Practical Testing for an Imperative World

robenkleene/testing-presentation

Roben Kleene

Topics

— Unit Testing
— Functional Programming
— Composition
— Dependency Injection
— Mock Objects
— Case Study: WSJ's Barfly

Why write unit tests?

— No more "moving the food around on your plate"
— Reduce feedback loops
— Facilitate refactoring
— Manual testing is boring

Functional Style

— First class functions
— Higher-order functions
— Declarative (vs. Imperative)

Functional Programming

— Calling a function with the same inputs always produces the same result.

— This means no state.

— Unlike Object-Orientated Programming, where methods can access objects state (e.g., through properties).

Class vs. Function: Simple Introducer

```
// Class
class SimpleIntroducer {
    func whoIsIt(_ name: String) -> String {
        return "It's \(name)"
    }
}
assert("It's Poppy" == SimpleIntroducer().whoIsIt("Poppy"))

// Function (Don't actually do this!)
func whoIsIt(_ name: String) -> String {
    return "It's \(name)"
}
assert("It's Poppy" == whoIsIt("Poppy"))
```

Class vs. Function: Less Simple Introducer

```
// Class
class LessSimpleIntroducer {
    var announcer = "Taylor Swift"
    func whoIsIt(_ name: String) -> String {
        return "\(announcer) says \"It's \(name)\""
    }
}
let lessSimpleIntroducer = LessSimpleIntroducer()
lessSimpleIntroducer.announcer = "Beyonce"
assert("Beyonce says \"It's Poppy\"" == lessSimpleIntroducer.whoIsIt("Poppy"))

// Function (Don't actually do this!)
func whoIsIt(announcer: String, name: String) -> String {
    return "\(announcer) says \"It's \(name)\""
}
assert("Kanye West says \"It's Poppy\"" == whoIsIt(announcer: "Kanye West",
                                                   name: "Poppy"))
```

Class vs. Function: Interfaces

```
// Class
class LessSimpleIntroducer {
    var announcer: String
    func whoIsIt(_ name: String) -> String
}

// Function
func whoIsIt(announcer: String,
             name: String) -> String
```

More Complex Interfaces

```
// Class
class MoreComplexIntroducer {
    var announcer: String
    var objectIdentifier: ObjectIdentifier
    var objectExplainer: ObjectExplainer
    func whoIsIt(_ name: String) -> String
    func whatIsIt(_ object: Any) -> String
    func whatDoesItDo(_ object: Any) -> String
}

// Function
func whoIsIt(announcer: String,
             name: String) -> String
func whatIsIt(objectIdentifier: ObjectIdentifier,
              object: Any) -> String
func whatDoesItDo(objectExplainer: ObjectExplainer,
                  object: Any) -> String
```

Reason #1 that functional programming facilitates testing is that it clarifies your API.

Confusing Async Introducer

```
class ConfusingAsyncIntroducer {
    var announcer = "Taylor Swift"
    func whoIsIt(_ name: String) {
        DispatchQueue.global().async {
            print("\(self.announcer) says \"It's \(name)\"")
        }
    }
}

let confusing = ConfusingAsyncIntroducer()

// This is straight-forward
confusing.announcer = "Beyonce"
confusing.whoIsIt("Poppy")
// Beyonce says "It's Poppy"

// But this is unexpected!
confusing.announcer = "Taylor Swift"
confusing.whoIsIt("Poppy")
confusing.announcer = "Kanye West"
// Kanye West says "It's Poppy"
```

Clear Async Introducer

```
class ClearAsyncIntroducer {
    class func whoIsIt(announcer: String, name: String) {
        DispatchQueue.global().async {
            print("\(announcer) says \"It's \(name)\"")
        }
    }
}

ClearAsyncIntroducer.whoIsIt(announcer: "Taylor Swift",
                             name: "Poppy")
// Taylor Swift says "It's Poppy"
// And it's always the same, no matter what happens later!
```

Reason #2 that functional programming facilitates testing is that it reduces the testing surface area.

As a general rule, to make your application more testable, write as much of your program functional as possible.

"Imperative shell, functional core"
— Gary Bernhardt, Boundaries, 2012

Composition

— "Composition over inheritance"
— Object composition - Wikipedia: "Combine simple objects or data types into more complex ones"
— For example, in a Twitter client, instead of having a `UIViewController` download and parse an API call itself, it could have a `TweetGetter` that performs that work. Then `TweetGetter` could have an `APICaller` and a `ResponseParser`.

Without Composition

```
class AllInOneTweetListViewController: UIViewController {
    let url = URL(string: "https://api.twitter.com/1.1/search/tweets.json")!

    override func viewDidLoad() {
        getTweets(at: url) { tweets in
            // Display the tweets
        }
    }

    func getTweets(at url: URL, completion: ([Tweet]) -> ()) {
        downloadTweets(at: url) { json in
            parseTweets(from: json) { tweets in
                completion(tweets)
            }
        }
    }

    func downloadTweets(at url: URL, completion: (String) -> ()) {
        // ...
    }

    func parseTweets(from json: String, completion: ([Tweet]) -> ()) {
        // ...
    }
}
```

What's wrong with this?

Without composition, tests are difficult to write because individual components can't be loaded separately.

With Composition #1

```
class ComposedTweetListViewController: UIViewController {
    let url = URL(string: "https://api.twitter.com/1.1/search/tweets.json")!
    let tweetGetter = TweetGetter()

    override func viewDidLoad() {
        tweetGetter.getTweets(at: url) { tweets in
            // Display the tweets
        }
    }
}
```

With Composition #2

```
class TweetGetter {
    let apiCaller = APICaller()
    let responseParser = ResponseParser()

    func getTweets(at url: URL, completion: ([Tweet]) -> ()) {
        apiCaller.downloadTweets(at: url) { json in
            responseParser.parseTweets(from: json) { tweets in
                completion(tweets)
            }
        }
    }
}

class APICaller {
    func downloadTweets(at url: URL, completion: (String) -> ()) {
        // ...
    }
}

class ResponseParser {
    func parseTweets(from json: String, completion: ([Tweet]) -> ()) {
        // ...
    }
}
```

With composition, individual components can be loaded separately.

```
let apiCaller = APICaller()
let responseParser = ResponseParser()
let tweetGetter = TweetGetter()
```

Reason #1 that composition facilitates testing is by allowing individual components to be loaded separately.

Dependency Injection

— Dependency injection - Wikipedia: "Dependency injection is a technique whereby one object supplies the dependencies of another object."

— James Shore: "'Dependency Injection' is a 25-dollar term for a 5-cent concept."

— For example, instead of the `TweetGetter` initializing the `APICaller` and `ResponseParser` itself, it takes those dependencies as initialization parameters.

```
// Without Dependency Injection

class StiffTweetGetter {
    let apiCaller = APICaller()
    let responseParser = ResponseParser()
}

// With Dependency Injection

class FlexibleTweetGetter {
    let apiCaller: APICaller
    let responseParser: ResponseParser
    init(apiCaller: APICaller, responseParser: ResponseParser) {
        self.apiCaller = apiCaller
        self.responseParser = responseParser
    }
}
```

Why use dependency Injection?

It allows dependencies to be mocked.

Mock Objects

— Mock object - Wikipedia: "Mock objects are simulated objects that mimic the behavior of real objects in controlled ways."

— For example, `TweetGetter` could be initialized with a `MockAPICaller`, that instead of making network calls, it returns a constant `string` for the API response.

Mock Objects Example

```
class MockAPICaller: APICaller {
    override func downloadTweets(at url: URL, completion: (String) -> ()) {
        // Use a built-in constant JSON response
    }
}

class TweetGetterTests: XCTestCase {
    var tweetGetter: TweetGetter!

    override func setUp() {
        super.setUp()
        tweetGetter = TweetGetter(apiCaller: MockAPICaller(),
                                  responseParser: ResponseParser())
    }

    func testTweetGetter() {
        // Test that `tweetGetter.getTweets(at:completion:)` produces
        // the correct tweets for the constant JSON response
    }
}
```

Reason #1 that dependency injection facilitates testing is that it allows dependencies to be mocked.

Reason #2 that composition facilitates testing is that it allows dependency injection.

Summary

— Functional programming clarifies a classes API, and reduces the testing surface area.

— Composition makes individual components loadable separately, and faciliates dependency injection.

— Dependency injection allows mocking a classes dependencies.

Case Study: WSJ's Barfly

— Barfly, because our backend system is called Pubcrawl (it crawls publications).

— Barfly is responsible for downloading all the content in the WSJ app.

Basic Building Block

— Copy a `TestData` folder into the test bundle as a build phase.

— Create a simple helper function to access the contents of the `TestData` folder.

```
extension XCTestCase {
    public func fileURLForTestData(withPathComponent pathComponent: String) -> URL {
        let bundleURL = Bundle(for: type(of: self)).bundleURL
        let fileURL = bundleURL.appendingPathComponent("TestData").appendingPathComponent(pathComponent)
        return fileURL
    }
}

class ManifestTests: XCTestCase {
    func testManifest() {
        let testDataManifestNoEntryPathComponent = "manifestNoEntry.json"
        let fileURL = fileURLForTestData(withPathComponent: testDataManifestNoEntryPathComponent)
        print("fileURL = \(fileURL)")
    }
}
```

Weird Trick #1: `XCTestCase` Subclasses

(These are postfixed with `TestCase` not `Tests`.)

```
class MockFilesContainerTestCase: XCTestCase {
    var mockFilesContainer: FilesContainer!
    override func setUp() {
        super.setUp()
        mockFilesContainer = MockFilesContainer()
    }
}

class MockCatalogUpdaterTestCase: MockFilesContainerTestCase {
    var mockCatalogUpdater: CatalogUpdater!
    override func setUp() {
        super.setUp()
        mockCatalogUpdater = MockCatalogUpdater(filesContainer: mockFilesContainer)
    }
}

class CatalogUpdaterTests: MockCatalogUpdaterTestCase { }
```

It's mocks all the way down

```
class BarflyCatalogUpdateTestCase: TestDataFilesContainerTestCase {
    var barfly: MockBarfly!
    func setUp() {
        barfly = MockBarfly(...)
    }

    func updateCatalog() -> Catalog {
        var updatedCatalog: Catalog!
        let updateCatalogExpectation = expectation(description: "Update catalog")
        updateCatalogWithCompletion { (error, catalog) -> Void in
            updatedCatalog = catalog
            updateCatalogExpectation.fulfill()
        }
        waitForExpectations(timeout: testTimeout, handler: nil)
        return updatedCatalog
    }
}
```

It Scales!

```
class Barfly {
    public init(catalogContainerLoader: CatalogContainerLoader,
                catalogController: CatalogController,
                containerLoader: ContainerLoader,
                trashDirectoryURL: URL,
                jobCoordinator: JobCoordinator,
                containerManifestLoader: ContainerManifestLoader,
                foregroundContainersUpdater: ForegroundContainersUpdater,
                backgroundContainersUpdater: BackgroundContainersUpdater,
                janitor: Janitor,
                maxConcurrentBackgroundDownloads: Int)
    {
        // ...
    }
}
```

Weird Trick #2: `Tester` Frameworks

Create "Tester" targets to share the same testing infrastructure across apps and frameworks.

`Barfly Targets`

* Barfly
* BarflyTester
* BarflyTests
    * Imports Barfly
    * Imports BarflyTester

`WSJ Targets`

* WSJ
    * Imports Barfly
* WSJ Tests
    * Imports Barfly
    * Imports BarflyTester

This way `WSJ Tests` can subclass `BarflyCatalogUpdateTestCase` and call `updateCatalog()`.

That's All Folks

Thanks for listening!

robenkleene/testing-presentation

Roben Kleene