- CS300- Software Testing

Project 2

Gonzalo Patino

1. Summary

a. Describe your unit testing approach for each of the three features

i. **To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.**

I applied a consistent unit testing approach to ensure each feature of Contact Service, Task Service, and Appointment Service met its functional requirements. For Contact Service, tests covered adding, deleting, and updating contacts, ensuring unique IDs and validating constraints like name length and phone format, with testAddContact() verifying successful addition and testAddDuplicateContactID() preventing duplicates. Task Service tests focused on adding, deleting, and updating tasks, enforcing unique IDs and validating input lengths (testAddTaskSuccess() and testAddTaskDuplicateId()). Appointment Service tests ensured appointments were only scheduled for future dates (testAddAppointment() for valid appointments and testInvalidAppointmentDate() for rejecting past dates).

ii. **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 my JUnit tests was strong, demonstrated by over 80% code coverage across Contact, Task, and Appointment Services, which significantly reduced untested paths and improved reliability. Both positive and negative scenarios were tested, such as testAddContact() for successful additions and testAddDuplicateContactID() for handling duplicate IDs. Exception handling was verified using assertThrows(), ensuring errors were raised for invalid operations, like past appointment dates.

Edge cases, such as invalid inputs and excessively long fields, were covered to ensure robustness. Using @BeforeEach isolated each test, preventing side effects and ensuring consistency. The combination of high coverage, balanced testing, proper exception handling, and isolation ensured that the code was reliable and met the requirements effectively, aligning with the quality standards set in Project 1.

b. Describe your experience writing the JUnit tests

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

To ensure that my code was technically sound, I focused on validating each function comprehensively by using assertions that verified both expected outcomes and error conditions. For instance, in ContactServiceTest.java, I used assertThrows(IllegalArgumentException.class, () -> service.addContact(contact)); to validate that attempting to add a contact with an existing ID raised the proper exception, ensuring uniqueness as required.

Similarly, in AppointmentTest.java, the line assertThrows(IllegalArgumentException.class, () -> new Appointment("12345", pastDate, "Valid description")); ensured that no appointment could be created with a date in the past, which directly aligned with the functional requirement for valid appointments. Additionally, I used assertEquals() statements, such as in TaskServiceTest.java (assertEquals("New Description", task.getDescription());), to confirm that updates were applied correctly to the task description, ensuring the functionality worked as intended. These specific lines of code illustrate how I maintained correctness and robustness throughout my testing approach.

ii. **How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.**
To ensure that my code was efficient, I minimized redundancy by reusing setup methods and validating multiple functionalities within the same test scope where appropriate. For example, in TaskServiceTest.java, I used @BeforeEach to initialize a fresh instance of TaskService (taskService = new TaskService();), which allowed all tests to share a consistent setup without repeating initialization code. Additionally, in testSetName() and testSetDescription(), I reused the same Task instance to test both properties, which streamlined the testing process while maintaining clarity and coverage. This reuse of resources and setup helped improve efficiency without compromising the thoroughness of my tests.

2. Reflection

a. Testing techniques

i. **What were the software testing techniques that you employed in this project? Describe their characteristics using specific details.**
In this project, I primarily employed unit testing as the main software testing technique.

Unit testing involves testing individual components in isolation to verify that each one functions correctly on its own. This technique allowed me to focus on the core functionalities of Contact, Task, and Appointment services—such as adding, updating, and deleting entries—by writing specific test cases for each. For example, in AppointmentServiceTest.java, I created testAddAppointment() to verify that appointments could only be created with valid future dates. Additionally, I used boundary value analysis by testing scenarios involving maximum and minimum acceptable values, like verifying that a task name or contact field did not exceed its character limit. Techniques like exception handling testing were also employed, using assertThrows() to confirm that invalid operations, such as adding a duplicate ID or creating an appointment in the past, correctly triggered exceptions. These characteristics ensured that each unit was rigorously validated against both expected functionality and edge cases.

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

I did not use regression testing. Regression testing is used to verify that changes or updates to the codebase do not introduce new bugs or negatively impact existing features. It involves re-running previously written test cases to ensure that the existing functionality remains intact after modifications, such as feature enhancements or bug fixes. This testing technique is crucial for maintaining stability in applications as they evolve, particularly when multiple features and dependencies are involved. While this project focused primarily on unit testing to verify individual component correctness, regression testing would be beneficial in a larger system undergoing frequent updates to ensure consistent and reliable performance over time.

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

Unit testing is used to validate individual components in isolation, ensuring they function correctly before integration. It is practical for early bug detection and maintaining code quality, particularly in modular projects where frequent refactoring is needed.

Regression testing ensures stability after changes, making it crucial for projects with frequent updates or in CI/CD environments. It helps verify that new features or bug fixes do not introduce new issues, maintaining software reliability over time.

Integration testing checks how different components interact, useful for detecting issues in complex systems where multiple modules must work together. It ensures seamless communication between components, which is critical in larger, interdependent projects.

Mocking simulates dependencies during testing, isolating units to ensure consistent results. It is practical when real dependencies are unavailable or costly to use, such as in applications involving databases or third-party services, allowing reliable unit tests without external factors affecting outcomes.

b. Mindset

i. **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.**

I adopted a cautious mindset throughout the project to ensure reliability and thoroughness in testing. This involved carefully considering edge cases, such as using assertThrows() in AppointmentTest.java to validate that appointments could not be scheduled in the past. Appreciating the complexity and interrelationships of the code was crucial, as even minor errors could affect other parts of the system. For example, testing unique contact IDs with testAddDuplicateContactID() in ContactService prevented cascading issues in updates or deletions, ensuring system stability. This cautious approach helped prevent unintended side effects and maintained the overall integrity of the application.

ii. **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.**

To limit bias, I tested both successful and failing scenarios, such as using testAddTaskSuccess() for valid task additions and testAddTaskDuplicateId() for detecting duplicate IDs in TaskServiceTest.java. Bias can be a concern when testing your own code, as developers may unconsciously favor expected outcomes. For example, I made sure to include testInvalidPhoneNumber() for ContactService to challenge assumptions, which helped identify flaws that might have been overlooked if I only tested positive cases.

iii. **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** Being disciplined about quality is crucial in software engineering to avoid long-term issues. Cutting corners in testing can lead to bugs that cause major problems later. For example, thoroughly testing constraints like unique contact IDs in ContactServiceTest.java ensured data integrity. To avoid technical debt, I will stick to best practices like comprehensive testing, proper documentation, and regular code refactoring. By consistently validating edge cases, as I did with testInvalidAppointmentDate(), I can ensure the software remains reliable and maintainable.

References:

Beck, K. (2003). *Test-driven development: By example*. Addison-Wesley Professional.

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). John Wiley & Sons.