CS-320 Summary and Reflections Report

Aydar Fayzullin

**Summary**

1. Describe your unit testing approach for each of the three features.
   1. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.

*Appointment Class:* the unit tests for the Appointment class focus on validating the class's behavior and ensuring that it adheres to the defined requirements. The tests cover scenarios such as null or invalid input values for the appointment ID, date, and description. The approach aligns with the software requirements by verifying that the Appointment class enforces the specified constraints on input values. For example, the test method **testAppointmentIdNull()** checks if an **IllegalArgumentException** is thrown when the appointment ID is null, which ensures that the requirement for a non-null ID is met.

*AppointmentService Class:* the unit tests for the AppointmentService class primarily test the functionalities related to adding and deleting appointments. The tests validate that appointments can be added successfully, and duplicate appointments or non-existing appointments are handled as expected. The approach aligns with the software requirements by verifying that the AppointmentService class correctly manages appointments and handles potential errors. For instance, the test method **testAddDuplicateAppointmentDescription()** checks if an **IllegalArgumentException** is thrown when attempting to add an appointment with a duplicate description, ensuring that the requirement to prevent duplicate descriptions is met.

*Contact Class:* the unit tests for the Contact class focus on validating the class's behavior and ensuring the correctness of input values. The tests cover scenarios such as null or invalid input values for contact ID, first name, last name, phone number, and address. The approach aligns with the software requirements by verifying that the Contact class enforces the specified constraints on input values. For example, the test method **testInvalidFirstName\_Null()** checks if an **IllegalArgumentException** is thrown when the first name is null, ensuring that the requirement for a non-null first name is met.

* 1. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were **effective** based on the coverage percentage?

The quality of JUnit tests can be evaluated based on their coverage and effectiveness in uncovering defects. The tests cover various scenarios, including both valid and invalid input values, ensuring that the code is tested against different cases. Assertions are used to verify the expected behavior, ensuring that the actual results match the expected outcomes. Test methods are named descriptively, indicating the purpose of each test case. The tests handle expected exceptions and verify that the appropriate exceptions are thrown in error scenarios. The use of the **setUp()** method helps initialize a fresh instance of the tested class before each test, reducing dependencies between tests.

Describe your experience writing the JUnit tests.

1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate.

When writing JUnit tests, it is essential to understand the requirements, the expected behavior of the code, and potential edge cases. Writing these comprehensive test cases helped me ensure the code's correctness and robustness. It is also important to consider the maintainability of tests, keeping them concise, readable, and focused on specific functionalities.

To ensure technical soundness, the code should stick to best practices and follow the principles of clean code. The tests use specific assertions (**assertEquals**, **assertThrows**, etc.) to validate expected results. The tests focus on one aspect of the code's behavior per test method, following the principle of test granularity. The use of **setUp()** helps maintain test independence by initializing a clean state before each test. The tests cover both positive and negative scenarios, validating the code's behavior in different cases.

1. How did you ensure that your code was **efficient**? Cite specific lines of code from your tests to illustrate.

Code efficiency is normally measured based on execution time, resource usage, and overall performance. Each test case should be independent of others, allowing them to run in any order without relying on shared state. In my code, each test method operates on its own instance of the tested class (e.g., Appointment, Contact, Task) and does not modify any shared resources. Tests should focus on asserting essential properties and behavior rather than redundant or overlapping assertions. By keeping the number of assertions to a minimum necessary for correctness, the tests can be executed faster. In my code, each test method includes a set of specific assertions to validate the expected behavior. When verifying exception scenarios, it is essential to guarantee that the thrown exceptions are accurately captured and handled. For instance, in the AppointmentTest and ContactTest, specific exceptions are expected and asserted using Assertions.assertThrows to validate the correct behavior. In the ContactServiceTest and TaskServiceTest, the @BeforeEach annotation is used to set up the contactService and taskService instances before each test method.

**Reflection**

1. Testing Techniques
   1. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details.

Several software testing techniques were employed in this project. *Positive Testing* tests focus on expected and valid scenarios, verifying that the code behaves correctly in such cases. *Negative Testing* covers scenarios with invalid or unexpected input, ensuring the code handles errors appropriately. Both positive and negative scenarios are covered in the project, ensuring comprehensive testing of the code.

*Boundary Testing* validates the behavior of the code at the boundaries of input constraints, such as null values or specific length limits. In the **testAppointmentIdTooLong** method of the **AppointmentTest** class, an appointment with an ID longer than the allowed limit is created, which should throw an **IllegalArgumentException**.

*Exception Testing* verifies that the code throws the expected exceptions in error scenarios. In the **testAppointmentIdNull** method of the AppointmentTest class, an appointment with a null ID is created, which should throw an IllegalArgumentException.

* 1. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details.

The testing techniques I didn’t use while working on the assignment are: *Integration testing* - verifying the interaction between different components or modules in the system. *Performance testing* - assessing the system's performance under various workloads to ensure it meets performance requirements security testing: testing the system for vulnerabilities and ensuring it is protected against potential security threats *Stress testing* - valuating the system's behavior under extreme loads or resource constraints. *Usability testing* - assessing the system's user-friendliness and ease of use.

* 1. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.

*Positive Testing* ensures that the code functions correctly under expected scenarios. It is useful for validating basic functionality and providing confidence in the system's behavior. *Negative Testing* helps identify how the code handles unexpected or invalid input. It is crucial for robustness and ensuring the system can handle exceptional cases gracefully. *Boundary Testing* validates the behavior of the code at the edges of input constraints. *Exception Testing* verifies that the code throws the expected exceptions in error scenarios.

The practical uses and implications of other testing techniques mentioned but not employed in this project are as follows: *Integration Testing* ensures that different components or modules of the system work together seamlessly, uncovering issues related to data flow, communication, or dependencies. *Performance Testing* validates the system's response time, resource utilization, and scalability to ensure it meets performance requirements under various loads. *Security Testing* helps identify vulnerabilities and weaknesses in the system's security measures, ensuring it can withstand potential attacks or unauthorized access. *Stress Testing* tests the system's behavior under extreme conditions to assess its robustness, stability, and ability to handle high loads or resource limitations. *Usability Testing* evaluates the system's user-friendliness, accessibility, and overall user experience to ensure it meets the end-users' expectations and needs.

1. Mindset
   1. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.

Adopting a cautious mindset is important to evaluate the code and uncover potential problems. It is crucial to ensure that duplicate appointments are handled correctly, and deleting an appointment does not cause unexpected side effects. Paying close attention to the code being tested and its behavior is vital to identify potential issues. For example, when reviewing the code, a tester should carefully analyze variable declarations, method calls, and control flow to understand how different components interact and impact each other. Testers have to think beyond the expected scenarios and consider different edge cases and exceptional situations. For instance, when reviewing the code, a tester should think about cases like null values, empty collections, or boundary values that might affect the behavior of the code.

* 1. Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.

When reviewing code, it is important to limit bias to ensure a fair evaluation. Bias can lead to overlooking potential issues or not testing certain scenarios thoroughly. Ways to limit bias include following a predefined test strategy to ensure comprehensive coverage of functionalities, documenting the expected behavior and requirements to guide the testing process and minimize subjective interpretations, collaborating with peers or other team members to gain different perspectives and gather feedback on the tests.

* 1. Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

Discipline in commitment to quality is essential for software engineering professionals. By avoiding shortcuts and adhering to best practices, software professionals can ensure code correctness and robustness, easier maintenance and bug fixing, increased code readability and understandability for future developers, reduced chances of introducing regression bugs. To avoid technical debt, practitioners can follow coding standards and best practices, perform code reviews, and seek feedback from peers, write comprehensive unit tests with high coverage, refactor code when necessary to improve its quality and maintainability, continuously prioritize and allocate time for code maintenance and improvement.

The use of meaningful methods and scenarios, along with the focus on test granularity and handling different scenarios, demonstrates a commitment to quality and avoidance of technical debt.Top of Form