LambdaSchool / **ios-tictactoe-unit-tests**

No description, website, or topics provided.

5 commits | **2** branches | **0** releases | **1** contributor

Branch: **afternoon-start** | View #1 | Create new file | Upload files | Find file | Clone or download

This branch is 3 commits ahead of master. #15 Compare

armadsen Update README.md Latest commit `fc7ee12` on Sep 12, 2018

TicTacToe.xcodeproj	Add afternoon project start	5 months ago
TicTacToe	Add afternoon project start	5 months ago
TicTacToeTests	Add afternoon project start	5 months ago
.gitignore	Initial commit of guided project README	5 months ago
README.md	Update README.md	5 months ago

README.md

Tic Tac Toe Unit Tests - Afternoon Project

The goal of this project is practice writing unit tests, and additionally using them to find and fix bugs, and to enable confident refactoring in iOS projects. You'll continue with the Tic Tac Toe project by adding additional unit test coverage, refactoring some of the game logic, adding additional tests, and fixing a couple bugs. This project will help you practice the concepts learned in the Unit Testing I - Intro to Unit Testing module of Sprint 8. After completing the lesson material and this project, you should be able to:

- understand and explain the purpose of unit testing
- add a test target to their existing app
- implement unit tests using XCTestCase
- use XCAssert functions to test whether code is working correctly
- run unit tests in Xcode

Part 0 - Switch to `afternoon-start` Branch

You'll be starting with the Tic Tac Toe app as we left it earlier. This repo includes the partly-finished version of the app that we've worked on before. In order to ensure that you're at a consistent starting point, you should consider using the repo's version rather than the app you built yourself following along with the guided project. Do the following:

1. If you haven't done so already, commit and push your own guided project version on `master`.
2. Check out the `afternoon-start` branch using `git checkout afternoon-start`.
3. Open TicTacToe.xcodeproj

If everything worked, you'll see the tests we wrote in `GameBoardTests.swift` and `GameAITests.swift`, along with some empty test methods in `GameAITests.swift`.

Part 1 - Finish AI Tests

The project includes the two game AI tests we wrote during the guided project, along with empty stubs for additional test methods. Fill out each of the additional test methods. Some of these methods have small board diagrams in comments. These are only suggests for board layouts to test. Feel free to use your own, or add additional layouts to your tests.

You are also welcome to add additional tests that you feel would better cover possible scenarios that should be tested.

As you work, run your tests frequently. As you know, tests can fail for two reasons: the code being tested has a bug, **or**, your test code itself has a bug. When a failure occurs, before fixing the problem, be sure that you understand whether the bug is in the app's code or the test code. Use the debugger to explore if need be.

Part 2 - Refactor Game Logic

Currently, the logic for managing game state, checking for completion, whether a win or a cat's game, is in GameViewController.swift. View controllers can be difficult to test, due to their dependence on user-facing UI, and it's good practice to factor as much logic out of them into discrete, small, easily testable units as possible. To that end, you'll factor as much of the game logic as possible into a new `Game` type, which the `GameViewController` will use to implement much of its logic. You'll also add additional tests to verify that `Game` 's functionality is correct. (Side note: You'll learn how to automate testing user interfaces in a future lesson.)

1. Create a new file, "Game.swift".
2. Define a struct called `Game` .
3. `Game` should have a public interface that looks like this:

```
struct Game {

  mutating internal func restart()
  mutating internal func makeMark(at coordinate: Coordinate) throws

  private(set) var board: GameBoard

  internal var activePlayer: GameBoard.Mark?
  internal var gameIsOver: Bool
  internal var winningPlayer: GameBoard.Mark?
}
```

A brief description of each method and property follows:

- `restart()` : Restarts the game to a fresh state with an empty board, and player X starting.
- `makeMark(at:)` : adds a mark for the currently active player at the given coordinate. Updates game state.
- `board` : Externally read-only property for the game board. `Game` itself modifies this as the game progresses.
- `activePlayer` : The currently active player, either `.x` or `.o` . That is, the player whose turn it is. `nil` if the game is over.
- `gameIsOver` : `true` if the game is over (either won or a cat's game), `false` if the game is still running.
- `winningPlayer` : The player that won the game, either `.x` or `.o` . `nil` if the game is still running, or it's a cat's game (no one won).

4. Begin by creating a new test case class, `GameTests` . Write (failing) test methods for as much of `Game` 's external interface as makes sense.
5. Implement `Game` 's methods and properties, adding any additional private methods, properties, or types you think you

need. The basic logic should be very similar to the existing logic in `GameViewController`.

6. As you implement, use an iterative process. Run a test, write code in `Game` to make the test pass, then extend the test to include more functionality, implement that functionality, and so on. Continue this process until you have a full suite of passing tests for `Game`.

After you've written `Game` and covered it well with tests, update `GameViewController` to use an instance of `Game` instead of its own internal logic. **Hint:* Because `Game` is a struct, you can add a `didSet` property observer on `GameViewController`'s `game` property which will be triggered anytime `game` changes state. This makes it easy to trigger updates to your UI in response to changes to `game`.

Part 3 - Test Thoroughly

Unit testing is a great way to ensure the quality of your code, and prevent bugs early on in the development cycle. However, it is not a complete substitue for manual testing. You still must test to make sure your app works as a whole system, from the user interface on down.

Thoroughly test your Tic Tac Toe game by running it in the simulator and testing various game scenarios to make sure the app correctly determines who won and when, prevents illegal moves, etc.

Go Farther

If you finish with time to spare or would like to push yourself, try the following bonus goals:

- Add the ability to undo the last move by tapping an Undo button.
- Add a true game AI so that the computer can play the game. Tic tac toe is a simple enough game that you can make a "perfect" AI, that is an AI that will always either win or cause a cat's game.

▼ Click here for a hint on creating an AI
You should investigate the use of the Minimax algorithm, which is a well-defined, classic algorithm that is well-suited to tic tac toe.

- Add tests for your true AI game opponent code.
- Style the app so it looks nicer