CS 320 Project Two

Mohamed Aziz Zaghdoudi

## Unit Testing Approach

For the development of the contact, task, and appointment services in Project One, I employed a comprehensive unit testing approach to ensure the robustness and reliability of each feature.

Contact Service:

Approach: I developed unit tests to cover the creation and validation of contact objects. Each test case focused on the different attributes of the Contact class, such as contactId, firstName, lastName, phoneNumber, and address.

Alignment to Requirements: My approach was strictly aligned with the software requirements, ensuring that all constraints (e.g., maximum lengths, non-null requirements) were thoroughly tested. For example, I validated that contactId could not exceed 10 characters and phoneNumber had to be exactly 10 digits. These constraints were explicitly defined in the requirements.

Evidence: The testInvalidContactId method in ContactTest.java demonstrates this alignment by checking that an IllegalArgumentException is thrown for invalid IDs.

Task Service:

Approach: The unit tests for the task service focused on the creation and manipulation of task objects, ensuring that the taskId, name, and description attributes met their specified constraints.

Alignment to Requirements: I ensured that all requirements were covered, such as the name and description not exceeding their maximum lengths. For instance, the testInvalidTaskName method validated that a name longer than 20 characters resulted in an exception.

Evidence: The testSetNameInvalid method verifies that the setter method for name throws an exception when given an invalid value, ensuring that the task object remains consistent with the requirements.

Appointment Service:

Approach: I wrote unit tests to validate the creation of appointment objects and the functionality of the AppointmentService class, which included adding, retrieving, and deleting appointments.

Alignment to Requirements: My tests ensured that appointmentId did not exceed 10 characters, appointmentDate was not in the past, and description did not exceed 50 characters.

Evidence: The testInvalidAppointmentDate method confirms that the creation of an appointment with a past date throws an IllegalArgumentException, adhering to the requirement that appointment dates must be in the future.

## Defending the Quality of JUnit Tests

The overall quality of my JUnit tests can be defended based on their high coverage percentage and the thoroughness of the test cases. By achieving over 80% test coverage, I ensured that the majority of the code paths and branches were tested.

Effectiveness: High coverage percentage is a strong indicator of the effectiveness of the tests. For example, by using tools like JaCoCo, I confirmed that key methods and logical branches were covered, which minimizes the risk of undetected bugs.

Evidence: Specific lines of code, such as the assertThrows statements in various test methods, demonstrate the thoroughness of the tests. These statements ensure that invalid inputs are properly handled by the code, as specified in the requirements.

## Experience Writing JUnit Tests

Writing JUnit tests was a meticulous process that required careful attention to detail to ensure all edge cases were covered.

Technically Sound: I ensured the code was technically sound by validating each attribute and method rigorously. For example, the testSetPhoneNumberInvalid method checks for various invalid phone numbers, demonstrating a thorough approach to testing.

Efficient: Efficiency was ensured by grouping assertions where appropriate using assertAll, which allows multiple checks to be evaluated within a single test method. For instance, assertAll("settersAndGetters", ...) in ContactTest.java efficiently checks multiple attributes in one method.

Reflection

## Testing Techniques Employed

Unit Testing:

Characteristics: Unit testing involves testing individual components or methods in isolation to ensure they perform as expected. It focuses on specific functionality and logic within the smallest testable parts of the application.

Details: Each method in the Contact, Task, and Appointment classes was tested individually, ensuring that both valid and invalid inputs were handled correctly.

Assertion Methods:

Characteristics: Assertions are used to verify that a condition is true. If the condition is false, the test fails. Assertions provide immediate feedback on the correctness of the code.

Details: The use of assertThrows to test exception handling and assertEquals to verify expected outcomes were critical in validating the behavior of the code.

Other Software Testing Techniques Not Employed

Integration Testing:

Characteristics: Integration testing focuses on verifying the interactions between integrated components or systems. It tests the interfaces and communication between modules.

Details: Integration testing ensures that combined parts of an application work together as expected, identifying issues related to data flow and interface mismatches.

System Testing:

Characteristics: System testing validates the complete and integrated software system against specified requirements. It includes functional, performance, and security testing.

Details: System testing ensures the entire application meets the defined requirements and performs well under expected conditions.

Acceptance Testing:

Characteristics: Acceptance testing involves verifying that the software meets business requirements and is ready for delivery. It often involves end-users or stakeholders.

Details: Acceptance testing provides final validation from a user's perspective, ensuring the software fulfills its intended purpose.

## Mindset

Caution and Complexity:

Mindset: Adopting a cautious mindset was essential in ensuring the quality and reliability of the code. Understanding the complexity and interrelationships of the code helped prevent potential issues.

Example: When testing the AppointmentService class, I carefully considered how appointments were added and removed, ensuring that no data inconsistencies could arise.

Limiting Bias:

Approach: I tried to limit bias by thoroughly reviewing my code and considering all possible edge cases. Additionally, I wrote tests for both valid and invalid inputs to ensure comprehensive coverage.

Example: Writing tests for invalid phone numbers in the Contact class helped identify potential issues that might not be apparent when only considering typical use cases.

Commitment to Quality:

Importance: Being disciplined in commitment to quality is crucial for a software engineering professional. Cutting corners can lead to technical debt and future issues.

Example: Ensuring thorough test coverage helped avoid technical debt. By validating all constraints and handling edge cases, I ensured the code was robust and maintainable.

Avoiding Technical Debt: To avoid technical debt, I plan to continue writing comprehensive tests, performing regular code reviews, and refactoring code when necessary to maintain quality. For example, regularly reviewing and updating tests as requirements evolve will help keep the codebase healthy.