August Moews

CS 320

Professor Bermudez

February 23, 2025

For the contact, task, and appointment services, I implemented JUnit tests to validate each feature's functionality, ensuring that methods performed as expected under various conditions. My approach focused on unit testing principles, including equivalence partitioning and boundary value analysis, to cover different test cases efficiently. To align with software requirements, I designed test cases that mirrored the functional specifications provided in the project documentation. For example, in the task service, the requirement stated that a task should not exceed 30 characters in length. I created a test case to validate that input beyond this limit would trigger an exception:

@Test  
void testTaskExceedsMaxLength() {  
 assertThrows(IllegalArgumentException.class, () -> new Task("1234567890123456789012345678901"));  
}  
 This ensured that the program adhered to the defined constraints.

To defend the quality of my JUnit tests, I measured the test coverage percentage using tools like JaCoCo. The coverage results indicated that my tests covered over 85% of the codebase, ensuring that most functionality was verified. JUnit tests were effective due to their structured assertions and edge case validation. For instance, when testing appointment scheduling, I checked scenarios such as valid date and time inputs, null or empty inputs triggering exceptions, and edge cases, such as scheduling an appointment on February 29 in a non-leap year. By implementing thorough test cases, I minimized the risk of defects in the final product.

Writing JUnit tests required a methodical approach to ensure both technical soundness and efficiency. To maintain technical integrity, I adhered to the Arrange-Act-Assert (AAA) pattern, which made tests clear and maintainable:

@Test  
void testValidContactCreation() {  
 Contact contact = new Contact("1", "John", "Doe", "1234567890");  
 assertEquals("John", contact.getFirstName());  
}  
 This approach ensured readability and maintainability for future testing needs. Regarding efficiency, I used parameterized tests where applicable to reduce redundancy. Instead of writing multiple tests for different input values, I leveraged JUnit’s @ParameterizedTest annotation:

@ParameterizedTest  
@ValueSource(strings = {"12345", "ABCDE", "!@#$%"})  
void testValidTaskId(String taskId) {  
 Task task = new Task(taskId, "Test Task", "Test Description");  
 assertEquals(taskId, task.getTaskId());  
}  
 This eliminated repetitive code while ensuring comprehensive coverage.

### Reflection:

Throughout the project, I employed several software testing techniques, including unit testing, boundary value analysis, and exception testing. Unit testing focused on isolated functionality testing using JUnit, boundary value analysis ensured that edge cases (e.g., minimum and maximum character limits) were handled properly, and exception testing validated that invalid inputs triggered appropriate errors (Myers et al., 2011). One testing technique I did not use but could be valuable is integration testing, which ensures that individual modules work together correctly. This technique could have helped verify interactions between contact, task, and appointment services. Each technique has practical implications in software development. Unit testing is crucial for catching small defects early, while integration testing is vital in multi-component applications where modules interact (Myers et al., 2011).

A cautious mindset was crucial during testing, as overlooking small defects could lead to significant failures. For example, in task scheduling, failing to account for null inputs could cause runtime crashes. By implementing null-check assertions, I proactively prevented such issues:

assertNotNull(task.getDescription());

Understanding code interrelationships was also critical. Testing task creation alone was insufficient—I had to ensure that tasks interacted correctly with the appointment service. Bias can be an issue when testing one’s own code. Developers tend to assume their code works as intended, leading to confirmation bias (Leveson, 2017). To mitigate this, I wrote test cases before implementing features (Test-Driven Development, or TDD). Additionally, peer reviews were an effective strategy. Reviewing code with a fresh perspective helped identify issues I had overlooked, reinforcing the importance of collaborative testing.

Commitment to quality prevents technical debt, which arises when shortcuts are taken. A disciplined approach meant writing clear and reusable tests, ensuring 100% requirement coverage before moving to the next phase, and documenting tests for future reference (Leveson, 2017). To avoid technical debt, I plan to incorporate automated regression testing in future projects to ensure changes do not introduce new defects.

The unit testing process for Project One reinforced the importance of structured testing strategies. By employing rigorous JUnit testing, edge case analysis, and bias mitigation techniques, I ensured that my code met functional requirements with high reliability. The experience underscored the need for caution, thoroughness, and discipline in software testing, essential skills I will carry forward into future development projects.

### References

Leveson, N. G. (2017). *The Therac-25: 30 years later*. IEEE Computer Society. Retrieved from <https://www.computer.org/computer-magazine/from-the-archives-computers-legacy>

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