



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

🚀 Day 28

Testing Strategies

(Part 2 of 2)

Advanced Testing Techniques: Testing Strategies
(Beyond Basics) for Flawless Software 🚀



Esmaeeil Moradi

@esmaeeil-moradi700

Day 28



of #30DayAndroidChallenge:

Advanced Testing Techniques: Testing Strategies (Beyond Basics) for Flawless Software 🚀

Hey Android Architects, ambitious developers, and anyone who values the stability and quality of their code!

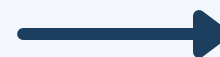
Did you learn the basics of testing? Excellent! But are you ready to unlock the secret tools of a system architect for building applications with unparalleled quality and endless scalability? This isn't just about finding bugs; it's about building a system that should never have bugs!

If yesterday you became familiar with the fundamentals of Unit, Integration, and UI Testing, today we step into a realm known only to architects of powerful systems. In today's world, where software has become the beating heart of industries and the slightest error can lead to irreparable damage, basic tests are no longer sufficient. The Cost of Bugs in Production, Security implications, and Scalability challenges in large projects compel us to go beyond superficial testing.



Esmaeeil Moradi

@esmaeeil-moradi700



This article is your roadmap to Test Excellence and mastering advanced testing.

We will introduce you to tools that help system architects write code that not only works but is also secure against attacks, maintainable at scale, and resilient for the future.

Especially in critical ecosystems like Web3 and Decentralized Finance (DeFi) platforms, where every minor flaw can be catastrophic, mastering advanced testing strategies is no longer just an advantage—it's an engineering and security imperative.



Esmaeel Moradi

@esmaeel-moradi700

Deeper into Test Doubles: Beyond Simple Mockito (Mock, Spy, Stub, Fake)

Description:

Test Doubles are substitutes for real dependencies during testing. Choosing the correct Test Double is crucial for tests to be accurate, fast, and maintainable.

- **Dummy**: An object passed around but never actually used. Only serves to fill argument lists.
- **Fake**: A simplified, lightweight implementation of a dependency (e.g., an InMemory implementation of a database).
- **Stub**: A Fake that provides pre-configured answers to specific method calls.
- **Mock**: A Stub that, in addition to returning answers, also Verifies that specific methods were called. Used for interaction testing.
- **Spy**: A real object that is monitored, and some of its methods can be Stubbed. Unlike a Mock, which is a completely fabricated object, a Spy operates on real objects.



Esmaeel Moradi

@esmaeel-moradi700

Why?

Selecting the correct Test Double is essential to ensure the true isolation of Unit Tests, reduce dependency on slow or unstable services (like networks and databases), and improve the efficiency and maintainability of tests at scale. In Mission-Critical systems, where the precise behavior of every component is vital, meticulous control over dependencies with advanced Test Doubles minimizes human error.

Advanced Code Examples:

Demonstrating the use of Mockito.spy(), Mockito.verify(), Argument Captors and their differences from Mockito.mock().

```
// Example: Service to be tested
class DataProcessor(private val logger: Logger) {
    fun process(data: String): String {
        logger.log("Processing: $data")
        return data.toUpperCase()
    }
}

// Example: Logger interface
interface Logger {
    fun log(message: String)
}
```



Esmaeel Moradi
@esmaeeil-moradi700

Using Mockito.mock() for behavior verification (interaction testing):



```
import org.junit.Before
import org.junit.Test
import org.mockito.ArgumentCaptor
import org.mockito.Mockito.* // Import all static methods for brevity

class DataProcessorWithMockTest {
    private lateinit var mockLogger: Logger
    private lateinit var dataProcessor: DataProcessor

    @Before
    fun setup() {
        mockLogger = mock(Logger::class.java) // Create a mock logger
        dataProcessor = DataProcessor(mockLogger)
    }

    @Test
    fun `process method logs the data and returns uppercase`() {
        val testData = "hello"
        val expectedResult = "HELLO"

        // When
        val result = dataProcessor.process(testData)

        // Then
        // Verify that logger.log was called exactly once with the expected message
        verify(mockLogger, times(1)).log("Processing: $testData")

        // Use ArgumentCaptor to verify the exact argument passed to log
        val captor = ArgumentCaptor.forClass(String::class.java)
        verify(mockLogger).log(captor.capture())
        assert(captor.value == "Processing: $testData")

        assert(result == expectedResult)
    }
}
```



Esmaeel Moradi
@esmaeeil-moradi700

Using Mockito.spy() for partial mocking on a real object:

A Spy is useful when you have a real object but want to temporarily alter the behavior of some of its methods or simply monitor specific method calls on that real object.

```
import org.junit.Assert.assertEquals
import org.junit.Before
import org.junit.Test
import org.mockito.Mockito.*

// A real logger implementation
class ConsoleLogger : Logger {
    var loggedMessages = mutableListOf<String>()
    override fun log(message: String) {
        println("Console Log: $message")
        loggedMessages.add(message) // For internal inspection
    }
}

class DataProcessorWithSpyTest {
    private lateinit var realLogger: ConsoleLogger // Real object
    private lateinit var spyLogger: Logger // Spy on the real object
    private lateinit var dataProcessor: DataProcessor

    @Before
    fun setup() {
        realLogger = ConsoleLogger()
        spyLogger = spy(realLogger) // Create a spy on the real logger
        dataProcessor = DataProcessor(spyLogger)
    }

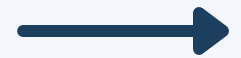
    @Test
    fun `spy can verify real method calls`() {
        val testData = "world"
        dataProcessor.process(testData)

        // Verify that the real log method was called
        verify(spyLogger).log("Processing: $testData")
        // You can also assert on the real object's state if needed
        assertEquals(1, realLogger.loggedMessages.size)
        assertEquals("Processing: $testData", realLogger.loggedMessages[0])
    }
}
```



Esmaeel Moradi

@esmaeeil-moradi700



```
@Test
fun `spy can stub specific real methods`() {
    val testData = "stubbed"
    // Stubbing a method on the real object
    doNothing().`when`(spyLogger).log(anyString()) // Prevent actual logging f

    dataProcessor.process(testData)

    // Verify that the method was still called, even if stubbed
    verify(spyLogger).log("Processing: $testData")
    // The real method was stubbed, so realLogger's state should not change
    assertEquals(0, realLogger.loggedMessages.size)
}
```

These techniques enable architects to test different layers of the system with unparalleled precision, even in scenarios involving complex or inaccessible dependencies.



Esmaeel Moradi
@esmaeel-moradi700

Test-Driven Development (TDD) & Behavior-Driven Development (BDD): The Philosophy of Building Flawless Code

Description:

Test-Driven Development (TDD) and Behavior-Driven Development (BDD) are more than just writing tests; they are philosophies for software development built on rapid feedback and clear definition of system behavior.

- **TDD (Test-Driven Development):**
 - **The Red-Green-Refactor Cycle:** First, you write a failing test (Red) that demonstrates a new feature or a bug. Then, you write the minimum code possible to make the test pass (Green). Finally, you refactor both the code and the test for improved design and readability (Refactor).
 - **Benefits:** Improved code design, reduced bugs, increased confidence, self-documenting code, and fast feedback.



Esmaeel Moradi
@esmaeel-moradi700

- **BDD (Behavior-Driven Development):**



- **Focus on Behavior:** Unlike TDD, which focuses on technical implementation, BDD emphasizes the system's behavior from the perspective of the business and the end-user.
- **Gherkin Language (Given-When-Then):** Behavioral scenarios are written in a human-readable format understandable by non-technical stakeholders. (Example: Given the user is on the login screen, When they enter valid credentials, Then they should be redirected to the dashboard.)
- **Tools:** Tools like Cucumber translate these scenarios into executable tests.

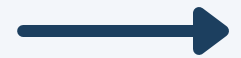
Why?

TDD and BDD help build higher-quality, more understandable code that aligns with real requirements. They empower teams to focus on what truly matters (correct behavior and business value). In large teams, BDD improves collaboration among developers, QA, and business stakeholders, as everyone shares a common language for defining expectations. These approaches are crucial for complex systems that require precise coordination among teams and high accuracy in business logic, especially in financial and blockc



Esmaeel Moradi

@esmaeel-moradi700



Example (Conceptual/Pseudo-code):

TDD Scenario (Adding a new feature:
Calculator can handle negative results):

1- Red (Write failing test):

```
// CalculatorTest.kt
@Test
fun `subtract returns negative difference when result is negative`() {
    val result = calculator.subtract(2, 5)
    assertEquals(-3, result) // This test will fail initially
}
```

2- Green (Write minimum code to pass):

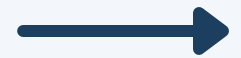
```
// Calculator.kt
class Calculator {
    fun add(a: Int, b: Int) = a + b
    fun subtract(a: Int, b: Int) = a - b // No change needed, it already works
}
```

3- Refactor (Improve code and test):

(If necessary, e.g., improve variable names, simplify logic)



Esmaeel Moradi
@esmaeel-moradi700



BDD Scenario

(User can send cryptocurrency to a valid address):

Feature: Cryptocurrency Transaction

As a crypto wallet user

I want to send cryptocurrency

So that I can transfer assets to others

Scenario: User sends valid amount to a valid address

Given I am logged into my wallet with a balance of 10.0 ETH

And I am on the "Send" screen

And I have entered a valid recipient address "0x123...abc"

When I enter "1.0" as the amount to send

And I tap the "Send" button

Then I should see a "Transaction Confirmed" message

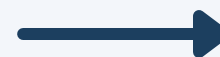
And my balance should be 9.0 ETH

And the transaction should appear in my transaction history



Esmaeel Moradi

@esmaeel-moradi700



These philosophies drive teams towards building code that is inherently reliable and perfectly aligned with precise business requirements.



Esmaeeil Moradi
@esmaeeil-moradi700

Smart Tests: Mutation Testing & Fuzz Testing (A Glimpse into the Future of Testing)

Description:

These techniques go beyond traditional testing to uncover hidden bugs and vulnerabilities that might remain undetected by conventional tests.

- **Mutation Testing:**

- **Concept of "Mutations" in Code:** This technique automatically introduces small changes (mutations) into your production code (e.g., changing `a + b` to `a - b` or `if(x > y)` to `if(x < y)`).
- **Goal:** Your existing tests are then run against these mutated code versions. If a test Fails (Red), it means your test is robust enough to detect this small change (the mutation is "killed"). If the test remains Green, it indicates insufficient test coverage for that part of the code.
- **How it Works (e.g., with Pitest):** Tools like Pitest for Java/Kotlin automate this process.



Esmaeel Moradi

@esmaeel-moradi700



- **Fuzz Testing (or Fuzzing):**
 - **Concept:** Feeding invalid, unexpected, random, or specially formatted input data (usually in large volumes) to program inputs to find bugs, Crashes, memory leaks, or security vulnerabilities.
 - **Application in Security:** Particularly effective in discovering security flaws in parsers, protocols, or input processing systems.

Why?

These techniques go beyond simple line coverage and help find hidden bugs and security holes that traditional tests might miss. They assure you that your tests genuinely cover the code's behavior, not just its lines. For Mission-Critical systems, especially those handling sensitive data (like cryptocurrency wallets), Mutation Testing and Fuzz Testing are essential to ensure that no malicious input can compromise the system or cause unpredictable behavior.



Esmaeel Moradi

@esmaeel-moradi700

Example (Conceptual):



Mutation Testing Scenario:

Suppose you have a function `isAdult(age: Int)` that returns `true` if `age >= 18`. A mutation tool might change `age >= 18` to `age > 18`. If your existing test for `isAdult(18)` passes, it means your test is not strong enough to "kill" this mutation, indicating a potential gap in your test suite for edge cases.

Fuzz Testing Scenario:

For a cryptocurrency address validator function: `isValidAddress(address: String)` Fuzzing would feed it millions of random, malformed, or extremely long strings (e.g., `"0xGHT123"`, `""`, `"0x"` repeated 1000 times, strings with special characters) to see if it crashes, throws an unexpected exception, or incorrectly validates an invalid address. This is crucial for wallet security.



Esmaeel Moradi

@esmaeeil-moradi700

Testing in CI/CD Pipeline & Modular Architecture:

Scalability and Automation



Description: In large projects with multiple teams and continuously changing codebases, automation and testing strategy are vital.

- **Role of Testing in CI/CD (Continuous Integration/Continuous Delivery):**
 - Explaining how to integrate Unit, Integration, and UI tests automatically into the CI/CD pipeline. Every time code is added to the repository, tests are automatically run, providing rapid feedback (Fail/Pass).
 - **Tools:** Mentioning tools like Jenkins, GitLab CI, GitHub Actions that facilitate this automation.
- **Testing in Modular Architecture:**
 - How modular architecture (e.g., modular features in Android or Feature Modules) significantly enhances testability. Each module can have its independent tests, and changes in one module only affect its tests and dependent modules.
 - Addressing inter-module testing challenges with specific Integration Tests for connection points.



Esmaeel Moradi

@esmaeel-moradi700



- **Advanced End-to-End Tests:**

- For more complex scenarios requiring interaction with multiple applications, external services, or even physical devices.
- **Tools (Conceptual Mention):** Using tools like Maestro (for Compose and View-based UI tests), Appium (for cross-platform and out-of-app testing) to simulate complete user flows.

Why? Test automation in CI/CD drastically reduces the time to identify bugs and allows teams to release code with greater confidence. Modular architecture combined with proper testing makes managing testing complexity in massive projects feasible. For large Web3 platforms with multiple components, these are operational principles that ensure the integrity and security of the entire system.



Esmaeel Moradi
@esmaeel-moradi700



Beyond Functionality: Non-Functional Testing – Performance & Accessibility

Description: Non-functional tests focus on the "quality" attributes of a system, not just its correct operation.

- **Performance Testing:**

- **Introduction to core concepts:** loading speed, UI responsiveness, resource consumption (CPU, RAM, Battery), and stability under load.
- **Tools:** Using AndroidX Benchmark for automated code performance tests on real devices, and Profiler in Android Studio for identifying performance bottlenecks and memory leaks.
- **Why?**
- In heavy applications, applications with many users, or those handling large data volumes (e.g., DeFi dashboards), poor performance translates to a terrible user experience and user churn.



Esmaeel Moradi
@esmaeel-moradi700



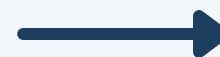
- **Accessibility Testing:**

- **Importance:** The significance of designing and testing for users with special needs (visual, auditory, motor impairments). This is a legal and ethical requirement in many countries.
- **Tools:** Using Accessibility Scanner (an Android application) for automated UI scanning and problem identification, and automated testing frameworks to verify compliance with accessibility standards.
- **Why?** These tests ensure your product is inclusive and usable by everyone. In competitive and performance-sensitive markets, these tests are crucial for a complete and successful product, demonstrating commitment to all users.



Esmaeel Moradi

@esmaeel-moradi700



Conclusion:

Build a Legacy of Quality with Advanced Testing!

Advanced testing not only finds bugs but also instills a culture of quality within the team and the product. It's an investment in a future where your applications are not just functional but also reliable, secure, and ready for continuous scalability and growth.

This article is just the beginning of your journey into the complex and rewarding world of advanced testing. As an Android Architect, I invite you to apply these approaches in your projects and join in building the next generation of critical and secure Android applications (especially in the Web3 domain). With this knowledge, you are no longer just a coder, but an architect of security and stability. Stay tuned for the next articles in the



Esmaeel Moradi

@esmaeel-moradi700

Looking forward to your
insights and experiences!
Let's elevate Android
development together.



Esmaeeil Moradi

@esmaeeil-moradi700