**Module 7: Project Two**

Nathalie Morales

CS-320

Southern New Hampshire University

10/19/2024

## Unit Testing Approach for Contact, Task, and Appointment Services

For the ContactService, TaskService, and AppointmentService, I employed a structured unit testing approach focusing on core functionality and error handling. Each service had the following testing structure:  
1. Adding entities: For contacts, tasks, and appointments, I ensured that each entity could be added successfully. Tests validated that the data was correctly stored and could be retrieved without issues.  
2. Handling multiple entries: The tests for each service confirmed that multiple contacts, tasks, or appointments could be stored and retrieved independently, ensuring no cross-contamination between entries.  
3. Error handling: Each service had tests for invalid operations, such as adding duplicate IDs, attempting to delete non-existing entries, or providing invalid data. This was essential to ensure robustness.

## Alignment to Software Requirements

The unit tests closely align with the software requirements for contact, task, and appointment management. For example, tests for adding entries ensure compliance with the requirement to allow new data creation, while tests for preventing duplicate IDs fulfill the requirement for unique data constraints.

## Defending Overall Quality of JUnit Tests

The overall quality of the JUnit tests is reflected in the strong test coverage for both success scenarios and error handling. The use of JUnit 5 features like assertEquals, assertThrows, and assertNotNull ensures that the system behaves as expected in both valid and invalid cases. The inclusion of exception handling tests provides an added layer of confidence in the robustness of the services.

## Experience Writing JUnit Tests

Writing these JUnit 5 tests was a technical and logical challenge that reinforced my understanding of unit testing best practices. I ensured technical soundness by following clear assertion patterns, such as:  
assertEquals(contact, service.getContact('12345'));  
assertThrows(IllegalArgumentException.class, () -> service.addContact(contact));  
These lines demonstrate both functional correctness and the ability to handle exceptions, ensuring reliable and maintainable code.

## Ensuring Code Efficiency

Efficiency was ensured by focusing on clear, concise tests that checked the essential functionality without unnecessary complexity. For instance:  
taskService.addTask('12345', 'Test Task', 'This is a description');  
Task task = taskService.getTask('12345');  
assertNotNull(task);  
The direct and focused nature of the tests ensures they are executed quickly, making the test suite efficient to run.

# Reflection

## Testing Techniques Employed

I primarily employed black-box testing techniques in this project, focusing on testing the input and output behavior of the services without delving into their internal structure. For example, I verified that valid inputs produced correct results and invalid inputs were handled appropriately. I also incorporated exception-based testing, which ensures that proper error handling mechanisms are in place when unexpected situations arise.

## Other Testing Techniques

I did not use white-box testing or code coverage tools such as JaCoCo in this project. White-box testing would involve testing internal structures or specific logic paths, which was not necessary for this project's scope. However, it could be useful for more complex algorithms or when optimizing performance.

## Practical Uses and Implications

In different software development projects, black-box testing is practical for ensuring a service behaves correctly from an external perspective, while white-box testing might be more appropriate for critical systems where every logic path needs validation.

## Mindset as a Tester

While working on this project, I adopted a cautious mindset, particularly regarding the complexity and interrelationship of the code. For instance, testing the updateContact method required careful consideration of how updating one contact field might affect the overall data integrity:  
contact.setFirstName('NewName');  
Ensuring that only the intended field changed was critical to the system's reliability.

## Limiting Bias

To limit bias, I treated the code as though it were written by someone else. This allowed me to be more objective in reviewing and testing my code. As a developer, it is easy to assume correctness, but this mindset ensures thorough testing, regardless of familiarity with the code.

## Commitment to Quality and Avoiding Technical Debt

Maintaining discipline in writing high-quality tests and code is crucial to avoid technical debt. Cutting corners can lead to issues that might not be immediately visible but could cause significant problems in the future. As a practitioner, I plan to continue following best practices in testing and code review, including automated testing and peer reviews, to ensure long-term code quality and maintainability.