Austin Antles

CS 320

18 April 2025

**Software Testing Approaches and Strategies Report**

Throughout the development of our mobile application, I implemented systematic unit testing across three key features: Contact Service, Task Service, and Appointment Service. My testing approach directly aligned with the software requirements by verifying each constraint documented in the class specifications. For Contact Service, tests confirmed that first and last names couldn't exceed 10 characters, phone numbers required exactly 10 digits, and addresses remained under 30 characters. When testing Task Service, I verified that task names stayed within 20 characters and descriptions didn't exceed 50 characters. The Appointment Service tests ensured no past dates were allowed and descriptions maintained appropriate length limitations.

These JUnit tests proved remarkably effective because they achieved comprehensive code coverage by testing normal operations and exception conditions. Each class method had corresponding test cases verifying valid inputs and boundary conditions. The ContactTest class, for instance, included methods that confirmed normal contact creation alongside tests that verified exceptions for every possible constraint violation. This dual approach ensured complete code coverage since every execution path was tested.

I maintained technical soundness by implementing established testing best practices. Using the @BeforeEach annotation to create fresh test objects before each test case ensured a known starting state, as demonstrated in all service test classes. Tests followed the Arrange-Act-Assert pattern, separating setup, execution, and verification phases. I also selected appropriate assertion methods for different scenarios, using assertThrows when testing exceptions and assertEquals when comparing expected values. This structured approach made the tests both reliable and maintainable.

The efficiency of my testing code stemmed from several practical techniques. I avoided unnecessary object creation by reusing test objects when appropriate, typically creating only two test objects rather than generating new ones for each operation. Tests maintained focus on single responsibilities, with separate methods verifying different aspects of each class. In AppointmentTest.java, distinct test methods independently validated basic creation, description, and date setters. This focused approach created more maintainable tests, identifying which functionality failed when issues arose.

I primarily employed unit testing as my primary software testing technique for this project. This approach focused on validating individual components in isolation through method-level testing, boundary testing, exception validation, and state verification. Each method received dedicated test cases to verify its behavior, with special attention to boundary conditions like maximum field lengths. I systematically verified that appropriate exceptions were thrown when invalid data was provided and confirmed that operations appropriately modified the state.

While unit testing served the project well, several valuable techniques that would benefit other contexts weren’t employed. Integration testing would verify that different components work together correctly, while system testing would evaluate the entire application’s functioning as a coherent whole. Performance testing would measure response times and resource utilization, and user interface testing would confirm proper display and data manipulation. Each technique offers unique advantages for different scenarios.

The practical applications of these techniques vary widely across projects. Unit testing excels when developing reusable components or libraries, helping isolate bugs to specific methods. Integration testing becomes essential when components must interact, particularly in systems like e-commerce applications, where inventory must update properly when orders are completed. System testing provides confidence in healthcare applications where patient data must flow correctly through all modules. Performance testing proves critical for applications with specific response requirements, such as financial platforms handling market volume spikes.

Throughout this project, I kept a cautious mindset by anticipating potential failure modes rather than assuming that the code would work as expected. I created tests for standard and exceptional conditions, which helped identify edge cases that might otherwise cause runtime errors. The interrelated nature of the code components demanded this caution because changes in one area could affect others. When testing AppointmentService, I needed a thorough understanding of how the Appointment class behaved since the service manages collections of appointments.

I limited bias in my testing by deliberately seeking ways the code might fail rather than ways it might succeed. This meant designing tests based on specifications rather than implementation details. Bias becomes particularly concerning when testing code, as developers naturally develop blind spots about their work. They might assume certain conditions will never occur or users will interact with the system only in expected ways. Counter-measures like pair testing or code reviews bring fresh perspectives that reveal hidden assumptions.

Discipline in software testing prevents today's convenience from becoming tomorrow's crisis. When developers take shortcuts in testing to meet tight deadlines, they create technical debt that grows increasingly expensive to address. Skipping proper exception handling tests might seem harmless during development, but could lead to mysterious crashes in production that damage user trust. The discipline to write comprehensive tests initially ultimately saves time compared to troubleshooting problems after deployment.

I plan to avoid technical debt as a practitioner by allocating appropriate time for testing during planning phases rather than treating it as optional. When faced with delivery pressure, I'll advocate for reducing scope rather than compromising quality. Furthermore, I'll invest in test automation to make thorough testing sustainable despite tight deadlines. This disciplined approach builds professional reputation and prevents the frustration of addressing the same bugs repeatedly, ultimately delivering more reliable software with less time spent on emergency fixes.