# Guided Learning (QLC-2): Building up the Application

### Initial Steps (QLC-1):

1. Create a new Xcode project with appropriate packages.
2. Create a model struct called Task.
3. Create a TaskListViewModel class with an empty array of tasks.
4. Write a unit test that verifies the TaskListViewModel starts with an empty task list.

### Objective

In this exercise, you will deepen your TDD practice by writing a test for the add(\_:) method on TaskListViewModel. Your goal is to verify that calling add(\_:) appends a new task to the tasks array.

### Step 1: Define the Problem & Requirements

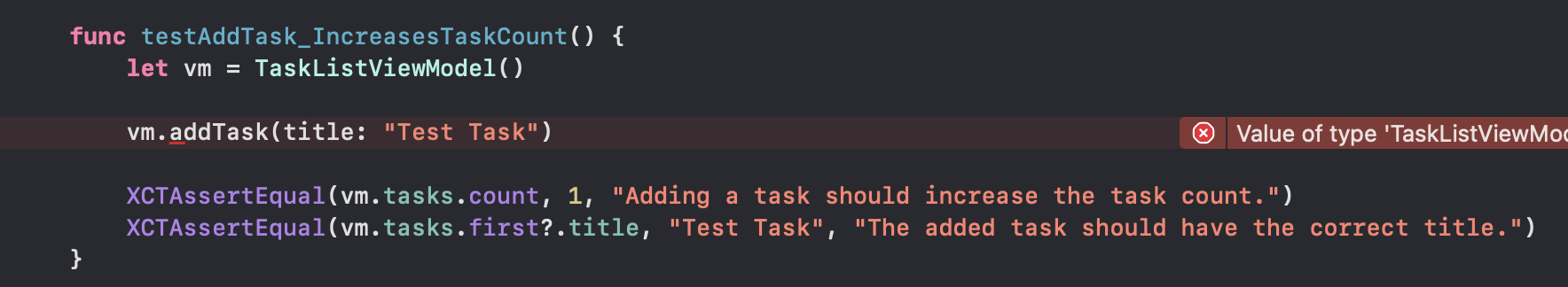
We need a new add(\_:) API on our view‑model that:

1. Accepts a String title.
2. Creates a Task with that title.
3. Append it to vm.tasks.

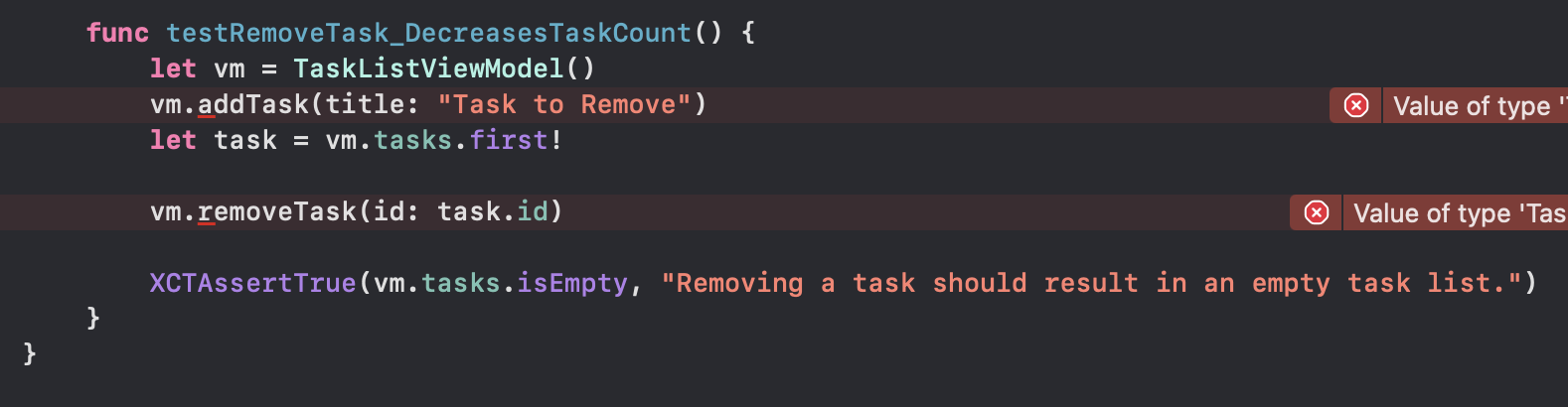
### Step 2: Write the Initial Test (RED)

Create (or update) **TaskListViewModelTests.swift** in your unit‑test target:

We’ll start off with a **add method:**



Follow this by writing a test for the **removal method**:



Notice how specific the test method names are, this is intentional for readability and maintainability. A good practice is to name your tests using a **method name + expected outcome** format.

For example, if you’re testing a conditional method, you should write **two separate test cases**:

* One for the success path
* One for the failure path

Both should have clearly descriptive names.

Additionally, if you ever feel the need to write an if statement inside a test: **stop and write another test instead**.

Tests should be focused, isolated, and not run sequentially. Each test should validate one specific behavior or outcome.

This keeps your test suite predictable, easy to maintain, and resilient to future changes.

### Step 3: Run the Test and Confirm Failure

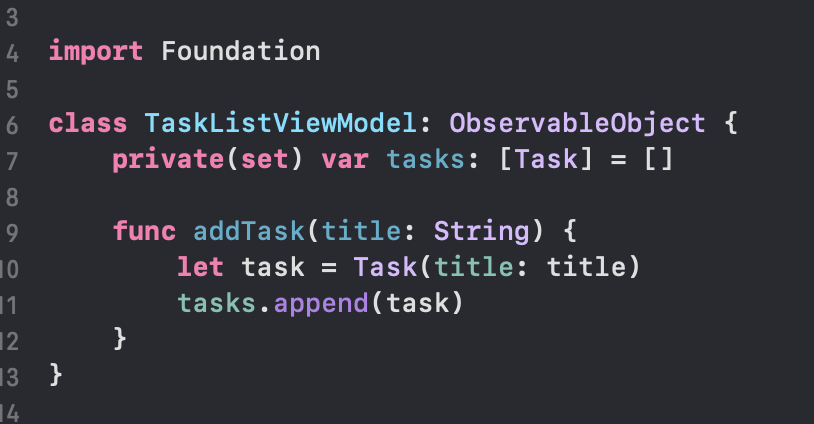
* Open the **Test Navigator** in Xcode (⌘ + 6).
* Run your new test by clicking the play button next to it.
* The test should **fail**, as expected.

### Step 4: Write the Minimal Implementation (GREEN)

Now that the test is failing, it’s time to implement the simplest possible code that makes the test pass.

* Open **TaskListViewModel.swift**
* Add the addTask(title:) method.
* Keep it minimal, no extra logic, just enough to pass the test.

Example:



### Step 5: Run the Test and Confirm Success

* Run your test again in Xcode.
* This time, the test should pass.
* If it still fails, double-check both your test logic and implementation.

**Why this matters:**

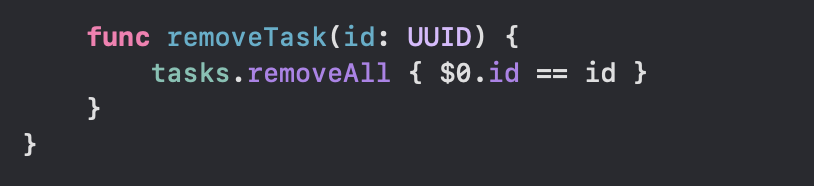
In TDD, we stop coding once the test passes. We write only as much code as necessary to make the test succeed, nothing more.

### 

### Step 6: Add the Removal Method

### After writing your failing test for removeTask(id:), you’ll now implement the minimal code to make the test pass.

Open **TaskListViewModel.swift** and add:



### Step 7: Refactor and Prepare for Production Use

When practicing TDD, your first goal is always to make the test pass, even if the code isn’t perfect yet. We’ve written the simplest possible code to satisfy our tests for adding and removing tasks.

However, while our current implementation works for tests, it lacks some of the structure and safety we want for a production-ready SwiftUI app.Now that our tests are green, it’s time to refactor and strengthen the code without changing its behavior.

#### What We’re Adding in This Refactor

**1. @Published for State Updates**

* This property wrapper notifies SwiftUI views when tasks changes.
* Without it, the UI won’t automatically update when tasks are added or removed.
* We’ll mark tasks as @Published and keep it private(set) so external callers can read, but not modify, the list directly.

**2. @MainActor for Concurrency Safety**

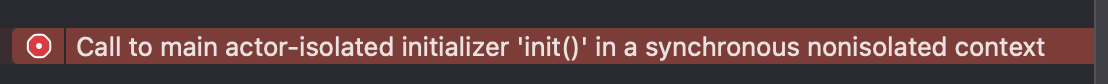
* In Swift 6, all state updates that affect the UI should happen on the **main actor (main thread)**.
* Marking the class with @MainActor guarantees that all its methods and properties are used on the correct thread.
* This prevents runtime crashes when TaskListViewModel is observed by SwiftUI views.

**3. final to Prevent Subclassing**

* We’ll make the class final since there’s no need for it to be subclassed.
* This is a Swift best practice for ViewModels unless you explicitly need inheritance.

### Handling the MainActor Isolation Warning

Once you annotate your ViewModel with @MainActor, you’ll likely notice that your **existing test code will throw a warning or error** like this:

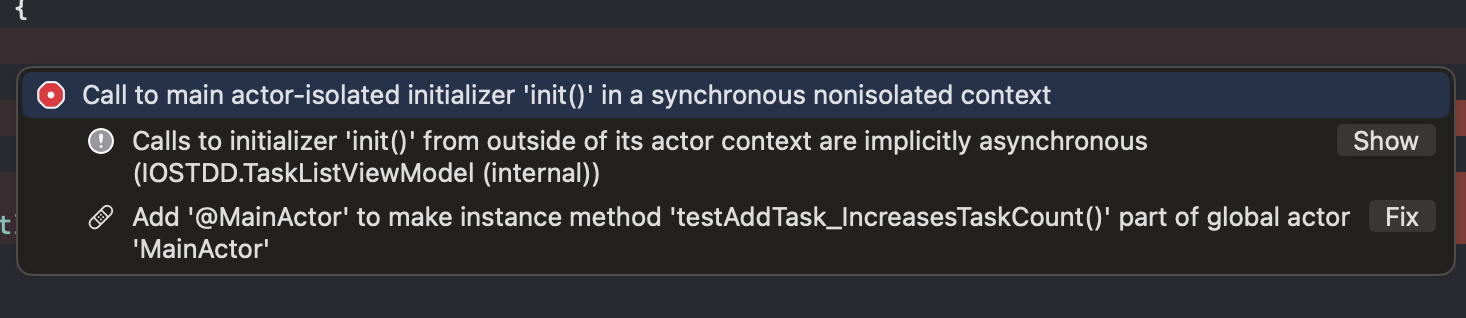


This happens because your test is trying to create an instance of TaskListViewModel, which is now isolated to the main actor, from a background thread or outside the actor’s context.

### Xcode’s Syntax Helper Makes This Easy

Thankfully, Xcode provides inline help to fix this automatically.

When you see the warning in the editor, click on it. Xcode will offer suggestions like:



Press **Fix**, and Xcode will add the @MainActor annotation to your test method for you.

## QLC‑2.1: Preventing Empty Tasks with TDD

### **Objective:**

Ensure that addTask(title:) only adds tasks with non-empty, trimmed titles.

### Steps:

1. Write a unit test that verifies addTask(title:) does not add a task when the title is empty or contains only whitespace.
2. Run the test and confirm it fails.
3. Update the addTask(title:) method to trim the title and ignore empty inputs.

* **Hint:** Use guard to exit early when the trimmed title is empty.

1. Run the test again and confirm it passes.
2. Ensure all previous tests still pass.

Check [https://github.com/Edrzapi/IOSTDD/tree/QLC-2,1-ANS](https://github.com/Edrzapi/IOSTDD/tree/QLC-1-ANS) for the solution

**Tip:**When you catch yourself writing if !condition { ... } else { ... } —  
consider if a guard might make it cleaner.