Fundamentals of Testing

Users interact with your app on a variety of levels, from pressing a **Submit** button to downloading information onto their device. Accordingly, you should test a variety of use cases and interactions as you iteratively develop your app.

Use an iterative development workflow

As your app expands, you might find it necessary to fetch data from a server, interact with the device's sensors, access local storage, or render complex user interfaces. The versatility of your app demands a comprehensive testing strategy.

When developing a feature iteratively, you start by either writing a new test or by adding cases and assertions to an existing unit test. The test fails at first because the feature isn't implemented yet.

It's important to consider the units of responsibility that emerge as you design the new feature. For each unit, you write a corresponding unit test. Your unit tests should nearly exhaust all possible interactions with the unit, including standard interactions, invalid inputs, and cases where resources aren't available.

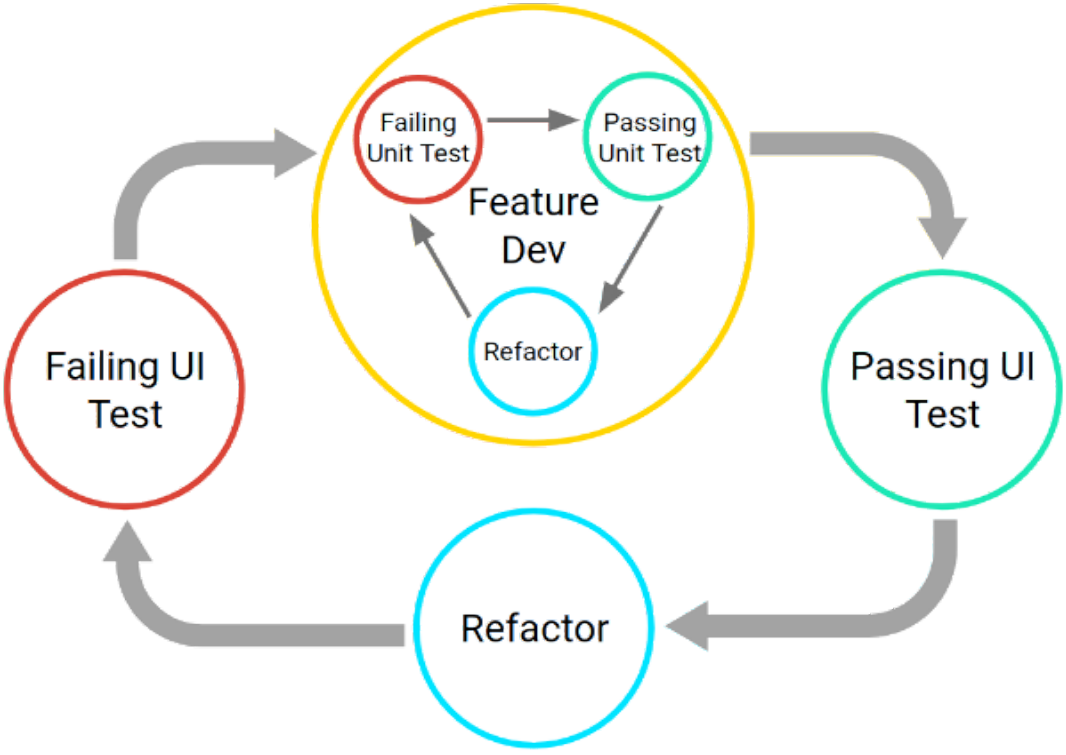

Figure 1. The two cycles associated with iterative, test-driven development

The full workflow, as shown in Figure 1, contains a series of nested, iterative cycles where a long, slow, UI-driven cycle tests the integration of code units. You test the units themselves using shorter, faster development cycles. This set of cycles continues until your app satisfies every use case.

Understand the Testing Pyramid

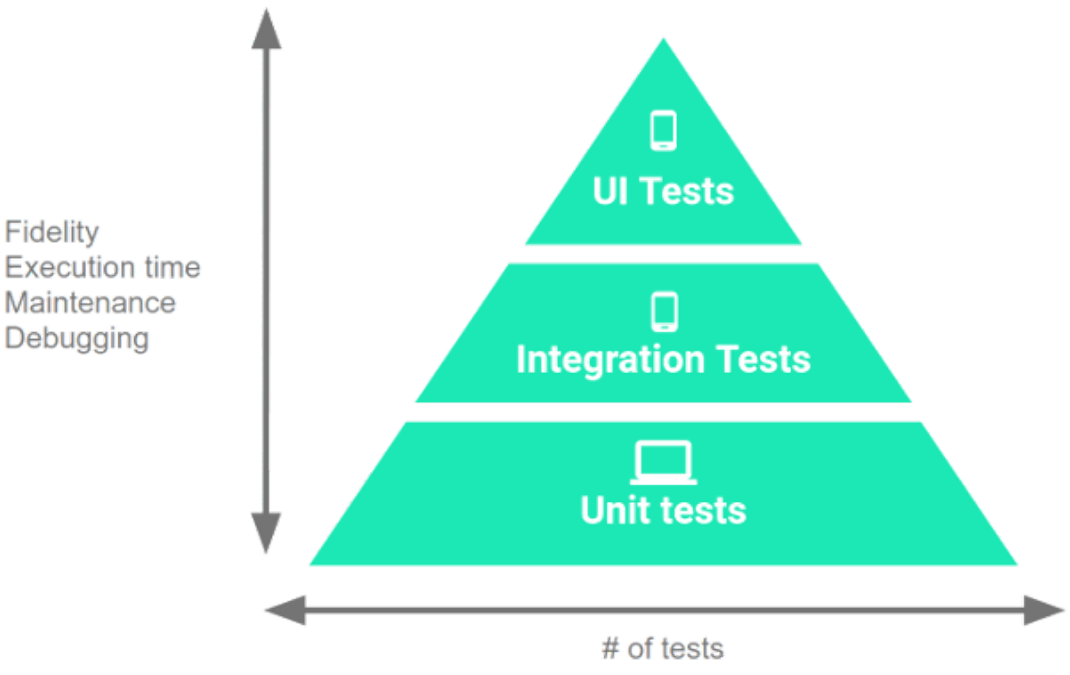

Figure 2. The Testing Pyramid, showing the three categories of tests that you should include in your app's test suite

The Testing Pyramid, shown in Figure 2, illustrates how your app should include the three categories of tests: small, medium, and large:

- Small tests are unit tests that you can run in isolation from production systems. They typically mock every major component and should run quickly on your machine.
- Medium tests are integration tests that sit in between small tests and large tests. They integrate several components, and they run on emulators or real devices.
- Large tests are integration and UI tests that run by completing a UI workflow. They ensure that key end-user tasks work as expected on emulators or real devices.

Although small tests are fast and focused, allowing you to address failures quickly, they're also low-fidelity and self-contained, making it difficult to have confidence that a passing test allows your app to work. You encounter the opposite set of tradeoffs when writing large tests.

Because of the different characteristics of each test category, you should include tests from each layer of the test pyramid. Although the proportion of tests for each category can vary based on your app's use cases, we generally recommend the following split among the categories: **70 percent small, 20 percent medium, and 10 percent large**.

To learn more about the Android Testing Pyramid, see the Test-Driven Development on Android session video from Google I/O 2017, starting at 1:51.

Write small tests

As you add and change your app's functionality, make sure that these features behave as intended by creating and running unit tests against them. Although it's possible to evaluate units on a device or emulator, it's usually quicker and easier to test the units in your development environment, adding stubbed or mocked methods as needed to interact with the Android system.

To learn about the characteristics of a well-defined small test, see the Test-

Driven Development on Android session video from Google I/O 2017, starting at 6:49.

Robolectric

If your app's testing environment requires unit tests to interact more extensively with the Android framework, you can use Robolectric. This tool executes testing-friendly, Java-based logic stubs that emulate the Android framework. The community maintains these stubs.

Robolectric tests nearly match the full fidelity of running tests on an Android device while still executing more quickly than on-device tests. It also supports the following aspects of the Android platform:

- Android 4.1 (API level 16) and higher
- Android Gradle Plugin version 2.4 and higher
- Component lifecycles
- Event loops
- All resources

Note: Robolectric has its own set of testing APIs and introduces some new concepts. For more information about integrating Robolectric's APIs with your app's tests, see the tool's user guide, as well as the Test-Driven Development on Android session video from Google I/O 2017, starting at 12:40.

Mock objects

You can monitor the elements of the Android framework with which your app interacts by running unit tests against a modified version of `android.jar`. This JAR file doesn't contain any code, so your app's calls to the Android framework throw exceptions by default. To test elements of your code that interact with the Android system, configure mock objects using a framework like Mockito.

If your code contains references to resources or complex interactions with the Android framework, you should use a different form of unit testing instead, such as Robolectric.

Instrumented unit tests

You can also run instrumented unit tests on a physical device or emulator, which doesn't involve any mocking or stubbing of the framework. Because this form of testing involves significantly slower execution times than local unit tests, however, it's best to rely on this method only when it's essential to evaluate your app's behavior against actual device hardware.

Write medium tests

After you've tested each unit of your app within your development environment, you should verify that the components behave properly when run on an emulator or device. Medium tests allow you to complete this part of the development process. These tests are particularly important to create and run if some of your app's components depend on physical hardware.

Medium tests evaluate how your app coordinates multiple units, but they don't test the full app. Examples of medium tests include service tests, integration tests, and hermetic UI tests that simulate the behavior of external dependencies.

Typically, it's better to test your app on an emulated device or a cloud-based service like Firebase Test Lab, rather than on a physical device, as you can test multiple combinations of screen sizes and hardware configurations more easily and quickly.

To learn more about how to write medium tests, see the Test-Driven Development on Android session video from Google I/O 2017, starting at 31:55.

Write large tests

Although it's important to test each layer and feature within your app in isolation, it's just as important to test common workflows and use cases that involve the complete stack, from the UI through business logic to the data layer.

If your app is small enough, you might need only one suite of large tests to evaluate your app's functionality as a whole. Otherwise, you should divide your large test suites by team ownership, functional verticals, or user goals.

Note: For each large, workflow-based test that you write, you should also write medium tests that check the functionality of each UI component included in the workflow. That way, your test suite can continue to identify potential issues within each step of a critical user journey, even when the corresponding large test keeps failing during one of the first few steps.

To learn more about how to write large tests, see the Test-Driven Development on Android session video from Google I/O 2017, starting at 30:38.

The `AndroidJUnitRunner` class defines an instrumentation-based JUnit test runner that lets you run JUnit 3- or JUnit 4-style test classes on Android devices. The test runner facilitates loading your test package and the app under test onto a device or emulator, running your tests, and reporting the results.

The `AndroidJUnitRunner` class also supports the following tools and frameworks from the Android Testing Support Library (ATSL):

JUnit4 Rules

ATSL includes code for managing the lifecycles of key app components involved in your tests, such as activities and services. To learn how to define these rules, see the JUnit4 Rules guide.

Espresso

Espresso synchronizes asynchronous tasks while automating the following in-app interactions:

- Performing actions on `View` objects.
- Completing workflows that cross your app's process boundaries. *Available only on Android 8.0 (API level 26) and higher.*
- Assessing how users with accessibility needs can use your app.
- Locating and activating items within `RecyclerView` and `AdapterView` objects.
- Validating the state of outgoing intents.
- Verifying the structure of a DOM within `WebView` objects.
- Tracking long-running background operations within your app.

To learn more about these interactions and how to use them in your app's tests, see the Espresso guide.

UI Automator

Caution: We recommend testing your app using UI Automator only when your app must interact with the system to fulfill a critical use case. Because UI Automator interacts with system apps and UIs, you need to re-run and fix your UI Automator tests after each system update. Such updates include Android platform version upgrades and new versions of Google Play services.

As an alternative to using UI Automator, we recommend adding hermetic tests or separating your large test into a suite of small and medium tests. In particular, focus on testing one piece of inter-app communication at a time, such as sending information to other apps and responding to intent results. The Espresso-Intents tool can help you write these smaller tests.

The UI Automator framework performs interactions within system apps on your app's behalf, such as inspecting the hierarchy of the currently-shown UI, taking screenshots, and analyzing the device's current state. For more details about how the UI Automator can observe an app under test, see the UI

Automator guide.

Android Test Orchestrator

Android Test Orchestrator runs each UI test in its own `Instrumentation` sandbox, increasing your test suite's reliability by reducing shared state between tests and isolating app crashes on a per-test basis.

For more information about the benefits that Android Test Orchestrator provides as you test your app, see the Android Test Orchestrator guide, as well as the Test-Driven Development on Android session video from Google I/O 2017, starting at 23:58.