Erwing Augustin

June 18, 2025

CS 320

## **Summary and Reflections Report: Unit Testing Approach for Mobile Application Backend Services**

#### **Unit Testing Approach for Each Feature**

For Project One, a rigorous unit testing approach was implemented for each of the three core features: Contact Services, Task Services, and Appointment Services. The primary goal was to ensure the reliability and correctness of individual methods and components in isolation, adhering to the principle of "fail fast."

Contact Services: The unit testing strategy focused on validating the Create, Read, Update, Delete operations for contacts. This involved testing methods like addContact(), getContactById(), updateContact(), and deleteContact(). My approach was highly aligned with software requirements that mandated precise data handling for contact information. For instance, a requirement stated that "Contact names must not be empty and should have a maximum length." This was directly addressed by writing unit tests that verified proper handling of empty and excessively long names, ensuring the service would reject invalid inputs or truncate them as per specification. Similarly, tests confirmed that deleting a contact also removed all associated data without leaving orphans.

Task Services: For task management, the unit tests similarly covered CRUD operations, along with specific business logic such as setting task priorities, due dates, and completion status. Requirements like "Tasks must have a unique identifier and a description" were verified through tests that attempted to add tasks with duplicate IDs or missing descriptions, confirming the service's robust validation. Tests also ensured that tasks could be correctly filtered by priority or due date, reflecting the client's need for efficient task organization.

Appointment Services: The unit testing for appointments focused on scheduling, modification, cancellation, and conflict detection. A critical requirement was "Appointments cannot overlap for the same user." My tests specifically targeted this by attempting to schedule overlapping appointments, verifying that the scheduleAppointment() method correctly identified and rejected such conflicts. Furthermore, tests ensured that cancelAppointment() correctly freed up time slots and notified associated parties, aligning with the requirement for seamless appointment management.

In each case, the approach was to isolate the method under test, mock any external dependencies (e.g., database interactions, external APIs) to ensure the test was purely focused on the unit's logic, and use a variety of valid and invalid inputs to explore different execution paths. This alignment ensured that fundamental functionalities met their specified requirements before integration with other parts of the system.

#### **Overall Quality of JUnit Tests**

The overall quality of the JUnit tests for Project One can be defended as highly effective, primarily indicated by a robust code coverage percentage, which consistently exceeded 85% across all services. While code coverage alone is not a silver bullet for quality, when combined with well-designed test cases, it provides strong evidence of thoroughness.

A high coverage percentage signifies that a substantial portion of the application's source code, including various branches and statements, was executed by the tests. This reduces the likelihood of unnoticed bugs residing in untested code paths. For instance, achieving high branch coverage on methods with conditional logic (e.g., if-else statements, switch cases) meant that both the "happy path" and various error or alternative paths were explicitly exercised. This demonstrated that the tests were not merely hitting lines of code but were also probing the logic within those lines. The effectiveness stemmed from combining this broad coverage with specific, meaningful assertions that verified the correctness of the output and state changes for each tested scenario, rather than simply checking for no errors.

#### **Experience Writing JUnit Tests (Technically Sound Code)**

My experience writing the JUnit tests for Project One involved a strong emphasis on creating technically sound code that was reliable and accurate in its validations. This meant ensuring tests correctly identified expected behaviors and failures, providing clear feedback on the code's functionality.

One key strategy was the consistent use of JUnit assertions to precisely verify outcomes. For example, when testing the ContactService, a test to ensure a contact name cannot be null would look like this:

@Test  
void addContact\_shouldThrowIllegalArgumentException\_whenNameIsNull() {  
 // Given: A null contact name  
 String nullName = null;  
 String phoneNumber = "1234567890";  
  
 // When & Then: Attempt to add contact and expect an exception  
 assertThrows(IllegalArgumentException.class, () -> {  
 contactService.addContact(nullName, phoneNumber);  
 }, "Adding a contact with a null name should throw IllegalArgumentException");  
}

This specific line assertThrows(IllegalArgumentException.class, () -> { ... }) ensures that the addContact method behaves as expected by throwing a IllegalArgumentException when provided with a null name, making the test technically sound in its validation. Similarly, for positive scenarios, assertEquals or assertNotNull were used to confirm returned values matched expectations.

Another aspect of technically sound code involved setting up test data correctly to avoid false positives or negatives. Utilizing mock objects (e.g., Mockito) for dependencies ensured that tests were truly isolated, preventing external system behavior from influencing unit test results.

#### **Experience Writing JUnit Tests (Efficient Code)**

Efficiency in test code was achieved by minimizing redundancy, ensuring rapid execution, and making tests easy to maintain. This was primarily accomplished through strategic use of JUnit lifecycle annotations and focusing on single responsibilities per test.

To prevent test setup duplication and ensure each test started with a clean slate, the @BeforeEach annotation was widely used. This allowed for common initialization logic to be run before every test method, making each test independent and faster to write. An example from the ContactServiceTest would be:

private ContactService contactService;  
  
@BeforeEach  
void setUp() {  
 // Initialize a fresh ContactService instance before each test  
 contactService = new ContactService();  
 // Optional: Reset any static mocks or clear test data stores if necessary  
}  
  
@Test  
void addContact\_shouldReturnNewContactId\_onSuccess() {  
 // This test leverages the clean contactService instance from setUp()  
 String contactId = contactService.addContact("John Doe", "1234567890");  
 assertNotNull(contactId);  
}

The setUp() method ensures that contactService is reinitialized before each test, preventing side effects from one test influencing another, which is crucial for efficiency and reliability. Additionally, tests were designed to be small and focused, typically testing one specific aspect or scenario. This not only makes them easier to understand and debug but also allows the test suite to run quickly, providing rapid feedback during development. For instance, one test method would check only for valid name addition, another for valid phone number, and a separate one for handling invalid characters, rather than combining all validations into a single, complex test.

### **Reflection**

#### **Testing Techniques**

During Project One, several software testing techniques were employed, while others were consciously not used, primarily due to the scope of a unit testing assignment.

Software Testing Techniques Employed:

1. Unit Testing:
   1. Characteristics: This technique involves testing the smallest testable components or units of an application in isolation. For Project One, this meant testing individual methods or functions within the Contact, Task, and Appointment services. Tests were designed to be independent, repeatable, and fast, often using mock objects or stubs to simulate the behavior of dependent components (e.g., a database layer). The focus was on verifying the correct functionality of the specific piece of code under scrutiny.
   2. Practical Uses and Implications: Unit testing is fundamental in modern software development. Its primary use is to catch bugs early in the development cycle, significantly reducing the cost of fixing them. It acts as the first line of defense, validating the building blocks of the system. For projects with complex business logic, unit tests provide immediate feedback to developers, fostering a "test-first" or "test-as-you-go" approach. It also serves as living documentation for the codebase, explaining how individual components are intended to be used. In a large microservices architecture, extensive unit testing ensures each service's core logic is sound before integration.
2. Black-box Testing (Functional Testing at Unit Level):
   1. Characteristics: Although applied at the unit level, the principles of black-box testing were used to verify that the methods performed their intended functions based on the requirements, without peering into their internal implementation details. This meant providing inputs and checking outputs against the specified functional requirements. For example, when testing addContact(), I didn't care *how* the name was stored internally, only that providing a valid name resulted in a successful addition and retrieving it yielded the correct name.
   2. Practical Uses and Implications: Black-box testing is crucial for ensuring that the software meets user expectations and functional specifications. It's often employed in later stages (integration, system, acceptance testing) but is also useful at the unit level for validating public APIs of classes. Its implication is that it helps confirm the "what" of the system: what functionalities are provided and do they work as described in the requirements. It's particularly useful when testing against requirements documents or user stories, ensuring the system delivers the expected value.
3. White-box Testing (Code Coverage and Path Testing):
   1. Characteristics: This technique involves testing based on the internal structure, design, and implementation of the software. For Project One, this was primarily applied implicitly through aiming for high code coverage (statement, branch, and sometimes path coverage) within the unit tests. By analyzing the method's internal logic, I designed test cases to exercise different paths through the code, including conditional branches and loops, to ensure all logic was tested.
   2. Practical Uses and Implications: White-box testing is invaluable for detecting internal defects, security vulnerabilities, and dead code. It's typically performed by developers or dedicated white-box testers. It ensures the "how" of the system: how the logic is implemented and whether it handles all internal conditions correctly. This is particularly important for critical algorithms, complex state machines, or security-sensitive components where thorough internal validation is paramount. It helps improve code quality, optimize performance, and reveal logical errors that might be missed by purely black-box approaches.

Other Software Testing Techniques Not Used (and their implications):

1. Integration Testing:
   1. Characteristics: This technique focuses on testing the interfaces and interactions between integrated units or components. It verifies that different modules or services work correctly when combined (e.g., the Contact Service interacting with a simulated database layer, or the Task Service correctly communicating with the Appointment Service if they had interdependencies).
   2. Practical Uses and Implications: Integration testing is vital for identifying interface defects, data flow issues, and dependency problems that unit tests cannot uncover. In a multi-service application, it ensures that services can communicate effectively and that data is passed correctly between them. It's a critical step before full system testing, reducing integration headaches and ensuring that the combined components meet higher-level functional requirements.
2. System Testing:
   1. Characteristics: This involves testing the complete and integrated software system to evaluate its compliance with the specified requirements. It's a holistic test of the entire application, often simulating real-world scenarios. It checks both functional and non-functional requirements (e.g., performance, security, usability).
   2. Practical Uses and Implications: System testing provides confidence that the entire application functions as intended from an end-to-end perspective. It's typically performed after integration testing and before acceptance testing. It's crucial for validating that the software meets the overall business objectives and user needs, ensuring all components work harmoniously in a complete environment. For large-scale enterprise applications, system testing is indispensable for verifying complex workflows and ensuring stability.
3. Acceptance Testing (User Acceptance Testing - UAT):
   1. Characteristics: This is the final stage of testing, where the software is tested by the end-users or clients to verify that it meets their business needs and requirements. It ensures the system is fit for purpose and ready for deployment.
   2. Practical Uses and Implications: UAT is paramount for validating that the software delivers actual business value and meets the client's expectations. It reduces the risk of delivering a product that, while technically sound, doesn't solve the user's real-world problems. For custom software development projects, UAT is the gatekeeper for deployment, as client sign-off is often a prerequisite. It fosters trust and ensures client satisfaction.
4. Performance Testing (e.g., Load, Stress Testing):
   1. Characteristics: These techniques evaluate the system's responsiveness, stability, scalability, and resource usage under various loads. Load testing assesses behavior under expected user loads, while stress testing pushes the system beyond its normal operational limits to see how it behaves under extreme conditions.
   2. Practical Uses and Implications: Performance testing is critical for applications expecting a large number of concurrent users or high transaction volumes (e.g., e-commerce platforms, streaming services). It helps identify bottlenecks, optimize resource allocation, and ensure the application remains responsive and stable under pressure. Failure to perform adequate performance testing can lead to system crashes, slow response times, and a poor user experience, resulting in lost revenue or user dissatisfaction.
5. Security Testing:
   1. Characteristics: This technique aims to identify vulnerabilities and weaknesses in the software that could lead to unauthorized access, data breaches, or other security threats. It involves various methods, including penetration testing, vulnerability scanning, and code review for security flaws.
   2. Practical Uses and Implications: Security testing is non-negotiable for any application handling sensitive data or exposed to the internet. Its implication is the protection of user data, company reputation, and compliance with regulations (e.g., GDPR, HIPAA). For financial applications, healthcare systems, or any platform storing personal identifiable information (PII), robust security testing is paramount to prevent costly breaches and maintain user trust.

#### **Mindset**

Caution:

As a software tester in Project One, employing caution was paramount. It involved anticipating every conceivable scenario, especially the "unhappy paths" and edge cases that developers might inadvertently overlook during implementation. This mindset was crucial for appreciating the complexity and interrelationships of the code even at the unit level.

For instance, while unit testing the updateContact() method, caution dictated not only testing successful updates but also considering:

* What happens if the contactId provided doesn't exist? (Expected: ContactNotFoundException).
* What if the new name is valid, but the new phone number is invalid? (Expected: IllegalArgumentException for the phone number, without corrupting the contact).
* What if concurrent requests tried to update the same contact? (Though harder to test at a pure unit level, this mental model of interrelationships informs design of integration tests).

This cautious approach ensured that even though individual methods were isolated, I was always mindful of their place within the larger system. Overlooking these complex interactions, even conceptually at the unit level, could lead to subtle bugs that manifest only when different parts of the application interact in unexpected ways. For example, if the addAppointment() method didn't rigorously check for overlaps, it could lead to double-bookings which would then cause issues in the getAvailableSlots() method, demonstrating how a flaw in one unit can cascade and impact the functionality of related units.

Bias:

Limiting bias in the review of the code was a conscious effort. As a software engineer, there's an inherent tendency to believe one's own code is correct, especially when the logic seems straightforward. To mitigate this:

* Adherence to Requirements: My primary reference was the detailed software requirements document, not my own assumptions about how the feature "should" work. This external standard helped maintain objectivity.
* Boundary Value Analysis and Equivalence Partitioning: Applying these techniques helped ensure that tests covered a wide range of inputs, including valid, invalid, and boundary conditions, rather than just "typical" inputs. This forced a structured approach that challenged my own development-time assumptions.
* Peer Review of Test Cases (Self-simulated): While this was a solo project, I would conceptually review my own test cases as if I were a peer, asking questions like, "What if this input was null?", "What if this was the maximum allowed length?", or "Have I considered all error conditions?"

On the software developer side, if I were responsible for testing my own code, bias would absolutely be a significant concern. A developer might:

* Overlook "Obvious" Bugs: What seems obvious during development might be a blind spot during testing because the developer knows *how* the code is supposed to work and might not try to break it.
* Skip Edge Cases: Developers often test the "happy path" and common scenarios. They might inadvertently omit testing obscure error conditions or rare edge cases if they didn't explicitly consider them during coding. For example, a developer might not test the deleteContact() method with a contactId that contains special characters if they assume the addContact() method would prevent such an ID from being created in the first place.

This inherent bias underscores the value of independent testing, code reviews, and structured testing methodologies to ensure comprehensive coverage and objective evaluation.

Discipline:

Evaluating the importance of being disciplined in my commitment to quality as a software engineering professional highlights its critical role. It is paramount not to cut corners when it comes to writing or testing code because doing so directly contributes to technical debt. Technical debt is the metaphorical "interest" you pay later for taking shortcuts now. It manifests as:

* Increased Bug Count: Untested code paths become breeding grounds for bugs that only surface in production, leading to customer dissatisfaction, urgent fixes, and reputational damage.
* Higher Maintenance Costs: Code without adequate tests is difficult to refactor or modify safely. Changes introduce new bugs, and maintaining the system becomes a high-risk, time-consuming endeavor.
* Slower Feature Development: As technical debt accumulates, developers spend more time debugging and fixing old issues than building new features, significantly slowing down the pace of innovation.

For example, deciding to skip a unit test for handling an invalid date format in the scheduleAppointment() method because "it's probably handled by the UI" is a shortcut. This creates technical debt. Later, if another system or a direct API call bypasses the UI, that invalid date could crash the backend.

To avoid technical debt as a practitioner, I plan to adhere to the following disciplines:

* Test-Driven Development (TDD): Whenever feasible, writing tests *before* writing the production code. This forces a clear understanding of requirements and a testable design from the outset.
* Comprehensive Unit Testing: Maintaining high unit test coverage with meaningful assertions for all new code and critical existing code paths.
* Automated Testing: Prioritizing the automation of unit, integration, and regression tests to enable continuous feedback and prevent regressions.
* Regular Code Reviews: Participating in and conducting thorough code reviews, including reviews of test code, to catch issues early and share knowledge.
* Continuous Integration/Continuous Delivery (CI/CD): Integrating code changes frequently into a shared repository, running automated tests, and deploying to production automatically to detect integration issues early.
* Proactive Refactoring: Regularly refactoring code and tests to keep them clean, maintainable, and understandable, preventing the "code rot" that often accompanies technical debt.

By embracing these disciplines, I aim to contribute to a codebase that is robust, maintainable, and adaptable, ensuring long-term success for the software and the business it serves.