Testing

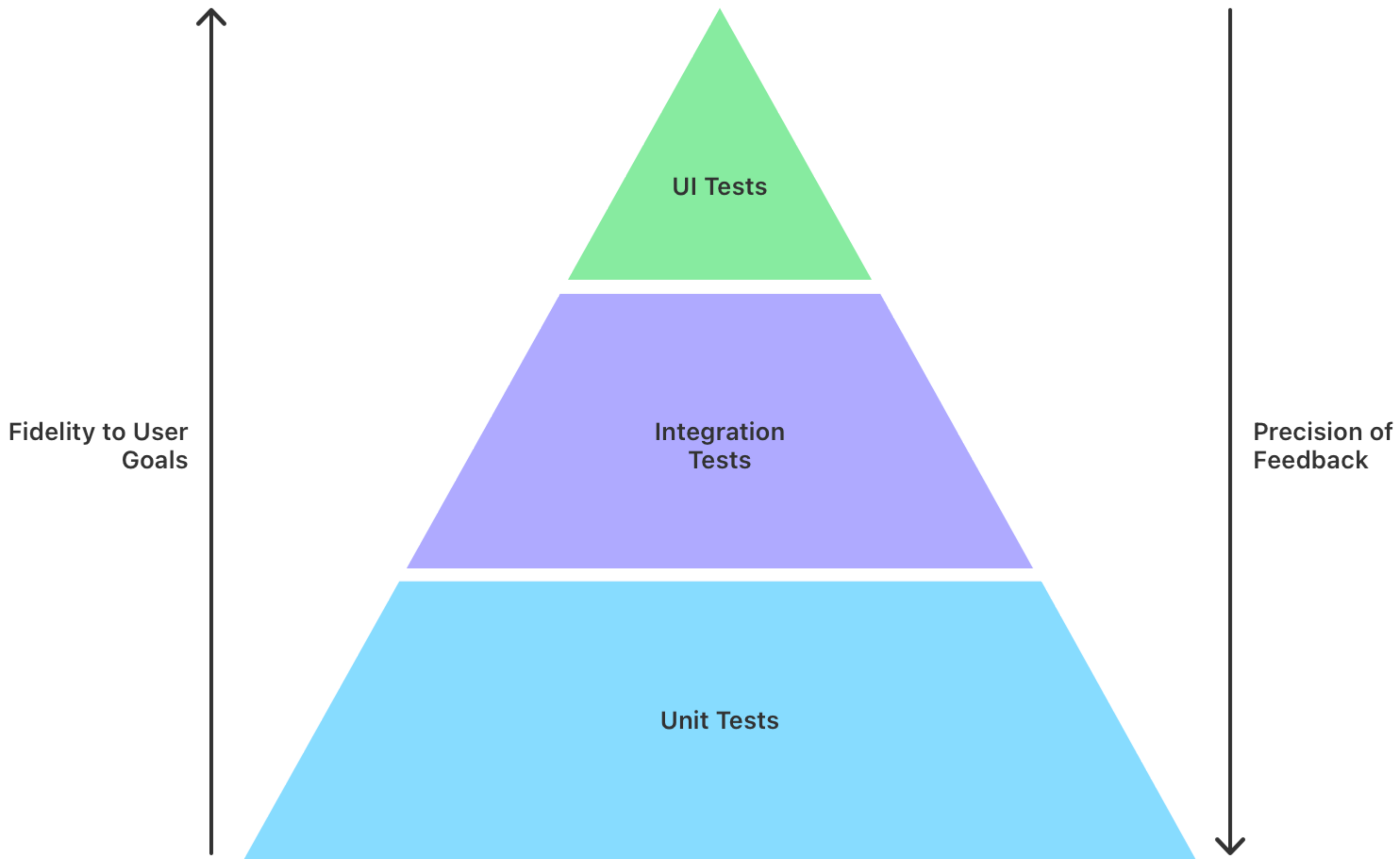
Develop and run tests to detect logic failures, UI problems, and performance regressions.

Overview

An important part of software development is testing your code. To catch issues early and deliver the highest quality products, use the frameworks and features Xcode provides to develop tests, understand code coverage, and evaluate your test results.

Xcode 16 and later includes Swift Testing, a new testing framework you can use to write unit tests that takes advantage of the powerful and expressive language capabilities of Swift with output that is expressive and actionable. Xcode continues to include XCTest to facilitate writing UI tests that control your app's UI using XCUIAutomation. A good testing strategy combines multiple types of tests, to maximize the benefits of each.

Aim for a "pyramid" distribution of tests, as shown in the figure below. Include a large number of fast, well-isolated unit tests to cover your app's logic, a smaller number of integration tests to demonstrate that smaller parts connect together properly, and UI tests to assert the correct behavior of common use cases.



UI tests are the ultimate indicator your app works the way you expect, but they take longer to run than other kinds of tests. There are various app variables that can introduce a failure in the same UI test. The test pyramid balances high-fidelity tests that demonstrate that people can complete their tasks, with tightly-focused tests that give you fast feedback about the correctness of your app's logic and the impact of changes you make.

In addition to the test pyramid, write performance tests to provide regression coverage of performance-critical regions of code. To learn about the process of identifying performance-critical code, see Improving your app's performance.

Topics

Test development

Adding tests to your Xcode project

Include test targets that build code to test the logic in your functions, check for integration issues, automate UI workflows, and measure performance.

Updating your existing codebase to accommodate unit tests

Remove coupling between components to increase test coverage and reliability.

Determining how much code your tests cover

Use code coverage to focus new test development on areas that lack adequate testing.

Improving code assessment by organizing tests into test plans

Control the information you receive from your tests at different stages in the software engineering process by creating and configuring test plans.

Execution and results

Running tests and interpreting results

Determine whether your project’s code behaves as you expect by running tests and understanding the results.

Performance tests

Writing and running performance tests

Repeatably gather metrics on the performance of your code.

Location

Simulating location in tests

Improve test reliability and coverage when working with location-based code.

StoreKit

Setting up StoreKit Testing in Xcode

Prepare your test environment to test in-app purchases with data you configure locally.

Testing in-app purchases with StoreKit transaction manager in Xcode

Use the transaction manager within Xcode to test in-app purchases without requiring a connection to App Store servers.

See Also

Tuning and debugging

Devices and Simulator

Configure and manage devices connected to your Mac or devices in Simulator and use them to run your app.

Debugging

Identify and address issues in your app using the Xcode debugger, Xcode Organizer, Metal debugger, and Instruments.

Performance and metrics

Measure, investigate, and address the use of system resources and issues impacting performance using Instruments and Xcode Organizer.