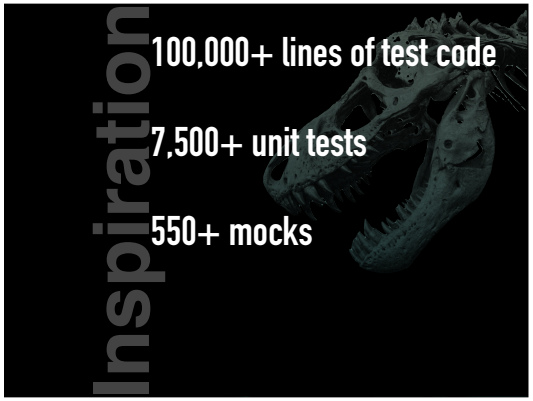

Rey was based on a large code base that one of our clients has. The tests in this code base are written in Quick and Nimble, with quite a few needing to test RxSwift code. Rey was the result of looking for ways to reduce the very large amount of boilerplate required to write these tests.

First we'll look at mocks and how they are written.

```
protocol HTTPClient {

    func postCompletable(url: String) -> Completable

    func getSingle(url: String) -> Single<RemoteCallResponse>

    func doMaybe(url: String) -> Maybe<RemoteCallResponse>

    func doObservable() -> Observable<Int>
}
```

Here’s a made example interface that encompasses a fair cross section of RxSwift functionality that we want to mock.

Here’s what a mock of this interface would look like if we wrote it to be used in multiple tests.

```
class MockHTTPClientOldSchool: HTTPClient {

    var postCompletableURL: String?
    var postCompletableURLSuccess: Bool?
    var postCompletableURLError: Error?
    func postCompletable(url: String) -> Completable {
        postCompletableURL = url
        if let _ = postCompletableURLSuccess {
            return Completable.empty()
        }
        if let error = postCompletableURLError {
            return Completable.error(error)
        }
        fatalError("Unexpected method call")
    }

    var getSingleURL: String?
    var getSingleURLResult: RemoteCallResponse?
    var getSingleURLError: Error?
    func getSingle(url: String) -> Single<RemoteCallResponse> {
```

As you can see we’ve got code for setting response values, choosing which one to returning, storing arguments for later validation and throwing errors.

```
            return Completable.error(error)
        }
        fatalError("Unexpected method call")
    }

    var getSingleURL: String?
    var getSingleURLResult: RemoteCallResponse?
    var getSingleURLError: Error?
    func getSingle(url: String) -> Single<RemoteCallResponse> {
        getSingleURL = url
        if let result = getSingleURLResult {
            return Single.just(result)
        }
        if let error = getSingleURLError {
            return Single.error(error)
        }
        fatalError("Unexpected method call")
    }

    var doMaybeURL: String?
    var doMaybeURLComplete: Bool?
    var doMaybeURLResult: RemoteCallResponse?
    var doMaybeURLError: Error?
```

And the same for every function.

```
        fatalError("Unexpected method call")
    }

    var doMaybeURL: String?
    var doMaybeURLComplete: Bool?
    var doMaybeURLResult: RemoteCallResponse?
    var doMaybeURLError: Error?
    func doMaybe(url: String) -> Maybe<RemoteCallResponse> {
        doMaybeURL = url
        if let result = doMaybeURLResult {
            return Maybe.just(result)
        }
        if let _ = doMaybeURLComplete {
            return Maybe.empty()
        }
        if let error = doMaybeURLError {
            return Maybe.error(error)
        }
        fatalError("Unexpected method call")
    }

    var doObservableResults: [Int]?
    var doObservableError: Error?
```

There’s also a hidden flaw in this in that it executes synchronously instead of asynchronously like the implementation does. Potentially leading to false positives in test suites.

```
protocol HTTPClient {

    func postCompletable(url: String) -> Completable

    func getSingle(url: String) -> Single<RemoteCallResponse>

    func doMaybe(url: String) -> Maybe<RemoteCallResponse>

    func doObservable() -> Observable<Int>
}
```

So now lets look at the same protocol done with a Rxy mock implementation.

```
class MockHTTPClientRxy: BaseMock, HTTPClient {

    var postCompletableURL: String?
    var postCompletableURLResult: CompletableResult?
    func postCompletable(url: String) -> Completable {
        postCompletableURL = url
        return mockFunction(returning: postCompletableURLResult)
    }

    var getSingleURL: String?
    var getSingleURLResult: SingleResult<RemoteCallResponse>?
    func getSingle(url: String) -> Single<RemoteCallResponse> {
        getSingleURL = url
        return mockFunction(returning: getSingleURLResult)
    }

    var doMaybeURL: String?
    var doMaybeURLResult: MaybeResult<RemoteCallResponse>?
    func doMaybe(url: String) -> Maybe<RemoteCallResponse> {
        doMaybeURL = url
        return mockFunction(returning: doMaybeURLResult)
    }

    var doObservableResults: ObservableResult<Int>?
    func doObservable() -> Observable<Int> {
        return mockFunction(returning: doObservableResults)
    }
}
```

First off it's considerably less code. All the variations on results are gone as is all the switching and throwing code. Instead we have a single result variable that can be set with a multitude of values and a single function that does all the dirty work.

Tests

So that's mocking taken care of. What about the unit test code that uses the mocks?

```
mockHTTPClientOldSchool.getSingleURLResult = RemoteCallResponse(aValue: "abc")

let disposeBag = DisposeBag()

var callDone: Bool = false
var response: RemoteCallResponse?

remoteService.makeSingleRemoteCall(toUrl: "xyz")
    .subscribe(
        onSuccess: { result in
            response = result
            callDone = true
        },
        onError: { error in
            fail("Unexpected error \(error)")
            callDone = true
        }).disposed(by: disposeBag)

expect(callDone).toEventually(beTrue())
validateSuccess(response: response)
```

Here's a typical test for a RxSwift call. Notice the DisposeBag and code to handle the different paths the execution could take. Luckily we're still using Nimble or there would be even more code to setup a test expectation and wait.

```
mockHTTPClientRxy.getSingleURLResult = .value(RemoteCallResponse(aValue: "abc"))

let response = remoteService.makeSingleRemoteCall(toUrl: "xyz")
                    .waitForSuccess()

validateSuccess(response: response)
```

Here's what it looks like with Rxy. All our boilerplate is gone, hidden in the .waitForSuccess() function call.

```
let disposeBag = DisposeBag()
var callDone: Bool = false
mockHTTPClientOldSchool.getSingleURLResult = RemoteCallResponse(aValue: "abc")

remoteServiceOldSchool.makeSingleRemoteCall(toUrl: "xyz")
    .asObservable()
    .concatMap { response -> Single<RemoteCallResponse> in
        expect(response.aValue) == "abc"
        self.mockHTTPClientOldSchool.getSingleURLResult = RemoteCallResponse(aValue: "def")
        return self.remoteServiceOldSchool.makeSingleRemoteCall(toUrl: "xyz")
    }
    .asObservable()
    .concatMap { response -> Single<RemoteCallResponse> in
        expect(response.aValue) == "def"
        self.mockHTTPClientOldSchool.getSingleURLResult = RemoteCallResponse(aValue: "ghi")
        return self.remoteServiceOldSchool.makeSingleRemoteCall(toUrl: "xyz")
    }
    .asSingle().subscribe(
        onSuccess: { response in
            expect(response.aValue) == "ghi"
            callDone = true
        },
        onError: { error in
           fail("Unexpected error \(error)")
           callDone = true
        })
    .disposed(by: disposeBag)

expect(callDone).toEventually(beTrue())
```

To really show the advantage, here's a real life example (slightly obfuscated) take from a larger test suite. Can you tell what it does?

```
mockHTTPClientRxy.getSingleURLResult= .value(RemoteCallResponse(aValue: "abc"))
expect(self.remoteServiceRxy.makeSingleRemoteCall(toUrl: "xyz").waitforSuccess()?.aValue) == "abc"

mockHTTPClientRxy.getSingleURLResult = .value(RemoteCallResponse(aValue: "def"))
expect(self.remoteServiceRxy.makeSingleRemoteCall(toUrl: "xyz").waitforSuccess()?.aValue) == "def"

mockHTTPClientRxy.getSingleURLResult = .value(RemoteCallResponse(aValue: "ghi"))
expect(self.remoteServiceRxy.makeSingleRemoteCall(toUrl: "xyz").waitforSuccess()?.aValue) == "ghi"
```

Now here's the same test after re-writing it with Rxy. Easier to understand now?


Rxy
github: outware/Rxy
