Automated Testing Documentation for Financial Tracker App

Developer: Roy T. Dimapilis
Date: November 26, 2025
Course: MDI 116 - iOS App Development

Introduction

Testing is one of the most important parts of building a quality app. Without proper testing, bugs and errors can slip through and cause problems for users. Automated testing allows developers to write code that checks if other code is working correctly, and these tests can be run automatically whenever changes are made to the app. This documentation covers the automated testing practices and exercises completed for the Financial Tracker app, including unit tests that check individual pieces of code and user interface tests that simulate real user interactions with the app.

Course Overview

This course helped me with the fundamentals of automated testing for iOS applications. The main goals were to understand why testing is important, learn how to write effective tests, and implement testing practices that ensure app quality and reliability. Throughout the course, I learned about different types of testing including unit testing for checking individual functions and components, user interface testing for verifying that screens and buttons work correctly, and performance testing for measuring how fast the app responds. The course also covered test-driven development, which is a practice where you write tests before writing the actual code, helping to ensure that code is designed properly from the start.

Exercises

Exercise 1: Introduction to Automated Testing

In this exercise, I learned the basics of automated testing and set up the testing environment for my Financial Tracker app. The first step was understanding the difference between manual testing, where a person clicks through the app to check if things work, and automated testing, where code does the checking automatically. I created a UI testing file in Xcode called FinancialTrackerUITests that uses Apple's XCTest framework. This framework provides tools for writing test functions that can verify if parts of the app are working correctly. I learned about

important concepts like assertions, which are statements that check if something is true, and test methods, which are functions that contain the testing code. The setup and teardown methods were also introduced, which run before and after each test to prepare the testing environment and clean up afterward.

Exercise 2: Test-Driven Development (TDD) and Unit Testing

This exercise focused on unit testing and the test-driven development approach. Unit tests are designed to test small, individual pieces of code like functions and classes to make sure they work correctly on their own. For my Financial Tracker app, I created unit tests for several important components. I wrote tests for the Transaction model to verify that transactions are created correctly with the right title, amount, and income or expense status. I also tested the User model to make sure user profiles are created properly with names, emojis, and optional photos. The StoreManager class was tested extensively to verify that the trial version limits work correctly, allowing only one user profile and five transactions for free users while giving unlimited access to pro users. I also wrote tests for the Currency Converter to ensure that currency formatting and conversion between different currencies works accurately. The test-driven development approach taught me to think about what the code should do before actually writing it, which helps create better designed and more reliable code.

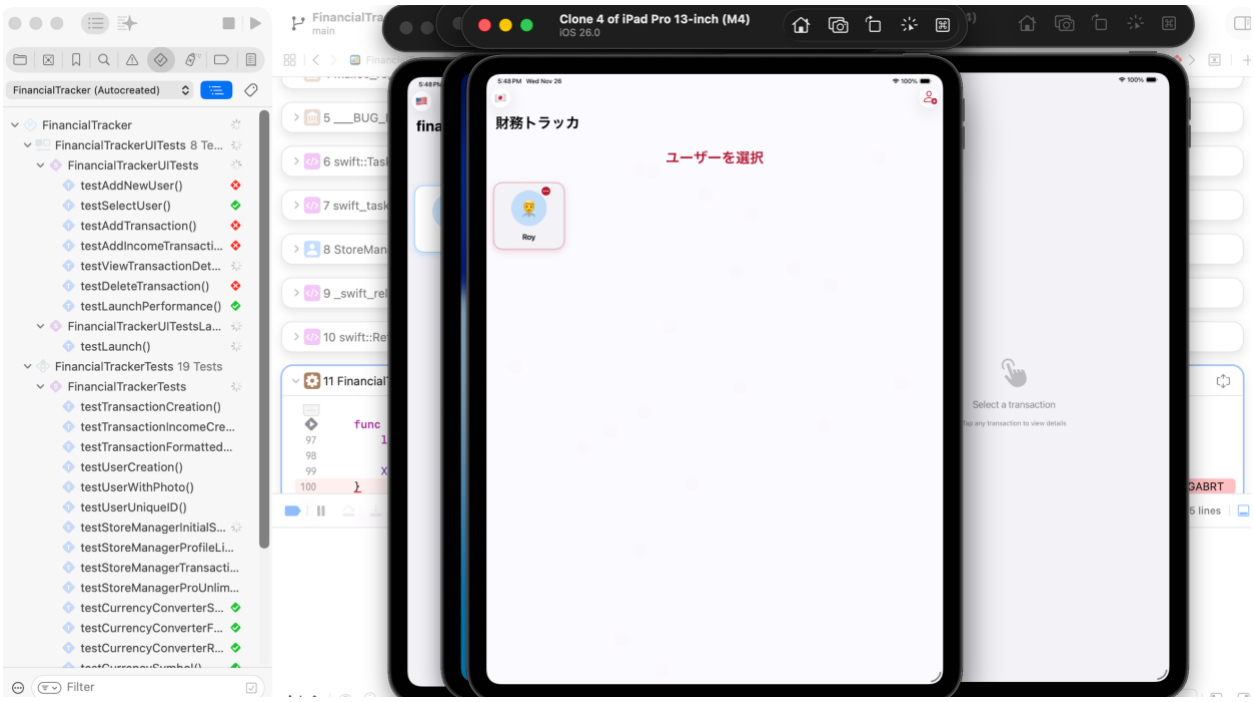
Exercise 3: UI Testing and Continuous Integration

User interface testing was the focus of this exercise, where I learned to write automated tests that simulate how a real person would use the app. Unlike unit tests that check code directly, UI tests interact with the app through the screen by tapping buttons, typing text, and navigating between screens. For my Financial Tracker app, I wrote UI tests that add new users by tapping the add button, entering a name, and saving the profile. I also created tests that add transactions by filling out the transaction form with a title and amount, then verifying that the transaction appears in the list. Other UI tests check that users can select profiles, view transaction details, and delete transactions. One important concept I learned was using accessibility identifiers, which are special labels added to buttons and text fields so that the testing code can find and interact with them reliably. I also added waiting commands to give the app time to respond before checking if something worked, which prevents tests from failing just because the app was still loading.

Exercise 4: Optimizing and Maintaining Automated Test Suites

The final exercise covered how to organize and improve test suites over time. As apps grow larger, the number of tests can become difficult to manage, so organization is important. I learned to use comment markers to separate tests into categories like user management tests, transaction tests, and performance tests, making it easier to find specific tests when needed. Performance testing was introduced, which measures how long certain operations take to complete. I added a performance test that measures how quickly the app launches and another that measures how fast transactions can be created. These tests help identify if changes to the code accidentally make the app slower. I also learned about code coverage, which shows what percentage of the app's code is actually being tested. Higher code coverage means more confidence that the app will work correctly. Finally, I learned about handling flaky tests, which

are tests that sometimes pass and sometimes fail for no clear reason. Using proper waiting and timeout values helps prevent flaky tests and makes the test suite more reliable.



Challenges and Solutions

Several challenges arose during the implementation of automated testing for the Financial Tracker app. The first major challenge was that my app supports multiple languages including English, Spanish, Japanese, and Arabic, which meant that button labels and text field placeholders change depending on the device language. Tests that looked for buttons labeled "Save" would fail when the device was set to Spanish because the button would be labeled "Guardar" instead. I solved this problem by adding accessibility identifiers to all interactive elements, which are invisible labels that stay the same regardless of language.

Another challenge was timing issues where tests would fail because they tried to check for something before the app finished loading or animating. For example, after tapping the save button, the test would immediately look for the new item in the list, but the app needed a moment to actually add and display it. I solved this by adding wait commands that give the app time to respond and using the waitForExistence function that waits up to a specified number of seconds for an element to appear.

Testing the trial version limits was also challenging because the StoreManager saves the pro purchase status to the device storage. This meant that after running a test that simulated

purchasing pro, subsequent tests would think the user was already a pro user. I solved this by resetting the relevant storage values in the setup method that runs before each test, ensuring every test starts with a clean state.

Finally, organizing the growing number of tests became difficult as more tests were added. With over twenty tests in the test files, finding specific tests required scrolling through a lot of code. I addressed this by using clear naming conventions for test functions that describe what they test and by using comment markers to create sections that group related tests together.

Conclusion

Completing these automated testing exercises has greatly improved the quality and reliability of my Financial Tracker app. I now have over twenty-unit tests that verify the core functionality of models like Transaction, User, and StoreManager, plus the CurrencyConverter that handles multi-currency support. The seven user interface tests ensure that the main user flows work correctly, including adding users, adding transactions, viewing details, and deleting items. These tests can be run automatically with a single keyboard shortcut, and they complete in under a minute while thoroughly checking that everything works.

The most valuable lesson from this course was understanding that testing is not extra work that slows down development, but rather an investment that saves time in the long run. When I make changes to my app, I can run the tests to quickly verify that I did not accidentally break something that was working before. This gives me confidence to continue improving the app without fear of introducing new bugs.

Going forward, I plan to continue adding tests as I add new features to the app. I will also work on increasing my code coverage percentage to ensure that more of my code is being tested. The practices I learned in this course will help me build better, more reliable apps throughout my development career.