to hidden bugs and make it challenging to write effective code tests. An example would be in Contact service I focused on writing methods like ***setNumber*** that interact with other parts of the code and allowed me to identify edge cases and potential integration issues. Simplifying the validation kept it straightforward and ensured that these checks were effective and easy to test. For the Appointment service, I ensured that the deleteAppointment method was thoroughly tested. This method’s interaction with the data store and other methods meant that any issues with its functionality could affect the entire appointment system. By keeping the code modular and focusing on comprehensive test coverage, I was able to validate that appointments were correctly deleted and handled across various scenarios.  
I relied on self-assessment and feedback from my teacher to ensure the quality of my code and its testing. As a developer testing my code, it’s easy to overlook issues due to familiarity and personal bias. For instance, while testing methods in my ContactService class, I might unintentionally assume that all edge cases are covered when in reality, some scenarios might have been missed.

Being disciplined in commitment to quality is essential for a software engineering professional because it ensures the reliability, maintainability, and performance of the software. Cutting corners in writing or testing code can lead to several issues, including undetected bugs, degraded performance, and increased technical debt. For instance, if I were to skip comprehensive unit testing for the AppointmentService class, critical edge cases might go unnoticed, potentially causing application crashes or incorrect appointment handling.

To avoid technical debt, it’s necessary to adhere to best practices in coding and testing. This involves writing clean, well-documented code, implementing thorough unit tests, and continuously refactoring to improve code quality. For example, ensuring that all methods in the ContactService class are tested with various scenarios, including edge cases, helps identify and fix issues early, reducing the likelihood of problems arising in production. Additionally, revisiting and refactoring code to simplify complex logic or improve performance contributes to long-term code health and reduces technical debt. By maintaining high standards of quality and avoiding shortcuts, I aim to deliver reliable software, preventing future complications and ensuring a smoother development process.

Works Cited

GeeksForGeeks (2024, May 29). *Performance Testing – Software Testing*. Retrieved August 18, 2024, from <https://www.geeksforgeeks.org/performance-testing-software-testing/>

GeeksForGeeks (2024, July 30). *System Testing – Software Engineering*. Retrieved August 18, 2024, from <https://www.geeksforgeeks.org/system-testing/>

GeeksForGeeks (2023, August 17). *Negative Testing in Software Engineering*. Retrieved August 18, 2024, from <https://www.geeksforgeeks.org/negative-testing-in-software-engineering/>

Carpenter, A. (2021, October 28). *Why is integration testing important?* Codecademy. Retrieved August 18, 2024, from <https://www.codecademy.com/resources/blog/what-is-integration-testing>