TDD IN SWIFT – PART DEUX

Chris Woodard for
Tampa Bay Cocoaheads

TDD IN SWIFT – PART DEUX

- Unit Testing (Recap)
- Testing template
- Organizing Unit Tests
- Designing for Testability
- Helpers
- Testing Asynchronous Code
- Testing Networking Code
- Testing Singletons
- Resources

UNIT TESTING

UNIT TESTING

- Write the unit tests first
- Write the just enough code to make the unit tests pass
- Repeat

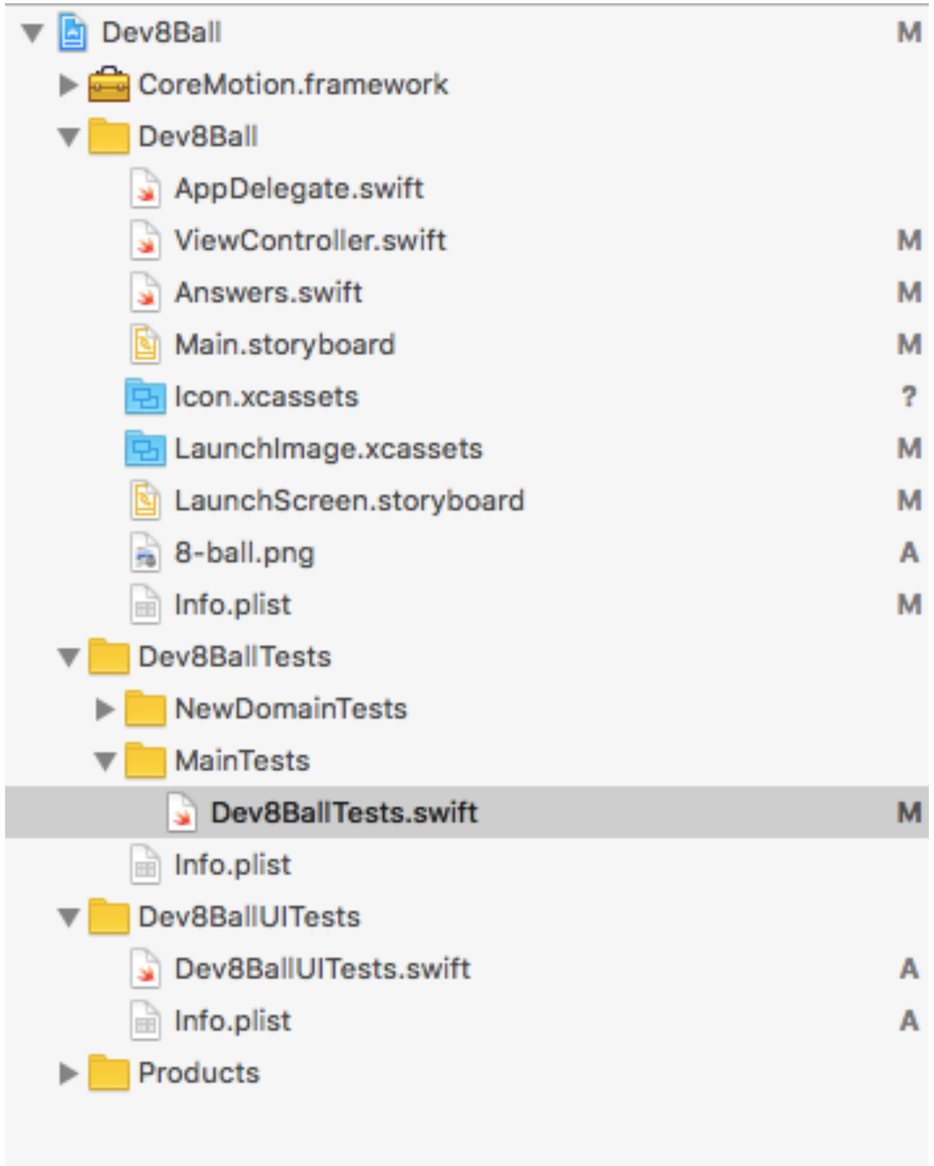
UNIT TESTING

UNIT TESTS

- Exercise small parts of your code
  - Start with known conditions
  - Execute a method or function
  - Test the final conditions
- Reports the result of the test, pass or fail
- *Minimize dependencies*

UNIT TESTS IN XCODE

- Live in their own bundle
- Are grouped together in “test cases”
- “Test cases” are subclasses of XCTestCase
- “Unit tests” are methods within the test cases
- Signature is *testXXXXX() {*
- Results are tested with XCTAssert*



UNIT TESTS IN XCODE

- Bundle name is “Dev8BallTests”
- Test Cases are added to testing bundle
- Test Cases can be organized into groups just like regular source code can.

```swift
//
//  Dev8BallTests.swift
//  Dev8BallTests
//
//  Created by Chris Woodard on 2/13/16.
//  Copyright © 2016 CW. All rights reserved.
//

import XCTest

@testable import Dev8Ball

class Dev8BallTests: XCTestCase {

    var answers:DeveloperAnswers? = DeveloperAnswers()

    override func setUp() {
        super.setUp()
    }

    override func tearDown() {
        super.tearDown()
    }

    func testMyFail() {
        let isNil:String? = nil
        XCTAssertNotNil(isNil)
    }

    func testFirstAnswer() {
        let firstAnswer = self.answers?.firstAnswer()
        XCTAssertNotNil(firstAnswer, "Did not return first answer.  BOO!")
    }

    func testLastAnswer() {
        let lastAnswer = self.answers?.lastAnswer()
        XCTAssertNotNil(lastAnswer, "Did not return first answer.  BOO!")
    }

}
```

UNIT TEST CASE

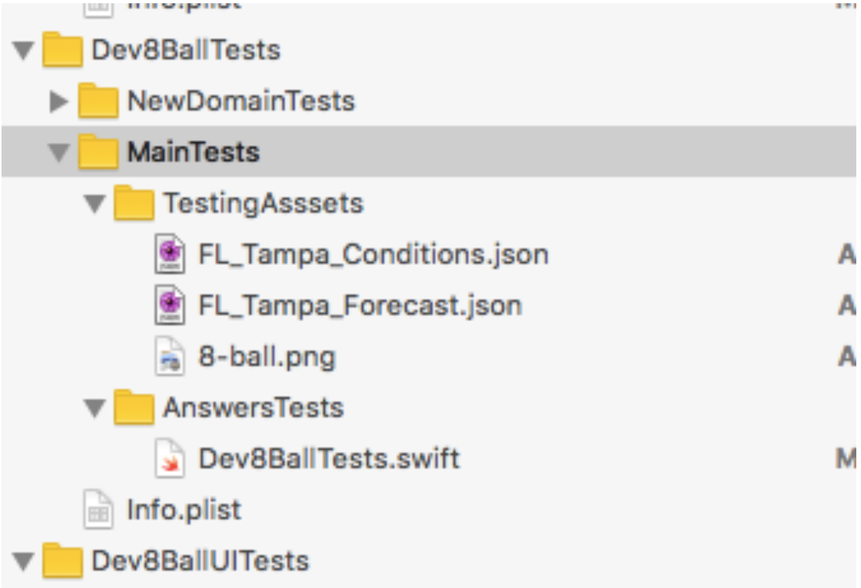
- Derived from XCTestCase
- Has helper methods that run with each unit test
- has XCTAssert* functions that generate the testing output

UNIT TESTING TEMPLATE

- func test*DescriptiveTestName* {
  - set initial conditions
  - call method or function under test
  - capture output
  - test output with XCTAssert* calls
    - use last parameter to provide a text message in case assert (and test) fails
- }

UNIT TESTING EXTRAS

- Organize unit tests along the lines of features and sub-features
- Put these in project groups
- Add resources (images, JSON files, etc) to the testing bundle in appropriate groups



DESIGN FOR TESTABILITY – THE BASICS

DESIGN FOR TESTABILITY – THE BASICS

- Design and write using functional style
  - Methods are small
  - No shared mutable state
  - Minimal shared state
  - Only input is parameters
  - Only output is return tuple… except for
  - Asynchronous methods (network calls or large data movement like saving a document)

TESTING ASYNC CODE

ASYNC CODE PATTERN

```
[someObject someMethodWithCompletion:^(NSData *result) {
    // some code to process result
}];
```

ASYNC CODE PATTERN – WITH EXTRAS

```
__block NSData *returnedData;
[someObject someMethodWithCompletion:^(NSData *result) {
   returnedData = result;
}];

// some code to process returnedData
```

ASYNC CODE PATTERN - IMPLEMENTATION

```
-(Boolean)someMethodWithCompletion:(^(void)(NSData
*result))completion {
    dispatch_async(dispatch_get_main_queue(), ^{
      // compute *result
      if(completion) {
         completion(result);
      }
    });
    return YES
 }
```

*Completion blocks are for **asynchronous** methods*

PROBLEMS

- Async pattern lends itself to being used inappropriately
- Async pattern can make your code difficult to follow
- If you have several calls to a shared database and a wrapper that uses a queueing model to serialize access *cough*FMDB*cough* and you have individual operations that need to proceed serially you wind up playing games with the run loop.

SO... TESTING ASYNC CODE?

OLD STYLE SOLUTION (OBJECTIVE C)

- set up your initial conditions
- method under test has to have a completion block called when the async operation is finished
- __block BOOL done = NO;
- call method under test
  - in completion block, set done = YES;
- in code below method call and completion block write
  - NSDate *endDateTime = [NSDate dateWithTimeIntervalFromNow];
  - while(!done) { [[NSRunLoop currentLoop] runUntilDate:endDateTime]; } **or similar**

EXAMPLE FROM THE WEB

```
- (void)testSaveAndCreateDocument {
    NSURL *url = ...; // URL to file
    UIManagedDocument *document = [[UIManagedDocument alloc] initWithFileURL:url];

    // Set the flag to YES
    __block BOOL waitingForBlock = YES;

    // Call the asynchronous method with completion block
    [document saveToURL:document.fileURL
        forSaveOperation:UIDocumentSaveForCreating completionHandler:^(BOOL success) {
            // Set the flag to NO to break the loop
            waitingForBlock = NO;
            // Assert the truth
            STAssertTrue(success, @"Should have been success!");
        }];

    // Run the loop
    while(waitingForBlock) {
        [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode
                                    beforeDate:[NSDate dateWithTimeIntervalSinceNow:0.1]];
    }
}
```

http://blog.dadabeatnik.com/2014/07/13/asynchronous-unit-testing-in-xcode-6/

THE NEW COOL WAY

- Use XCTestExpectation
  - let exp = expectationWithDescription("name")
  - waitForExpectationsWithTimeout(seconds, message) grabs the run loop and runs it until the expectation object's *fulfill()* method is invoked ***or*** the timeout expires and a fail message appears.
  - execute XCTestAsserts
  - call exp.fulfill() to allow execution of the unit test to complete

EXAMPLE FROM THE WEB

```
    func testSaveAndCreateDocument() {
        let url = NSURL.URLWithString("path-to-file")
        let document = UIManagedDocument(fileURL: url)

        // Declare our expectation
        let readyExpectation = expectationWithDescription("ready")

        // Call the asynchronous method with completion handler
        document.saveToURL(url, forSaveOperation:
UIDocumentSaveOperation.ForCreating, completionHandler: { success in
            // Perform our tests...
            XCTAssertTrue(success, "saveToURL failed")

            // And fulfill the expectation...
            readyExpectation.fulfill()
        })

        // Loop until the expectation is fulfilled
        waitForExpectationsWithTimeout(5, { error in
            XCTAssertNil(error, "Error")
        })
    }
```

http://blog.dadabeatnik.com/2014/07/13/asynchronous-unit-testing-in-xcode-6/

TESTING NETWORK CODE

ISSUES TESTING NETWORKING CODE

- Asynchronous, client/server
- Involves a large dependency
- Difficult to ensure the same initial conditions in the test
- Cannot ensure consistent test operation
  - Low network quality
  - Loss of connection
  - Variable latency

OBJECTIVE-C SOLUTION

- Mock the server API with a dependency injection framework like OHHTTPStubs
  - Capture JSON (or XML) output from the real API
  - Set that JSON / XML data to be returned from the mock API
  - Test the output
- OHHTTPStubs inserts itself into the NSURLConnection stack and redirects calls to its internal framework
- Your code doesn't know the difference.
- ***Depends on Objective C runtime architecture***

EXAMPLE FROM EARLIER COCOAHEADS TALK

```
-(void)testMockWeatherAPITampaFLConditionsNotNil
{
    self.didFinish = NO;
    self.didFail = NO;
    __block NSError *err = nil;
    __block CurrentWeather *weather = nil;
    self.mockAPI.httpStatusCode = 200;

    [_fetcher fetchCurrentLocalWeatherWithCompletion:^(NSHTTPURLResponse *response,
        NSError *error){
            weather = [_fetcher currentWeather];
            self.didFinish = YES;
            if(nil == error)
                self.didFail = NO;
            else
                self.didFail = YES;
            err = [error copy];
        }
    ];

    while(!self.didFinish && !self.didFail)
    {
        [[NSRunLoop currentRunLoop] runUntilDate:[NSDate dateWithTimeIntervalSinceNow:5
            ]];
    }

    XCTAssertNotNil( weather, @"weather should not be nil" );
}
```

SWIFT SOLUTION – STRATEGY PATTERN

- Design your networking class or function to separate the *variants* from the *invariants*. Assuming a login function that sends a username and password to a server and retrieves an access token, the invariants (for example) will be:
  - the code that does the actual login transaction with your server
  - the code that caches the token for use by other code in the app
- Your login() function takes two closures as parameters: one for login and one for caching.

SWIFT SOLUTION – STRATEGY PATTERN

```
typealias LoginProc = (String, String) -> (Int, String)
typealias CacheProc = (String) -> Int

func login(username:String, password:String, lp:LoginProc, cp:
    CacheProc ) -> Bool {

    var loggedIn:Bool

    let (status, token) = lp(username, password)
    if 0 == status {
        let result = cp(token)
        if 1 == result {
            loggedIn = true
        }
        else {
            loggedIn = false
        }
    }
    else {
        loggedIn = false
    }

    return loggedIn
}
```

SWIFT SOLUTION – STRATEGY PATTERN

```
var fauxLogin:LoginProc = { (username, passwd) -> (Int, String) in

    var result:(Int, String)

    if "frankenstein" == username && "edgar winter" == passwd {
        result.0 = 0
        result.1 = "90802ahshs88-8930"
    }
    else {
        result.0 = -1
        result.1 = ""
    }

    return result
}

var fauxCache:CacheProc = { (token) -> Int in

    var result:Int
    if "90802ahshs88-8930" == token {
        result = 1
    }
    else {
        result = 0
    }
    return result
}

let loggedIn:Bool = login( "frankenstein", password: "edgar winter",
    lp: fauxLogin, cp: fauxCache )

let otherLoggedIn = login( "walk this way", password: "aerosmith", lp:
    fauxLogin, cp:fauxCache )
```

DESIGN FOR TESTABILITY – DEEPER STUFF

DESIGN FOR TESTABILITY – DEEPER STUFF

- Helper functions/methods
  - Data generators
    - Random user data
    - Specific dates and timespans
    - Random arrays
  - Resource modifiers
    - JSON templates with substitutable keywords in them
    - Ditto XML

WAIT – WHAT ABOUT THE TDD?

ABOUT TDD

- You still have to generally plan what you're building
- When doing TDD stay within a feature or sub-feature
- When doing TDD, don't forget the helpers
- Apart from that, have fun.

RESOURCES

- http://goshdarnclosuresyntax.com for when you can't figure out how to declare closures for the strategy pattern
- https://en.wikipedia.org/wiki/Software_design_pattern
- https://github.com/ochococo/Design-Patterns-In-Swift
- https://developer.apple.com/library/tvos/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/04-writing_tests.html
- http://masilotti.com/better-swift-unit-testing/