**7-2 Project Two Submission**

**Allison Doyle**

**Southern New Hampshire University**

**CS-320 Software Testing, Automation, and Quality Assurance**

**Instructor Federico Bermudez**

**15 December 2024**

My unit testing approach was to align with the given software requirements, specifically with the goal of validating functions and constraints for the Contact, Task, and Appointment services. For the Contact Service, tests such as testInvalidIDInConstructor() ensured IDs were unique, non-null, and within the 10-character limit, while testInvalidPhoneNumberInConstructor() validated that phone numbers were exactly 10 digits. In the Task Service, tests like testAddTask() and testDeleteTask() confirmed proper creation, deletion and editing of tasks. On the other hand, testDuplicateTaskIDHandling() ensured task IDs remained unique and cannot be edited. For the Appointment Service, testAppointmentDateNotInThePast() verified that appointments could not be scheduled in the past, and testAppointmentDescriptionTruncation() ensured descriptions longer than 50 characters were truncated as required. These tests addressed specifications directly, with error handling, proper data validation, and overall system integrity, as reflected in the coverage percentages achieved.

The quality of my JUnit tests can be seen in their effectiveness, as is evident by having over 90% code coverage across all the service classes. I intentionally made effort to make sure that all of the modules in every class were tested properly. For example, tests like testInvalidIDInConstructor() and testInvalidPhoneNumberInConstructor() validated both valid and invalid input scenarios, confirming proper execution and compliance with the requirements. Tests like testAppointmentDateNotInThePast() and testDuplicateTaskIDHandling() covered key branches in the code, ensuring that all aspects of the code was tested.

I ensured that my code was technically sound by implementing validation practices in the tests and verifying edge cases to ensure the system works properly in every scenario. For example, in ContactTest, the test testInvalidIDInConstructor() includes a line to validate that a null ID throws an appropriate exception with a relevant message, therefore, ensuring the constructor handles invalid input properly. Also, in AppointmentTest, the test testAppointmentDateNotInThePast() includes a line which confirms that the system rejects past dates. These tests validate the constructors and methods, to make sure the code was technical soundness among all the different features.

I ensured that my code was efficient by making effort to reduce redundancy in test cases and focusing on precise assertions instead. One example is in the ContactTest, the parameter test is used to allow me to test multiple invalid inputs in a single method, avoiding repetitive tests. Another example is assertions used like assertThatThrownBy(() which verified behavior clearly without further action. This approach makes the testing process smoother while ensuring all the required scenarios were covered within the code.

In this project, I primarily used unit testing techniques to ensure the functionality and consistency of the system. Unit testing focused on validating individual components, such as the Contact, Task, and Appointment classes, in isolation from other parts of the system. This allowed me to confirm that each method and constructor behaved as expected under different conditions. Example, tests like testInvalidIDInConstructor() checked edge cases for invalid IDs, making sure the system rejected null or values that are too long. Boundary testing was used to assess inputs at their limits, like testing descriptions exceeding 50 characters in testAppointmentDescriptionTruncation() and verifying invalid dates in testAppointmentDateNotInThePast(). These techniques were characterized by their focus on specific inputs, outputs, and edge cases, ensuring robust validation of all critical functionality.

Other software testing techniques that I did not use for this project include integration testing, system testing, and performance testing. Integration testing focuses on validating interactions between different modules to ensure they work together as expected. For example, testing how the ContactService interacts with the TaskService when associating tasks with contacts would fall under this category. System testing evaluates the entire application, ensuring all components function correctly in a real-world scenario, such as testing the complete flow of creating and managing contacts, tasks, and appointments in the mobile application. Performance testing examines the system under certain metrics like response time and resource usage. Simulating hundreds of concurrent task additions could assess the system's efficiency and stability under heavy load. These techniques are important for larger systems but were outside the scope of this project, which focused on unit testing.

Each software testing technique serves different purposes based on the project’s scope. Unit testing is crucial in early development to figure out individual components, as used in this project to ensure the system works as it should. Integration testing is essential for systems with interacting modules to confirm communication. System testing validates the system’s functionality as a whole, ensuring that it meets specific requirements. Performance testing assesses system stability under heavy loads, crucial for high-demand platforms like social media or financial systems. Together, these techniques ensure comprehensive quality assurance.

While working on this project, I took a very detail-oriented approach to identify issues and correct them. While testing the code, I deliberately challenged the code by testing edge cases and invalid inputs to find potential errors. As an example, in the Appointment service, I ensured appointments with past dates were rejected by writing tests like testAppointmentDateNotInThePast() to validate strict date constraints. Another example ensuring that TaskService maintained unique IDs prevented downstream errors when tasks were linked to other modules. This mindset allowed me to anticipate potential issues and verify that the system handled all scenarios appropriately.

To attempt at limiting bias in my review of the code, I approached testing as if I were an external party and focused on assessing the code rather than confirming it worked. I intentionally tested scenarios that might not meet my expectations, such as providing null or overly long inputs in the Contact class through tests like testInvalidIDInConstructor(). Bias can be a concern when testing your own work because you might subconsciously allow your assumptions to lead you to conclusions instead of reviewing the code objectively. When creating the TaskService class, I initially assumed task IDs would always be unique. By writing the testDuplicateTaskIDHandling() test, I was able to disprove this assumption and improve the service's error handling. This external perspective helped ensure a thorough and objective review, minimizing overlooked edge cases.

Being disciplined and committed to producing quality work is critical when developing software as a professional because cutting corners in writing or testing code can lead to possibly costly, damaging software. Skipping thorough testing may result in defects that surface later, requiring significant rework and damaging user’s trust in your work. To avoid this, I plan to make effort to adhere to best practices such as thorough unit testing, maintaining comprehensive documentation, and using automated testing mechanisms to ensure consistency. For example, automated tests can regularly check edge cases like the uniqueness of task IDs or the validity of appointment dates, reducing manual effort while maintaining high-quality code. By consistently prioritizing quality and addressing potential issues early, I can ensure maintainable and reliable software.