## Guided Learning (QLC-5): Introducing Mocks & Delegation

### Objective

Refactor business logic out of the ViewModel into a dedicated **TaskService**, and use **manual mocks** to verify that the service correctly invokes the existing repository. This illustrates how to test interactions rather than just state, and lays the groundwork for more advanced mocking in QLC-5.1+.

### Why This Matters

* **Separation of Concerns**ViewModel remains focused on UI state; TaskService owns all task-creation and removal logic.
* **Interaction Testing**By mocking the repository, we ensure the service calls exactly the right methods with the right data.
* **Real-World Architecture**Production apps rarely keep business rules in ViewModels—services (or “use-cases”) handle that layer.

### Step 1 — Define the Problem

We need a TaskService with two responsibilities:

1. **createTask(title:)** calls repository.add(\_:)
2. **deleteTask(id:)** calls repository.remove(\_:)

The existing TaskRepositoryProtocol already defines add(\_:) and remove(\_:); we will build on that.

### 

### 

### Step 2 — RED: Write the Minimal Failing Test

1. **Create the mock repository**In **Tests/Mocks/TaskRepositoryMock.swift**, define a TaskRepositoryMock that conforms to RepositoryProtocol and records two pieces of information:
   * Whether **addTask(\_:)** was called, and what TodoItem was passed in
   * Whether **removeTask(by:)** was called, and which UUID was passed in

**Why an actor?**

Because our protocol methods are async, Swift’s concurrency model enforces that any mutable state used across concurrency domains must be *isolated* to prevent data races. By defining our mock as an **actor**, we get:

* **Automatic isolation** of its stored properties
* Safe, compiler‐enforced access to mutable state
* No extra annotations or boilerplate



**Alternative 1: @MainActor**If you know your tests will always execute on the main thread, you can annotate the mock class with @MainActor. This tells Swift: “All methods and properties of this type run on the main thread.”

**Pros:**

* Simple to apply
* No need to convert your mock into an actor

**Cons:**

* Tightly couples your test double to the main thread
* Less flexible if you ever need to run tests on background executors or in parallel

**Alternative 2: @unchecked Sendable**You can also declare your mock as @unchecked Sendable, which opts out of the compiler’s isolation checks: “I promise this type is safe to share across threads, so don’t warn me about it.”

**Pros:**

* Minimal changes to your existing class
* Avoids the overhead of actor syntax

**Cons:**

* You take full responsibility for thread safety
* If you accidentally introduce unsynchronized mutations, you can get data races with no compiler help

**Write two failing tests** in **TaskServiceTests.swift**:

* + **testCreateTask\_CallsRepositoryAdd()**
    - **Arrange:** Instantiate TaskService with your TaskRepositoryMock
    - **Act:** Call service.createTask("Example")
    - **Assert:** Observe that the mock recorded addTaskCalled == true and that addedTask.title == "Example"
  + **testDeleteTask\_CallsRepositoryRemove()**
    - **Arrange:** Reuse (or re-instantiate) the same mock
    - **Act:** Call service.deleteTask(id: someUUID)
    - **Assert:** Observe that the mock recorded removeTaskCalled == true and removedID == someUUID

1. **Run the tests**. They should **fail** immediately, because no TaskService implementation exists yet.



### Step 3 — GREEN: Provide the Minimal Implementation

1. **Implement TaskService**
   * Add an initializer that accepts a TaskRepositoryProtocol.
   * Implement createTask(title:) to call repository.addTask(\_:).
   * Implement deleteTask(id:) to call repository.removeTask(by:).
2. **Verify the Mock**
   * Ensure your TaskRepositoryMock actor is available in Tests/Mocks/ and conforms to RepositoryProtocol.
   * After running the service methods, read back the mock’s recorded properties (addTaskCalled, addedTask, removeTaskCalled, removedID) to confirm they reflect the exact calls you expect.
3. **Run the Tests**
   * Execute your test suite.
   * Confirm that both testCreateTask\_CallsRepositoryAdd() and testDeleteTask\_CallsRepositoryRemove() now pass, and that the mock’s flags and stored values match the titles and IDs you provided.

This minimal implementation satisfies exactly the behavior your RED-phase tests expect, no more, no less.

### 

### Step 4 — REFACTOR

* Extract any duplicated setup in your tests into helper methods.
* Simplify mock implementation where possible.
* Verify production TaskService is minimal, no test-only code remains.
* Re-run **all** tests to guard against regressions.

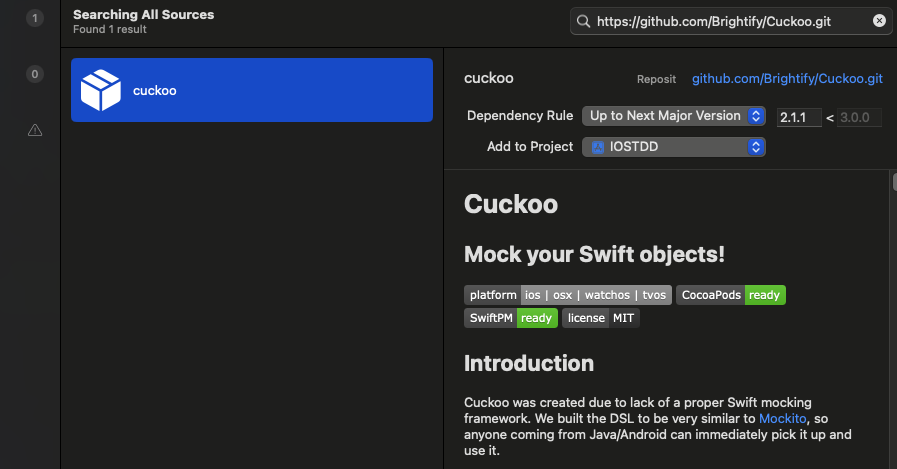
## QLC-5.1 — Automated Mocks with Cuckoo (Swift 5 Compatible)

### Objective

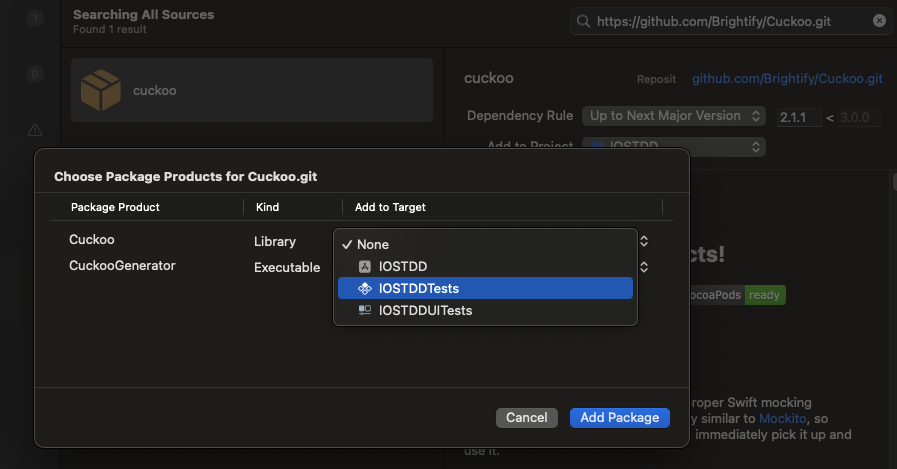
* Eliminate manual mock boilerplate by using **Cuckoo** to generate a mock for RepositoryProtocol.
* Gain expressive stub(...) and verify(...) APIs for interaction testing.

### Step 1 — Add Cuckoo as a Test Dependency

* Add the Cuckoo package to your project via Swift Package Manager.
* Include it **only** in your test target. (https://github.com/Brightify/Cuckoo.git)



Set the Cuckoo target as IOSTDDTests:



### Step 2 — Configure the Cuckoo Run Script

* In your test target’s **Build Phases**, add a **Run Script** before “Compile Sources.”
* Point the script at the Cuckoo generator and your RepositoryProtocol.swift file. (The **CuckooGenerator** product is an **executable**, not a library, so you don’t add it to a target, instead you invoke it from your Run Script phase)
* Specify an output folder (e.g., Tests/Mocks/) for the generated mock.

**Step by Step:**

* In Xcode’s Project Navigator, click on your project.
* Under “TARGETS,” select **IOSTDDTests**.

**Open Build Phases**

* With the test target selected, switch to the **Build Phases** tab.

**Add a New Run Script**

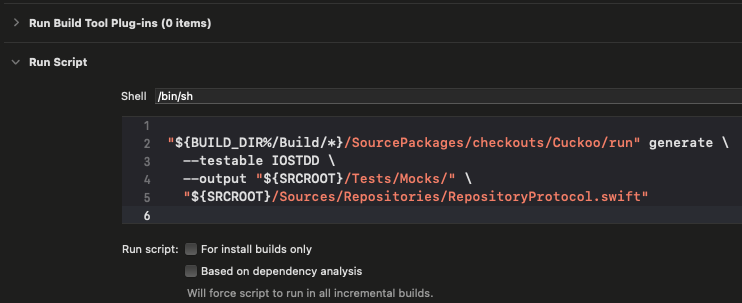
* Click the “+” in the top left of the Build Phases list.
* Choose **“New Run Script Phase.”**

**Drag It Above “Compile Sources”**

* In the phases list, drag your new Run Script so that it sits **above** the “Compile Sources” step. That way mocks are generated before the tests compile.

**Paste the Generation Command**

* In the script editor, enter something like (see below):



* Adjust paths/target names as needed for your project layout.

**Ensure the Output Folder Exists**

* Make sure there’s a **Tests/Mocks/** directory in your project.
* If not, create it so Cuckoo has a place to write MockRepositoryProtocol.swift.

**Build or Run Tests**

* Now whenever you build or run your tests (⌘U), Xcode will first invoke Cuckoo to regenerate mocks, then compile and run your test suite using the fresh mocks.

### Step 3 — Generate the Mock

* Build your test target.
* Confirm that Cuckoo has created MockRepositoryProtocol.swift in the designated mocks folder.
* Verify it’s included in your test target.

### Step 4 — Rewrite Your Service Tests

* Replace the hand-rolled mock with the **generated mock** type.
* Use Cuckoo’s **stub(...)** to define default behavior.
* Use Cuckoo’s **verify(...)** to assert the expected method calls and arguments.

## QLC-5.2 — Generated Mocks (Swift 6-Compatible with Sourcery)

### Objective

Automate the creation of mocks in a way that works cleanly with Swift 6 and modern concurrency. Instead of hand-writing mock implementations or relying on outdated tools like Cuckoo, we’ll use Sourcery to generate mocks for RepositoryProtocol. By marking protocols with an AutoMockable annotation and running a script, we get async-safe mocks with zero boilerplate.

### Why This Matters

* **Swift 6 Safety:** Works with actors, async/await, and Sendable types.
* **Clean Architecture:** Your mocks are defined by templates, not cluttering your test logic.
* **Reusable:** The same tool can generate Equatable, Hashable, and more for future use.

### Prerequisite — Install Sourcery

To use Sourcery, you need to install the tool locally on your machine.

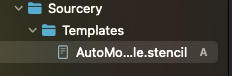
**Homebrew : brew install sourcery**Use Homebrew to install Sourcery globally.

### Step 1 — Annotate Your Protocol

Open RepositoryProtocol.swift and add a sourcery: annotation. This marks it for mock generation.

### Step 2 — Create the Sourcery Template

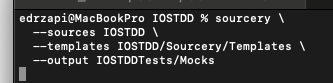
Inside your project, create a directory Sourcery/Templates.  
In that folder, create a template file that defines how your mocks should be generated, call it AutoMockable.stencil. This template can be customized for async methods, actors, or any other mocking pattern that matches your architecture.



(This is at the root of the project)

### Step 3 — Run Sourcery

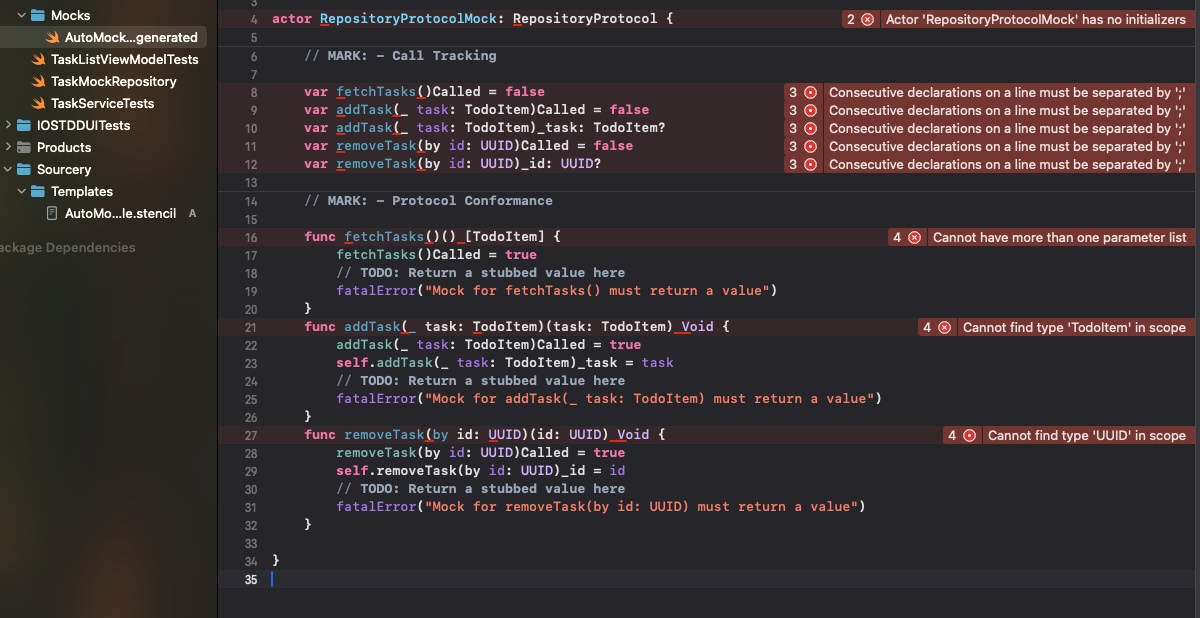
Run the Sourcery CLI with parameters pointing to your source code, template folder, and desired output directory.



Make sure the output directory exists, or Sourcery will fail to write the generated mocks. Also, double check where you’re running this command, ensure you set –templates correctly.

### Step 4 — Verify Generated Output

After running Sourcery, confirm that a file has been generated in your test target directory (e.g., IOSTDDTests/Mocks/RepositoryProtocolMock.swift).  
Open the generated file and inspect the mock. It should conform to RepositoryProtocol and include properties to track calls and parameters.

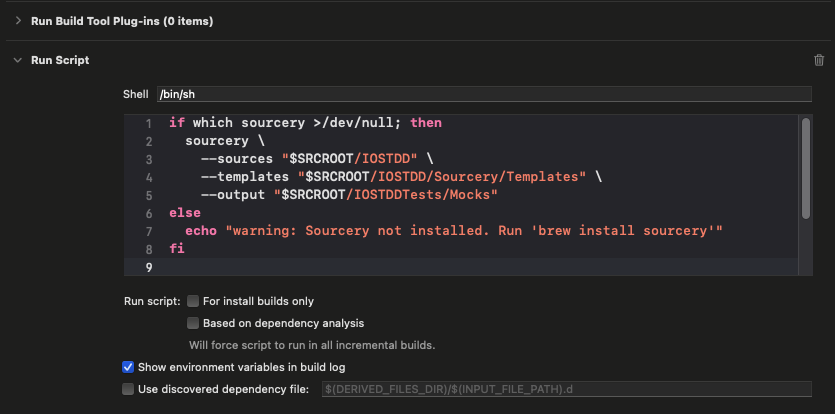


### Step 5 — Rewrite Your Tests (Optional)

Replace your manual TaskRepositoryMock with the generated mock class.  
Use the mock's tracking properties to assert that the right repository methods were called with the correct arguments. This eliminates boilerplate and ensures your tests remain focused on verifying behavior.

### Step 6 — Optional: Automate Sourcery in Xcode

* You can integrate Sourcery into your build pipeline by adding a custom Run Script Phase to your test target. Place the script above "Compile Sources" in the build phase order, so mocks are generated before compilation. This ensures your mocks are always up to date when running tests or building your test target, see below.
* If you want to give this method a go ensure you clean your build folder with Shift + Cmd + K and re-run your tests: Cmd + U.



### Summary

Sourcery allows you to define lightweight annotations and templates that automatically generate Swift 6 compatible mocks. Compared to hand-written mocks or legacy tools like Cuckoo, this approach is safer, cleaner, and more scalable. Whilst a little test modification might be required, this method integrates smoothly with async/await and lets you customize your mocks without manually updating them each time your protocols change.

## QLC-5.3 — Refactor: Wire the TaskService into Your ViewModel

### Objective

Refactor the app so that all business logic lives in TaskService and your view model is purely a state presenter. No behavior should change, this is a pure **Refactor** step in TDD (Tests must pass post change).

### Why This Matters

* **Separation of Concerns:** Keeps your view model focused on UI state only.
* **Test-Driven Confidence:** You’re restructuring with a safety net of existing tests.
* **Cleaner Codebase:** Future features build on a solid, service-based foundation.

### Steps

1. **Open** your TaskListViewModel file.
2. **Remove** any direct calls to the repository from inside the view model.
3. **Add** a private TaskService property—either injected or instantiated in your initializer.
4. **Redirect** all task operations through the service:
   * Replace repository.fetchTasks() with service.fetchTasks() (or leave fetching in the view model but via the service).
   * Replace repository.addTask(…) with service.createTask(title:…).
   * Replace repository.removeTask(by:…) with service.deleteTask(id:…).
5. **Maintain** any logic that updates your @Published var tasks: after each service call, ensure the view model reloads or mutates tasks accordingly.
6. **Run** the app:
   * Add a task—UI behavior should be identical.
   * Delete a task—again, no visible change.
7. **Execute** your full test suite:
   * All existing tests must still pass without modification.
   * No new tests are required for this refactor.

### Summary

You’ve now centralized all business rules in TaskService while your view model remains a simple bridge between service and UI. This refactor sets you up for subsequent steps, adding coordinators, delegates, or new features on a clean, test-driven foundation.

Check [https://github.com/Edrzapi/IOSTDD/tree/QLC-5.3-ANS](https://github.com/Edrzapi/IOSTDD/tree/QLC-1-ANS) for the solution

## 

## QLC-5.4 — Refactor: Update the ViewModel to Depend Only on the TaskService

### Objective

Refactor your TaskListViewModel so it depends **only** on a tested TaskService, not the repository directly.  
Remember to follow TDD principles:

* Update your tests first (**Red**)
* Change your implementation to match (**Green**)
* Then polish and clean up (**Refactor**)

### Scenario

You’ve already tested all business logic using TaskService.  
Now, improve your code by updating your TaskListViewModel to use this service exclusively.  
You’ll start by updating your test suite to expect the new initializer, which will break until you update the ViewModel. Once both are aligned, all tests will pass again.

### Steps

**Red: Update Your Tests**

* Change all your ViewModel initializations in test methods to pass a TaskService (built with a fake repository), not a repository directly.
* At this point, your tests won’t compile or will fail, because the ViewModel doesn’t support this initializer yet.

**Green: Update the ViewModel and Service**

* Change the ViewModel’s initializer to accept a TaskService (with a suitable default).
* Remove the repository property and all references to it.
* Update all methods to use the TaskService instead of the repository.
* **If your TaskService does not already have a fetchTasks() method, add it now.**This keeps your ViewModel completely decoupled from the repository.
* Update your ViewModel’s code to fetch tasks via the service, not directly from the repository.
* Now your tests should all compile and pass.

**Refactor: Clean Up and Update Mocks**

* Extract any repeated test setup (repo/service/viewModel) into helpers or setUp().
* Remove any leftover code or variables from the old repository-based approach.
* **Update your manual mocks (e.g., TaskRepositoryMock) to stub and track calls to fetchTasks() if you test it via the service.**
* **Add or update tests in your TaskServiceTests to verify the new method and all interactions, including the async fetch logic.**
* For async property access in your assertions (e.g. XCTAssertTrue(await mock.fetchTasksCalled)), always assign the awaited result to a variable before the assertion.
* Double-check your code for clarity and duplication.

### Summary

You’ve now centralized all business rules in TaskService, while your view model remains a simple bridge between service and UI.  
This refactor lays a clean foundation for future changes or features.

For the complete solution, check:  
<https://github.com/Edrzapi/IOSTDD/tree/QLC-5.4-ANS>

## QLC-5.5 — Working with Truth Tables

### Objective

Turn your TaskListViewModel test suite into **living documentation** by mapping each behavior to a blank truth table and ensuring you have one dedicated test per row.

### Scenario

You’ve already implemented and verified your addTask(title:) and removeTask(id:) logic. Now you’ll document every important case in a truth table, then align your tests so each table row corresponds to one test method—making your suite both a specification and a safety net.

### Step 1 — Analyse with a Blank Truth Table

Draw a table with **multiple rows**. You’ll fill in:

* scenarios for addTask(title:)
* scenarios for removeTask(id:)

Table to get you started

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### Step 2 — One Test per Row

For **each** filled-in row of the truth table:

1. **Write a new test method** in TaskListViewModelTests.swift:
   * Instantiate a fresh FakeRepository and TaskListViewModel(repository:) with the scenario’s inputs.
   * Invoke the appropriate method (addTask(title:) or removeTask(id:)).
   * Assert the view model’s tasks and the repository’s state match the table’s expected outputs.

### Step 3 — Refactor Test Setup

* Extract repository and view-model instantiation into setUp() or a helper method.
* Confirm each test remains self-contained and clearly named to reflect its table row.

### Step 4 — Review and Clean Up

* Run the full suite: **every** test should pass.
* If any table row lacks a passing test, add it now.
* Remove any leftover boilerplate or duplicated assertions, your suite should read like a precise specification.

### Step 5 — Reflect

* How does the truth table make it easier to see which cases are covered?
* Which future edge cases would you add to the table?
* Does each test name clearly describe its scenario as shown in the table?