**7-2 Project Two Submission**

Achna Hettiarachchi

Department of Computer Science, Southern New Hampshire University

CS 320: Software Test Automation and QA

Kalysa Wilson

August 13, 2023

**7-2 Project Two Submission**

**Summary**

**Unit testing approach**

My unit testing approach for each of the three features involved creating comprehensive test suites that covered various scenarios to ensure the functionality of the classes aligned with the software requirements. For the Appointment Service, I meticulously tested the addition of appointments with different validity checks, including duplicate IDs and past dates. The approach was aligned with the requirements as it rigorously validated appointment creation and ensured appointments were stored correctly in the service. Regarding the Task Service, I focused on testing the addition, deletion, and updating of tasks, paying attention to edge cases like invalid task IDs and descriptions. While my approach to the Contact Service was thorough, I noticed that the coverage was around 31%, which indicated a need for improvement. I then dedicated time to adding more tests and increasing coverage to enhance the reliability of the service. This demonstrates my commitment to aligning my testing approach with requirements, even if it meant revisiting and enhancing my test suites.

The overall quality of my JUnit tests is underpinned by the principle of thorough coverage. While I confess that I didn't initially focus on coverage, the Contact Service's coverage growth highlights my responsiveness to the need for more comprehensive testing. My commitment to quality is evidenced by the extensive range of scenarios I tested, ranging from valid inputs to boundary cases, and by my proactive response to low coverage. By striving to align my testing with requirements and adapting based on coverage insights, I aimed to enhance the effectiveness of my JUnit tests.

**Experience writing the JUnit tests.**

Writing JUnit tests was an enlightening experience that demanded technical precision and efficiency. To ensure technical soundness, I meticulously reviewed the provided test code and cross-referenced it with the corresponding class methods. For instance, in the ContactTest class, I carefully analyzed the tests for invalid contact creation scenarios, such as null contact IDs and invalid name lengths, as shown in the following lines:

A screen shot of a computer program

Description automatically generated

By evaluating these specific lines of code, I aimed to ensure the accurate simulation of exceptional scenarios, thereby validating the robustness of the contact creation process. In the TaskTest class, I followed a similar approach to ensure technical correctness, focusing on the provided test cases such as testing task creation with a null task ID and an invalid task ID length:

A computer screen shot of a program code

Description automatically generated

These lines demonstrate my commitment to accurately simulating various scenarios to verify the correct functioning of the code.

Regarding efficiency, I optimized my approach by selecting tests that targeted critical functionality and potential edge cases. By focusing on specific fields and exceptional scenarios, I aimed to create meaningful tests without unnecessary redundancy. This strategic approach allowed me to achieve comprehensive coverage while maintaining efficiency.

**Reflection**

**Testing Techniques (Hettiarachchi, 2023)**

In Milestone 1, ContactTest.java employs Unit Testing, which focuses on isolating and testing individual units or components of the software system. JUnit is utilized to write tests for the Contact class, each method targeting specific behaviors like contact creation with valid and invalid data, and validation of contact fields' length. These tests ensure that each unit within the Contact class performs as expected. Similarly, in Milestone 2, TaskTest.java continues with Unit Testing, utilizing JUnit to test the Task class in isolation. The tests concentrate on the Task class's behavior and functionality, ensuring valid task creation, validating task fields' length, and handling invalid input data. These tests verify that each unit of the Task class operates as intended. Moving on to Milestone 3, AppointmentTest.java and AppointmentServiceTest.java embrace both Unit Testing and Integration Testing. The former follows unit testing principles by testing individual methods of the Appointment class in isolation, validating correct Appointment object creation, and their getter methods' behavior. On the other hand, the latter conducts integration testing, checking the interaction between the Appointment class and the AppointmentService class. These integration tests ensure that the AppointmentService correctly adds and deletes appointments while effectively handling various validation conditions.

In the completed milestones, certain software testing techniques were not utilized, including Integration Testing beyond interactions of two classes. This form of testing ensures that multiple components or modules in the system work together seamlessly, verifying their interactions within a complex interconnected environment. Additionally, System Testing, which assesses the system as a whole rather than its individual parts, was not employed to validate the software's overall functionality against specified requirements and identify any inconsistencies or integration-related issues. Furthermore, Performance Testing was not conducted, which would have evaluated the system's responsiveness and stability under various load conditions, measuring factors like response time, throughput, and resource utilization. This type of testing is crucial for systems handling large user volumes or high data processing, as it identifies performance bottlenecks and ensures the system's ability to handle expected workloads (Pearson, 2015).

Unit testing is a suitable method for examining individual methods or components of a software system, enabling developers to promptly detect and rectify bugs while ensuring the proper functionality of each code unit. Swift feedback during development aids in issue identification and resolution, with unit tests as documentation for code behavior expectations. On the other hand, integration testing verifies seamless interactions between diverse components, identifying integration problems early on and mitigating unexpected issues during system integration. System testing, vital for real-world scenarios, validates the entire system's functionality against specified requirements and user needs, ensuring a satisfactory performance in a production-like environment. Performance testing assesses the system's response under diverse load conditions, identifying performance bottlenecks and enhancing user experience and system reliability. Employing a comprehensive testing approach throughout the software development lifecycle guarantees an effective evaluation, combining unit, integration, system, and performance testing to deliver high-quality and dependable software that meets end-users’ expectations.

**Mindset**

Throughout this project, I adopted a cautious and methodical mindset while working as a software tester. I recognized the critical role of thorough testing in identifying potential issues that could arise when the code is deployed in real-world scenarios. I approached the testing process with the understanding that even a seemingly minor oversight could have far-reaching consequences. Appreciating the complexity and interrelationships of the code was essential to ensure that all functionalities were properly tested. For instance, in the AppointmentServiceTest class, I carefully considered the validation rules for adding appointments. By assessing the interconnected conditions, such as appointment date validity and description length, I aimed to guarantee that the code would function flawlessly and avoid unexpected behavior. Recognizing these interdependencies helped me uncover potential bugs that could have been overlooked otherwise.

To limit bias in reviewing the code, I approached each test case objectively, focusing on its compliance with the requirements and not favoring any particular outcome. I tried to avoid assumptions about how the code should behave and instead relied on the expected behavior outlined in the requirements. For example, in the TaskTest class, I evaluated the task creation process without presuming that certain values would be accepted or rejected. This impartial stance ensured that my tests accurately reflected the desired functionality. If I were responsible for testing my own code as a software developer, bias could indeed be a concern. To mitigate this, I would seek external feedback and validation from colleagues or other testers. Additionally, I would adhere to a thorough test plan that covers a wide range of scenarios, not just those that align with my initial assumptions. This proactive approach would help identify any blind spots and ensure that my tests are unbiased and comprehensive (Margheim, 2020).

Being disciplined in committing to code quality is paramount for a software engineering professional. Cutting corners in writing or testing code can lead to technical debt, where shortcuts taken now result in more complex and costly maintenance efforts later. In the long run, this erodes product stability and hampers the ability to deliver new features promptly. For instance, if I were to disregard comprehensive testing of the AppointmentService class, overlooking specific validation rules, I could introduce errors that may not manifest immediately but could cause system failures down the line. Similarly, skimping on writing meaningful comments or documentation might save time initially but can lead to confusion for fellow developers working on the codebase. To avoid technical debt, I plan to adhere to best practices in software development. I will consistently write clean, modular, and well-documented code. Regular code reviews and collaboration with peers will help catch potential issues early. I will also dedicate time to thorough testing, ensuring I cover various scenarios and edge cases. By adopting this disciplined approach, I aim to build robust and maintainable software solutions without accumulating unnecessary technical debt (Arora, 2022).

**References**

Arora, V. (2022, May 4). How To Avoid Technical Debt. *Planet Crust.* https://www.planetcrust.com/how-to-avoid-technical-debt?utm\_campaign=blog

Hettiarachchi, A. (2023, July 30). 5-2 Journal: Software Testing Techniques. *SNHU CS 320 Assignments.*

Margheim, S. (2020, December 17). How to Combat Bias in Software Testing. *Test10.* https://test.io/resources/blog/how-to-combat-bias-in-software-testing

Pearson, L. (2011, September 11). The Four Levels of Software Testing. *SEGUE Technologies.* https://www.seguetech.com/the-four-levels-of-software-testing/