
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Table of Contents: Extended

9: Spring Cleaning Your App

Have you ever run into a legacy app with a Massive View Controller or other architectural problems? In this session, you'll learn how to give legacy apps a spring cleaning. You'll learn how to iteratively split apart code, add testing, and prevent problems from happening again.

25 Spring Cleaning Your App: Demo 1

By Alex Curran

In this demo, you will learn how to recognize parts of your codebase which are difficult to deal with, and which will be most valuable to refactor. You'll learn techniques to refactor larger parts of your app safely into a state where they're easily testable and can be further improved.

The steps below will be explained in the demo, but here's the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo1\starter**.

1) Splitting apart responsibilities of a view

Build and run the app to make sure everything is working.

Create a new Swift file, **CountsView.swift**. Add the following protocol to the file:

```
protocol CountsView: class {

}
```

Open **CountsViewController.swift** and make the class `CountsViewController` implement the `CountsView` protocol.

Create a method at the bottom of the file with the two lines called found in completion block of `getCounts` in `viewDidLoad`. Call this method `onCountsUpdated`:

```
func onCountsUpdated(_ counts: [Count]) {
  adapter.update(with: counts)
  tableView.reloadData()
}
```

Call this method in the completion block:

```
getCounts(then: { [weak self] counts in
  self?.onCountsUpdated(counts)
}, onError: { error in
  print(error)
})
```

Next, add the method signature of `onCountsUpdated` into the `CountsView` protocol:

```
protocol CountsView: class {
  func onCountsUpdated(_ counts: [Count])
}
```

Do the same steps for the `increment(_:then:)` call in `tableView(_:didSelectRowAt:)` method, and `insert(_:then:)` in `addCountView(_:didFinishCreating:)`:

1. Extracting a method
2. Putting the method in the `CountsView` protocol
3. Refactor the code to use the `view` property

Your **CountsView.swift** file will contain:

```
protocol CountsView: class {
  func onCountsUpdated(_ counts: [Count])
  func onCountInserted(_ update: IncrementalUpdate)
  func onCountIncremented(_ update: IncrementalUpdate)
}
```

And your `CountsViewController` should now have code similar to the following:

```
func tableView(_ tableView: UITableView,
               didSelectRowAt indexPath: IndexPath) {
  let selectedCount = adapter.item(at: indexPath)
  increment(selectedCount, then: { [weak self] update in
    self?.onCountIncremented(update)
  })
  tableView.deselectRow(at: indexPath, animated: true)
}

func addCountView(_ view: AddCountView,
                  didFinishCreating count: CountRequest) {
  insert(count, then: { [weak self] update in
    self?.onCountInserted(update)
  })
}

func onCountsUpdated(_ counts: [Count]) {
  adapter.update(with: counts)
  tableView.reloadData()
}

func onCountIncremented(_ update: IncrementalUpdate) {
```

```
    adapter.update(with: update.counts)
    tableView.reloadRows(at: update.updatedIndexes.asIndexPaths(),
                         with: .automatic)
  }

  func onCountInserted(_ update: IncrementalUpdate) {
    adapter.update(with: update.counts)
    tableView.insertRows(at: update.updatedIndexes.asIndexPaths(),
                         with: .automatic)
  }
```

2) Extracting a presenter

First, create a property in the `CountsViewController` called `countsView`, which is `CountsView?` and just returns the `CountsViewController` itself. It will become clearer why it must be optional later:

```
  var countsView: CountsView? {
    return self
  }
```

Next, replace the code in the `getCounts(then:)`success handler with the following:

```
  self?.countsView?.onCountsUpdated(counts)
```

Do the same for `increment(_:then:)`, and `insert(_:then:)`.

Create a file **CountsPresenter.swift**, and in it create a class `CountsPresenter`:

```
  class CountsPresenter {

    init(countsView: CountsView) {

    }

  }
```

Create a lazy property of the `CountsPresenter` in `CountsViewController`:

```
  private lazy var presenter = CountsPresenter(countsView: self)
```

Into this class, move the following methods and properties from `CountsViewController`:

```
  private var scheduler: GrandCentralScheduler!
  private var persistentContainer: NSPersistentContainer!

  func getCounts(then handler: @escaping CountsHandler,
                 onError errorHandler: @escaping ErrorHandler)

  private func counts() throws -> [Count]
```

```
func increment(_ count: Count,
               then resultHandler: @escaping IncrementalUpdateHandler)

func insert(_ countRequest: CountRequest,
            then resultHandler: @escaping IncrementalUpdateHandler)
```

Add `import CoreData` to the top of **CountsPresenter.swift**.

Fix the compile errors in **CountsViewController.swift** by pointing the method calls you've just moved to the `presenter` property.

Build and run to make sure the app is compiling.

Delete the assignment of `scheduler` and `persistentContainer`. Change the `scheduler` and `persistentContainer` properties from `var` to `let`, making them no longer implicitly-unwrapped and assign them in the initialiser of the presenter:

```
class CountsPresenter {

  private let scheduler: GrandCentralScheduler
  private let persistentContainer: NSPersistentContainer

  init(countsView: CountsView) {
    scheduler = GrandCentralScheduler()
    persistentContainer = NSPersistentContainer(name: "Model")
  }
}
```

> By looking at where implicitly-unwrapped optionals are created, you can identify whether there's a missing object. An implicitly-unwrapped optional means that the property "lives" for a shorter time that the class it is in, indicating that you can extract a new class with a shorter lifecycle.

Next, modify the initialiser for the `CountsPresenter` so that it saves the parameter `countsView` to a new corresponding `countsView` property:

```
private weak var countsView: CountsView?

init(countsView: CountsView) {
  self.countsView = countsView
  scheduler = GrandCentralScheduler()
  persistentContainer = NSPersistentContainer(name: "Model")
}
```

Build and run to make sure this is compiling.

> Normally in iOS development you'd set a `delegate` or `countsView` property, which is an easy source of bugs if you forget to set it. Making it required in a method or initialiser will force the client of the presenter class to supply one.

3) Further steps

Refactor `CountsPresenter` to use its `view` property directly in `getCounts()`, `insert(_:)` and `increment(_:)`. Remove the requirement for `CountsViewController` to handle the result from the success handlers entirely, and then remove the handlers entirely from the methods.

4) That's it!

You now have a significantly smaller view controller. Much of the interaction with the model is now in your `CountsPresenter` class, and is now testable without requiring the creation of a view controller and faking the view controller lifecycle in your test.

You've not directly modified any code here, just moved things around and changed their lifecycles. A lot of the actions performed could be automated (and in other IDEs and languages, are automated), which reduces the risk of mistakes in refactors, and keeps the app compiling at all times. If you're interested in investigating more, AppCode has very robust tools for Objective-C refactoring.

As you've not changed any code behaviour, you can be relatively confident you haven't broken anything. You'll next see how this refactor has made testing the code easier, enabling us to make later, bigger, refactors less risky.

26 Spring Cleaning Your App: Demo 2

Alex Curran

In this demo, you will test your new, more separate code. At the same time, you will identify more areas to refactor and simplify.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Important note**: You **must** begin work with the starter project in **Demo2\starter**.

Between Demo 1 and Demo 2, the completion handlers were removed in the methods `getCounts()`, `insert(_:)` and `increment(_:)`, as they can now be replaced by calling the `CountsView` methods directly.

A new method on `CountsView` was added to handle loading failures, `onCountsNotLoaded(with error: Error)`.

1) Testing CountsPresenter

Open the test class **CountsPresenterTests.swift**, found in the `Tests` group. In the class `CountsPresenterTests`, add the function:

```
func testInsertingACountWillReturnAnUpdateWithOneMoreItem() {

}
```

You will write a test that inserts a new `Count` and then verifies that the `CountsView` you pass in gets the method `onCountInserted(_:)` called. As the insertion is asynchronous, we'll need to use `XCTestExpectations`. Uncomment the class `MockCountsView` at the bottom of the file.

This class implements the `CountsView` protocol, and is initialised with an

`XCTestExpectation`. When `onCountsUpdated(_:)` is called (by the `CountsPresenter`), it saves a copy of the counts that were loaded. When `onCountInserted(_:)` is called (by the `CountsPresenter`), a copy of the `update` property is saved and the expectation marked as fulfilled. This allows the test to continue past any waiting code.

Returning to the test named `testInsertingACountWillReturnAnUpdateWithOneMoreItem`, add this code into the test body:

```
// 1
let incrementExpectation =
  expectation(
    description: "Increment callback wasn't triggered")
// 2
let mockView = MockCountsView(expectation: incrementExpectation)
let presenter = CountsPresenter(countsView: mockView)
// 3
presenter.getCounts()

// 4
presenter.insert(Count(id: nil,
                       title: "hello",
                       count: 0,
                       total: 2,
                       interval: .weekly,
                       resetTime: nil,
                       increment: 0))
// 5
wait(for: [incrementExpectation], timeout: 1.0)

//6
XCTAssertEqual(mockView.insertionUpdate?.counts.count,
mockView.initialCounts.count + 1)
```

In the code above you:

1. Create an expectation, with a message indicating what will happen if our expectation is not fulfilled
2. Initialise a `MockCountsView` which accepts the expectation you defined, and a `CountsPresenter`, the target of our testing
3. Trigger a fetching of the counts. This is required because of how the CoreData stack is set up. We would ideally fix this, but ignore it for now.
4. Attempt to insert a `CountRequest` via the `CountsPresenter`.
5. Wait for the expectation which will be fulfilled once a count has been inserted into CoreData. After this, you know `mockView.insertionUpdate` has been set (looking back at `MockCountsView.onCountInserted(_:)`).
6. Assert that the insertion update contained one more count than was initially loaded.

Run the test by clicking the white diamond to the left of the test name (or pressing Cmd+U). The test should pass.

This test seems like it could be made simpler by just asserting that after the insertion we have one item, as the test should have 0 items to start with! Let's make that change to see. Change the assertion to be:

```
XCTAssertEqual(mockView.insertionUpdate?.counts.count, 1)
```

Run the test and you will see it fail. This is because CoreData is not deleted between test runs! You saw earlier that tests are better when they are isolated from unreliable dependencies such as the file-system. This would be something to improve in the future.

For now, change the assertion back to what is shown below and make sure it passes successfully:

```
XCTAssertEqual(mockView.insertionUpdate?.counts.count,
mockView.initialCounts.count + 1)
```

Now there is a reliable test, you must ensure it is testing the right thing. Often, tests can pass even though they *should* fail, because they were set up incorrectly. The way to verify this is by breaking some code, and ensuring the test fails. The code you break should always be in the **production** code, not the test code.

Open **CountsPresenter.swift** and go to the `insert(_:)` method. Highlight the lines below and comment them out by pressing **Cmd+/**:

```
insertionContext.insert(storedCount)
try insertionContext.save()
```

This will prevent the newly created Count being inserted into CoreData.

Run the tests again (**Cmd+U**), and notice the test fails with an error mentioning that the two numbers are not equal. This verifies that your test is indeed testing the right thing. Uncomment the lines in `CountsPresenter.insert(_:)` again, and run the test one more time to ensure it works again.

3) Creating a stronger test

The test is good but still can be improved. The assertion asserts that there is one new item, but makes no checks about what that item contains. It could be something completely made up!

Create another test:

```
func testInsertingACountWillReturnAnItemWithTheSameTitle() {
```

```
  }
```

It is often easiest to write a test by writing the assertion first. Here, you need to check that the title of the new `Count` is the same as the `CountRequest` you define. Copy this into the body of the new test (it won't compile for now):

```
  XCTAssertEqual(
    mockView.insertionUpdate?.firstUpdatedCount?.title,
    countRequest.title)
```

This assertion checks whether the first updated count (there is only one item updated in an insertion) has the same title as the `countRequest` property.

Copy this code and place it above the assertion:

```
  let incrementExpectation =
    expectation(
      description: "Increment callback wasn't triggered")
  let mockView = MockCountsView(expectation: incrementExpectation)
  let presenter = CountsPresenter(countsView: mockView)
  presenter.getCounts()

  let countRequest = Count(id: nil,
                           title: "hello",
                           count: 0,
                           total: 2,
                           interval: .weekly,
                           resetTime: nil,
                           increment: 0)
  presenter.insert(countRequest)

  wait(for: [incrementExpectation], timeout: 1.0)
```

This performs similar to our previous test, except that we store the `countRequest` as a local variable.

You will find the code still not compiling, with the error "Value of type `IncrementalUpdate` has no member `firstUpdatedCount`". Fix this by uncommenting the extension to `IncrementalUpdate` in the file:

```
  extension IncrementalUpdate {

    var firstUpdatedCount: Count? {
      guard let firstUpdateId = updated.first else {
        return nil
      }

      return counts.first(where: { count in
        return count.id == firstUpdateId
      })
    }
  }
```

We keep this extension in the test code as it is only useful there. Run the test

(**Cmd+U**) and it should pass.

Again, break the test in production code to ensure it is correct. Open **CountsPresenter.swift**, and in the `insert(_:)` method change the line:

```
storedCount.title = countRequest.title
```

To something which would break the test, for example:

```
storedCount.title = "RWDevCon 2018"
```

Run the tests, and the test `testInsertingACountWillReturnAnItemWithTheSameTitle` should fail. Undo this change in **CountsPresenter.swift** and run the tests again. They should all pass.

4) That's it!

Congrats, at this time you should have a good understanding of how to write end-to-end tests on legacy code! Take a break, and then it's time to move onto refactoring the code and preventing it building up into large classes again.

Spring cleaning your app: Demo 3

Alex Curran

In this demo, you will find ways to make your code less brittle and harder to break in the future.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo3\starter**.

1) Creating value types

Open **Count.swift** and add an `Id` struct inside the `Count` struct:

```
struct Id: Equatable {

  let rawValue: String

}
```

Change the return type of the property `id` from `String?` to `Id?`:

```
let id: Id?
```

> Patterns like this - replacing primitives types with strongly-typed, named objects - may have an impact on the amount of memory your app uses, if the object is not a struct. However, most software craftspeople agree that strongly-typed wrappers like this are worth the impact, as they remove a whole set of potential bugs. You can read more about this pattern by looking up the term "Primitive Obsession".

Build and run, and notice our new compile errors. Fix the one in `CountsPresenter.counts()` by replacing the `id` parameter in initialisation of the `Count` with `Count.Id(rawValue: rawId)`.

The second compile error in `increment(_:)` in **CountsPresenter.swift** can be solved by changing the second line of the guard-statement from:

```
let url = URL(string: countId),
```

to

```
let url = URL(string: countId.rawValue),
```

Open **CountsViewController.swift** and change the `IncrementalUpdate`'s `updated` property to be of type `[Count.Id]`, and the `updated(_,in:)` function to accept a `Count.Id` too. Go to **CountsPresenter.swift**, and in `CountPresenter.insert(_:)` change the line:

```
return .updated(rawId, in: try self.counts())
```

to

```
return .updated(Count.Id(rawValue: rawId), in: try self.counts())
```

Build and run (including the tests) by pressing Cmd+U.

> Interesting to note is that we're having to change between a CoreData `NSManagedObjectID` frequently, from our `Count.Id`. It would be unwise to make the `rawValue` of `Count.Id` be of type `NSManagedObjectID`, as that ties your implementation of your domain objects directly to CoreData. If you wanted to move to Realm, that would make things very difficult!

2) Improve domain modelling

Create a new file, called **Foo.swift**. In it, create a struct called `Foo`:

```
struct Foo {

}
```

Open **Count.swift** and copy all the properties to the `Foo` class. Remove the two optional attributes, `id` and `resetTime`. Fix the compile error on the `interval` property by changing the type from `Interval` to `Count.Interval`.

Now it is clearer what this object will be, right click the struct name > Refactor > Rename. Rename the struct to `CountRequest`.

> Xcode may well fail to perform the refactor at this point. If so, just rename it manually.

Still in `CountRequest`, create a computed variable `asCount` like this:

```
var asCount: Count {
  return Count(id: nil,
    title: title,
    count: count,
    total: total,
    interval: interval,
    resetTime: nil,
    increment: increment)
}
```

This allows us to move between the two representations.

Open **AddCountView.swift** and in the function `textFieldShouldReturn(_:)`, replace the lines:

```
let request = Count(id: nil,
  title: inputText,
  count: count,
  total: 3,
  interval: .weekly,
  resetTime: nil,
  increment: increment)
delegate?.addCountView(self, didFinishCreating: request)
```

with the following:

```
let request = CountRequest(title: inputText,
  count: count,
  total: 3,
  interval: .weekly,
  increment: increment)
delegate?.addCountView(self, didFinishCreating: request.asCount)
```

> This technique of converting between "old" and "new" models is slower, but allows us to control the spread of the new model. This is particularly useful in large codebases as you prevent having to change many different files all in one go.

Next, still in **AddCountView.swift**, change `AddCountViewDelegate`'s delegate method to have the signature:

```
func addCountView(_ view: AddCountView,
                  didFinishCreating count: CountRequest)
```

Build and run to see the compile error in the line you changed further up the file. Remove the `.asCount` to fix this error. Build and run to see a new error in **CountsViewController.swift**.

Open **CountsViewController.swift** and replace the `addCountView(_,didFinishCreating:)` method with:

```
func addCountView(_ view: AddCountView,
                  didFinishCreating count: CountRequest) {
  presenter.insert(count.asCount)
}
```

Build and run again to ensure the app is compiling.

Next open **CountsPresenter.swift** and navigate to the `insert(_:)` method. Change the method signature to:

```
insert(_ countRequest: CountRequest)
```

Go back to **CountsViewController.swift** and in `addCountView(_,didFinishCreating:)`, change the call to the presenter with:

```
presenter.insert(count)
```

Build and run the app, which should compile. The tests will not compile, so replace the two instances of:

```
Count(id: nil,
      title: "hello",
      count: 0,
      total: 2,
      interval: .weekly,
      resetTime: nil,
      increment: 0)
```

with:

```
CountRequest(title: "hello",
             count: 0,
             total: 2,
             interval: .weekly,
             increment: 0)
```

Run the unit tests to ensure they pass. Now we no longer need to switch between our old and new models, remove the `CountRequest.asCount` property.

3) Removing our technical debt

Open **Count.swift** and make `id` and `resetTime` non-optional values, removing the `?`.

Build to see the compile errors.

Open **CountsPresenter.swift** and go to the `increment(_:)` method. Move the `let countId = ...` assignment from the guard block and place it directly before the guard. The start of the method should look like this:

```
let countId = count.id.rawValue
guard let url = URL(string: countId),
  let id = self.persistentContainer
               .persistentStoreCoordinator
               .managedObjectID(forURIRepresentation: url) else {
    return .noUpdate(from: try self.counts())
}
```

Open **CountRequest.swift** and remove the `asCount` property. You no longer need it as the refactor you used it for is complete.

4) That's it!

Congrats! At this time you should have a good understanding of how to identify code to refactor, how to refactor code in incremental steps, and how to recognise where a domain model is being used inappropriately.

Whilst you could do refactoring like this in one big go, this is very difficult on legacy codebases where many things are shared. Instead, working methodically up (or down) a call chain allows you to control how far your refactoring goes. Avoiding long periods of time where the code is not compiling is beneficial, as when your code does not compile, you get no feedback about issues, and you can't run tests which will spot logical errors.

Splitting domain models like `Count` and `CountRequest` is critical if you have a large and confusing codebase, and absolutely essential if you plan to modularise your app into separate pieces or feature teams. Taking the time to realise missing domain models also will clarify when an object is or isn't valid.