

# Test Driven Development in Swift

*Chris Woodard*  
*Tampa Bay Cocoaheads*

# Unit Testing

What?

- **Smallest** complete cohesive bits of code
- **Known** inputs and outputs
- **Automatic** and exhaustive

# Unit Testing

Why?

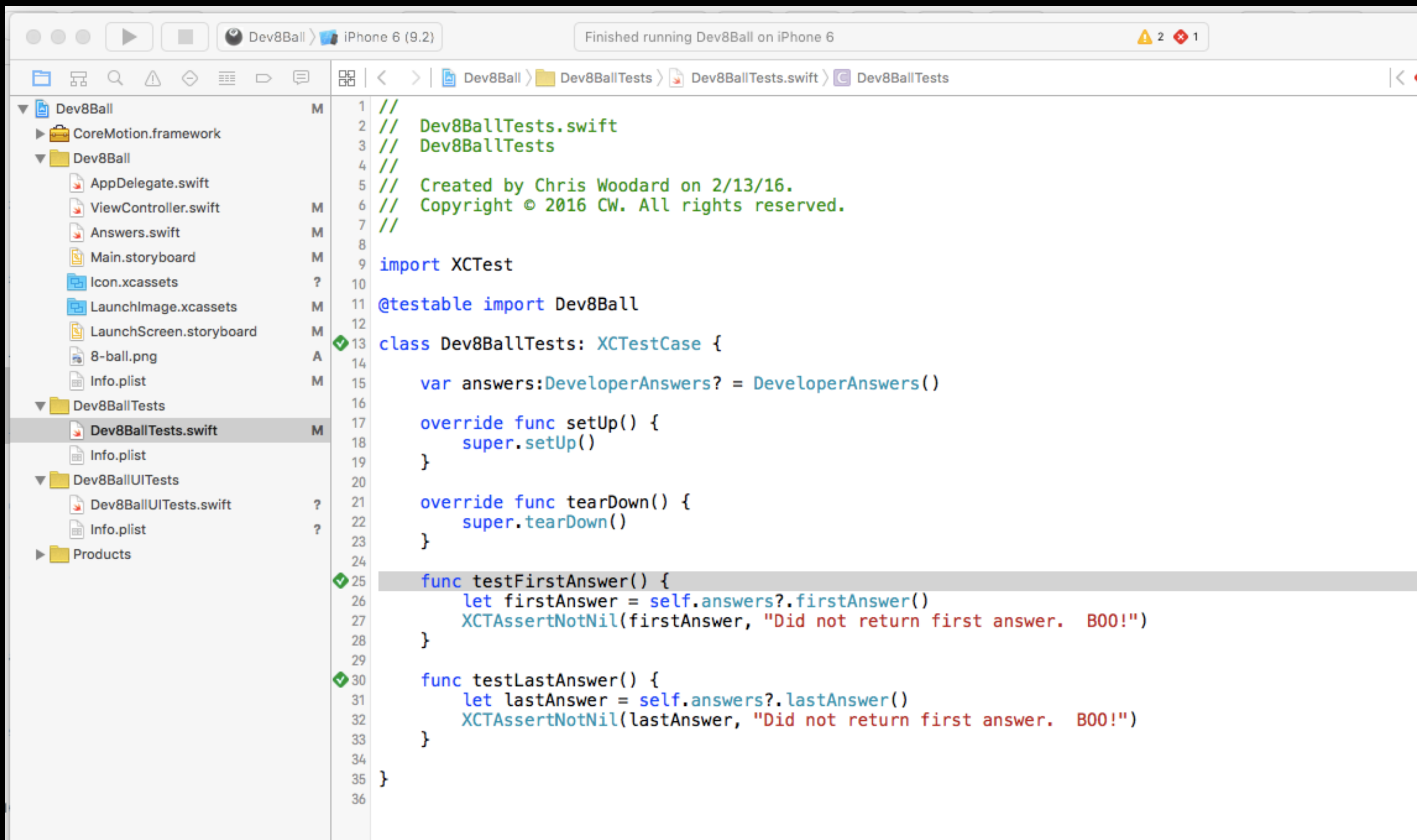
- **Automatic** testing
- **Isolation** of testing
- **Repeatability** of testing
- **Granularity** of testing

# Unit Testing

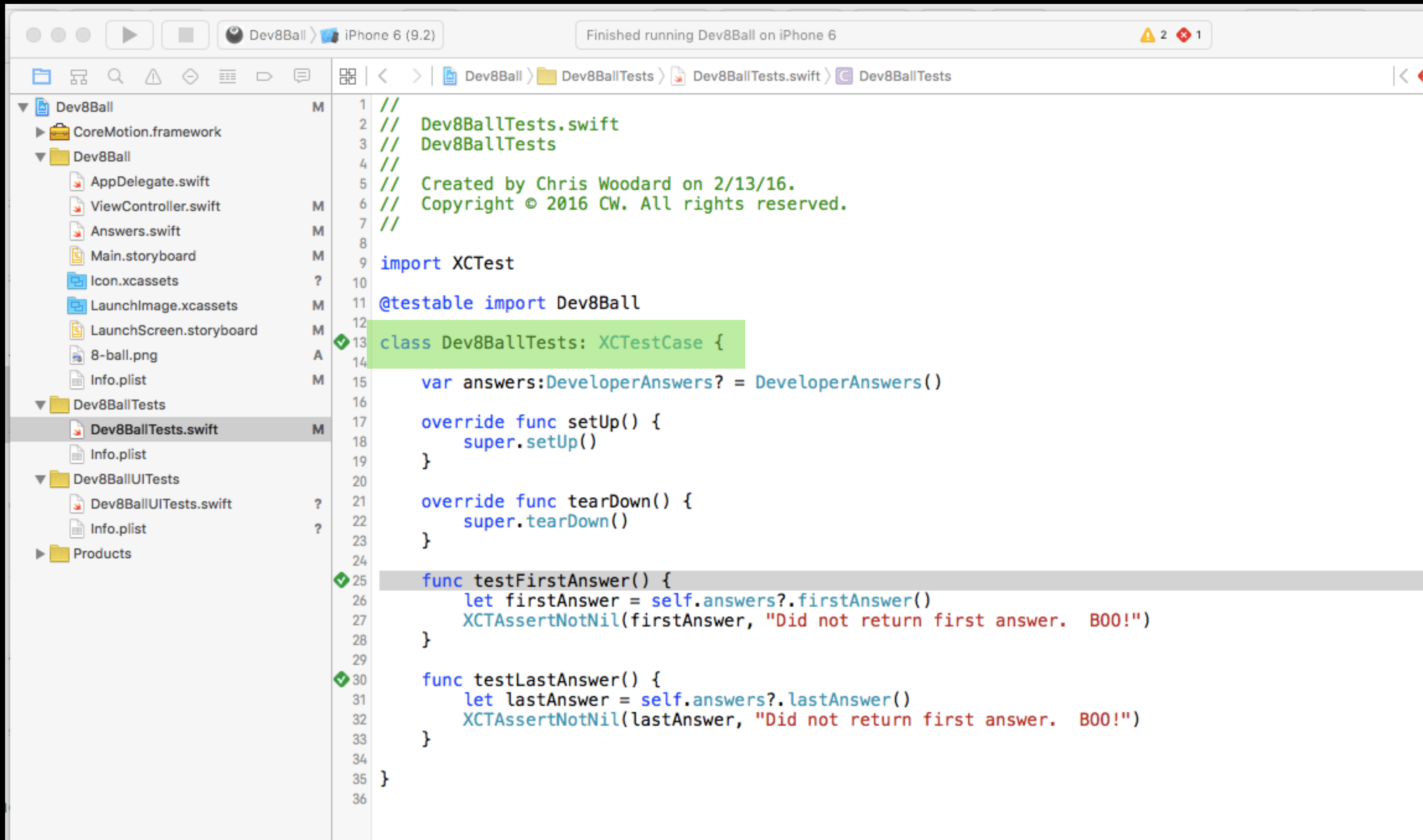
How?

- **Xcode Testing Bundle**
- **XCTestCase** class
- Individual methods within XCTestCase
- **XCTAssert** macros
- Helper methods

# Xcode Testing Bundle

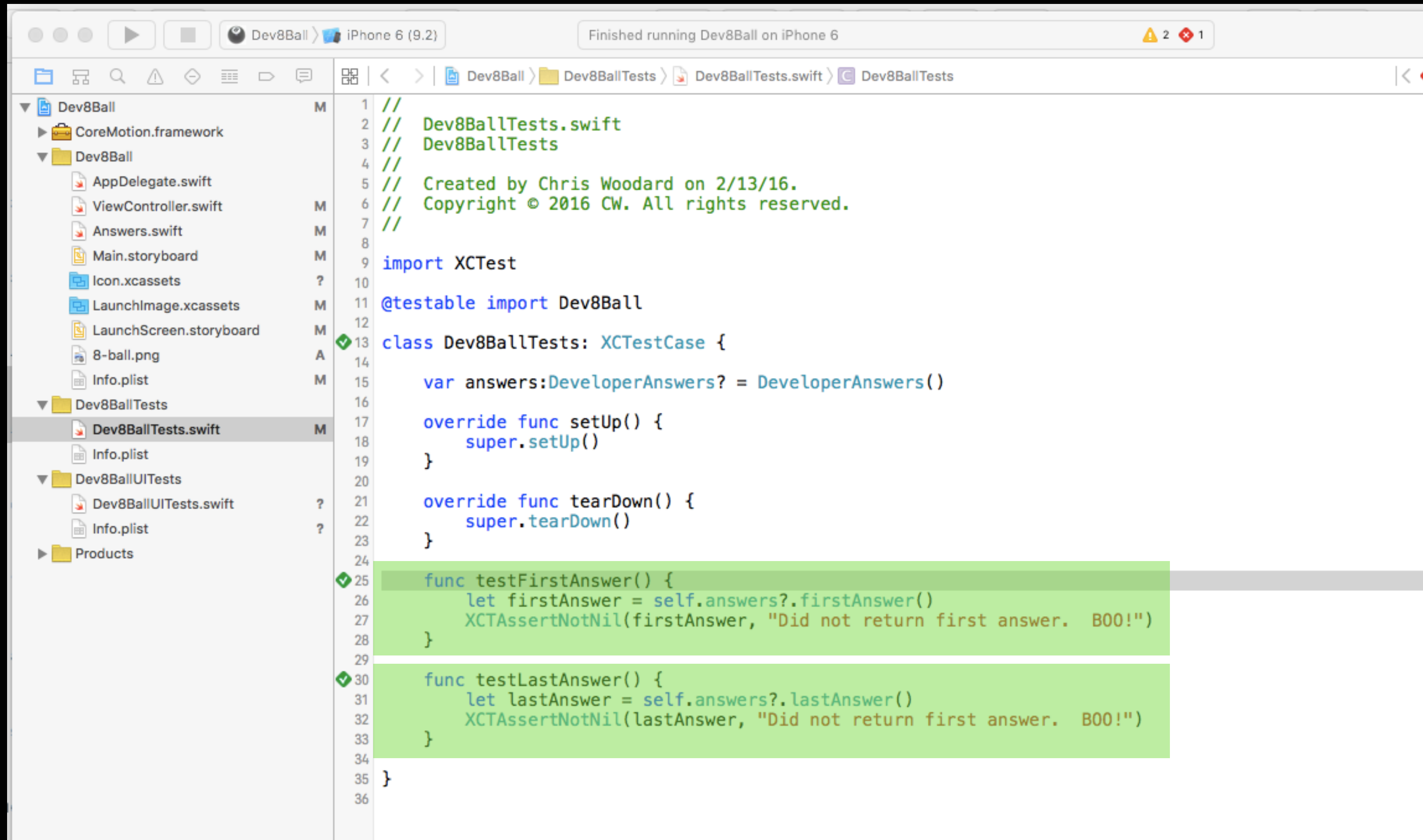


# XCTestCase

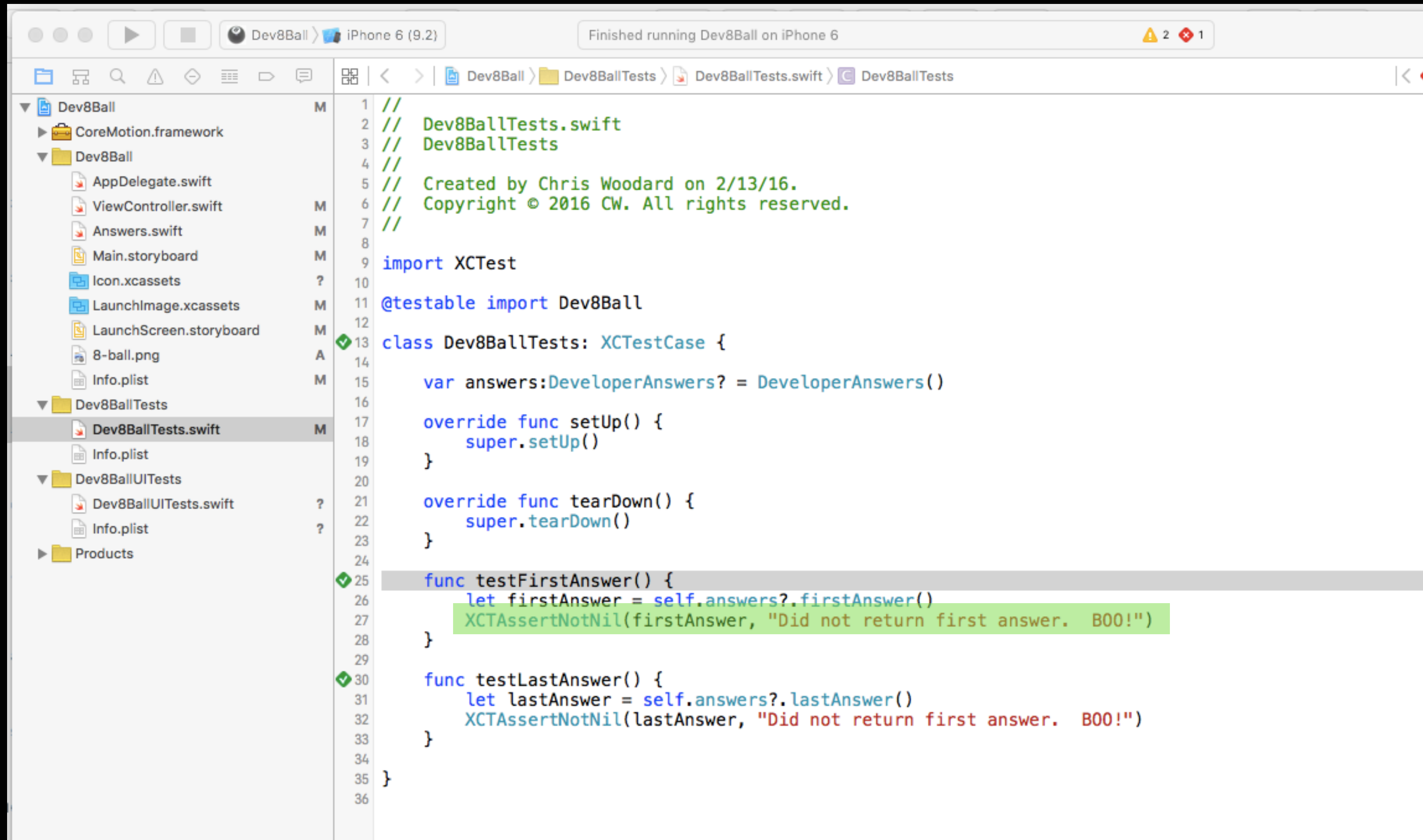




# Test methods within XCTestCase



# XCTAssert macros





# XCTAssert macros

**XCTFail**(message)

**XCTAssertNotEqual**(expression1, expression2, message)

**XCTAssertEqual**(expression1, expression2, message)

**XCTAssertEqualWithAccuracy**(expression1, expression2, accuracy, message)

**XCTAssertNil**(expression, message)

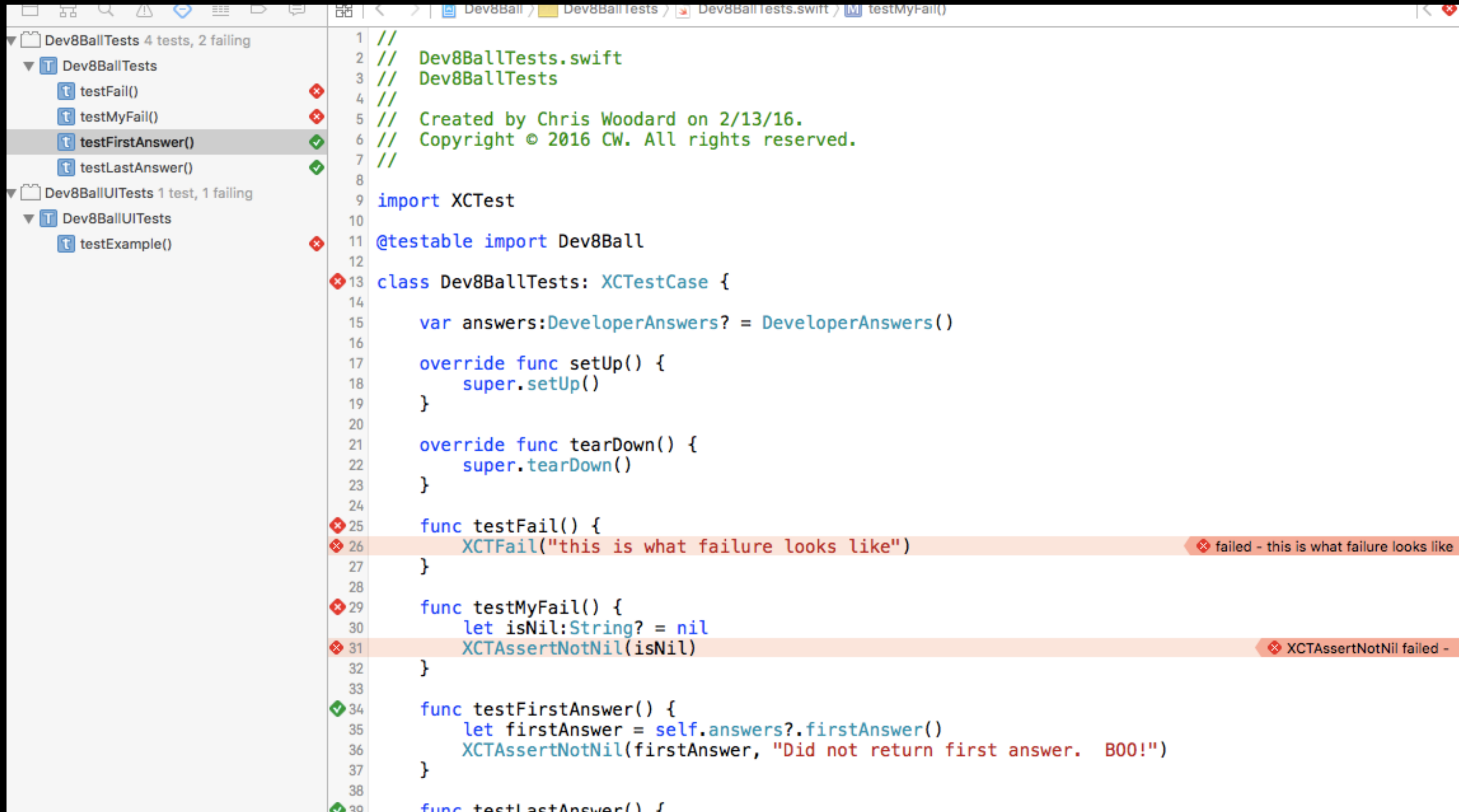
**XCTAssertFalse**(expression, message)

**XCTAssertTrue**(expression, message)

**XCTAssertGreaterThan**(expression1, expression2, message)

**XCTAssertLessThan**(expression1, expression2, message)

# XCTAssert macros



The screenshot shows the Xcode IDE with a Swift test file named `Dev8BallTests.swift`. The left sidebar displays a test hierarchy: `Dev8BallTests` (4 tests, 2 failing) and `Dev8BallUITests` (1 test, 1 failing). The `Dev8BallTests` group contains `testFail()`, `testMyFail()`, `testFirstAnswer()`, and `testLastAnswer()`. The `Dev8BallUITests` group contains `testExample()`. The main editor shows the Swift code for `Dev8BallTests`. The code includes comments, imports, and test methods. Two test methods are highlighted with red error bars: `testFail()` and `testMyFail()`. The `testFail()` method calls `XCTFail("this is what failure looks like")`, and the `testMyFail()` method calls `XCTAssertNotNil(isNil)`. The `testFirstAnswer()` method calls `XCTAssertNotNil(firstAnswer, "Did not return first answer. BOO!")`. The `testLastAnswer()` method is partially visible at the bottom.

```
1 //
2 // Dev8BallTests.swift
3 // Dev8BallTests
4 //
5 // Created by Chris Woodard on 2/13/16.
6 // Copyright © 2016 CW. All rights reserved.
7 //
8
9 import XCTest
10
11 @testable import Dev8Ball
12
13 class Dev8BallTests: XCTestCase {
14     var answers:DeveloperAnswers? = DeveloperAnswers()
15
16     override func setUp() {
17         super.setUp()
18     }
19
20     override func tearDown() {
21         super.tearDown()
22     }
23
24     func testFail() {
25         XCTFail("this is what failure looks like")
26     }
27
28     func testMyFail() {
29         let isNil:String? = nil
30         XCTAssertNotNil(isNil)
31     }
32
33     func testFirstAnswer() {
34         let firstAnswer = self.answers?.firstAnswer()
35         XCTAssertNotNil(firstAnswer, "Did not return first answer. BOO!")
36     }
37
38     func testLastAnswer() {
```

# Helper Methods

- Useful for building known **NSDate** values
- Useful for building known **NSArray**, **NSDictionary** or other collection values
- Loading databases with initial values
- Useful for generating more complex data structures

# Test-Driven Development

## What?

- Writing the unit tests first.
- Writing just enough code to make the tests pass.
- **Repeating until you're done.**

# Test-Driven Development

## Why?

- Test Coverage
- Design influence
- **DRY** - *Don't Repeat Yourself*
- **YAGNI** - *You Ain't Gonna Need It*



# Test-Driven Development

How?

- Write the tests first
- Build with ...
- Design for Testability

# Designing for Testability

- Divide and Conquer
- Use Design Patterns
- Adopt a Functional Style

# Adopting a Functional Style

- Do not use **shared state**.
- Do not use **shared mutable state**.
- A method's behavior depends **only** on its **inputs** and its **code**

# Shared State

```
var shouldPrint = YES  
var x:Int = 10  
var y:Int = 15  
var z:Int = 20
```

```
func computeSum1() -> Int {  
    let m = x + y  
    if(shouldPrint) {  
        NSLog("sum is \(m)")  
    }  
}
```

```
func computeSum2() -> Int {  
    let m = y + z  
    if(shouldPrint) {  
        NSLog("sum is \(m)")  
    }  
}
```

Shared state  
introduces a hidden  
dependency

Three orange arrows originate from a central point on the right and point to the 'shouldPrint' variable in the three code blocks on the left, illustrating that all three functions depend on this shared state.

# Shared Mutable State

```
var shouldPrint = YES
```

```
var x:Int = 10
```

```
var y:Int = 15
```

```
var z:Int = 20
```

```
func computeSum1() -> Int {
```

```
    let m = x + y
```

```
    if( m>5) {
```

```
        shouldPrint = NO
```

```
    }
```

```
    if(shouldPrint) {
```

```
        NSLog("sum is \ (m)")
```

```
    }
```

```
}
```

```
func computeSum2() -> Int {
```

```
    let m = y + z
```

```
    if(shouldPrint) {
```

```
        NSLog("sum is \ (m)")
```

```
    }
```

```
}
```

Shared **mutable** state  
introduces **two** hidden  
dependencies

The diagram consists of a central text block on the right that says 'Shared mutable state introduces two hidden dependencies'. From this block, four arrows point to specific parts of the code on the left. Three orange arrows point to the 'shouldPrint' variable in its initial declaration, its use in the 'if(shouldPrint)' condition of computeSum1, and its use in the 'if(shouldPrint)' condition of computeSum2. A red arrow points to the 'shouldPrint = NO' assignment statement within the computeSum1 function.



# Curing Shared Mutable State

```
var x:Int = 10  
var y:Int = 15  
var z:Int = 20
```

```
func computeSum1(shouldPrint:Boolean) -> Int {  
    let m = x + y  
    if(shouldPrint) {  
        NSLog("sum is \(m)")  
    }  
}
```

Parameters allow control  
w/o using shared state



```
func computeSum2(shouldPrint:Boolean) -> Int {  
    let m = y + z  
    if(shouldPrint) {  
        NSLog("sum is \(m)")  
    }  
}
```

# Functions Depend on Their Inputs

```
func computeSum1 (x:Int, y:Int, shouldPrint:Boolean) -> Int {  
  let m = x + y  
  if(shouldPrint) {  
    NSLog("sum is \(m)")  
  }  
}
```

Parameters provide input  
to the functions

```
func computeSum2(y:Int, z:Int, shouldPrint:Boolean) -> Int {  
  let m = y + z  
  if(shouldPrint) {  
    NSLog("sum is \(m)")  
  }  
}
```

# Functions Return Their Outputs

```
func computeSum1 (x:Int, y:Int, shouldPrint:Boolean) -> Int {  
    let m = x + y  
    if(shouldPrint) {  
        NSLog("sum is \(m)")  
    }  
    return m  
}
```

Returns yield values to  
whatever called the  
function

```
func computeSum2(y:Int, z:Int, shouldPrint:Boolean) -> Int {  
    let m = y + z  
    if(shouldPrint) {  
        NSLog("sum is \(m)")  
    }  
    return m  
}
```

# Methods, too Return Their Outputs

```
import Foundation

public class DeveloperAnswers : NSObject {

    let eightBallAnswers = [
        "That's Out of Scope",
        "Wrong Sprint",
        "Requirements Unclear",
        "Should Only Take An Hour",
        "It's 90% Done",
        "It's 98% Done",
        "You Changed the Scope",
        "Works on My Machine",
        "Ship It!",
        "It Needs Refactoring",
        "It Needs A Rewrite"
    ]

    public func firstAnswer() -> String? {
        return eightBallAnswers.first
    }

    public func lastAnswer() -> String? {
        return eightBallAnswers.last
    }

    public func pickRandomAnswer() -> String {
        srand(UInt32(time(nil)))
        let i:Int = Int(rand()%Int32(eightBallAnswers.count))
        return eightBallAnswers[i]
    }
}
```

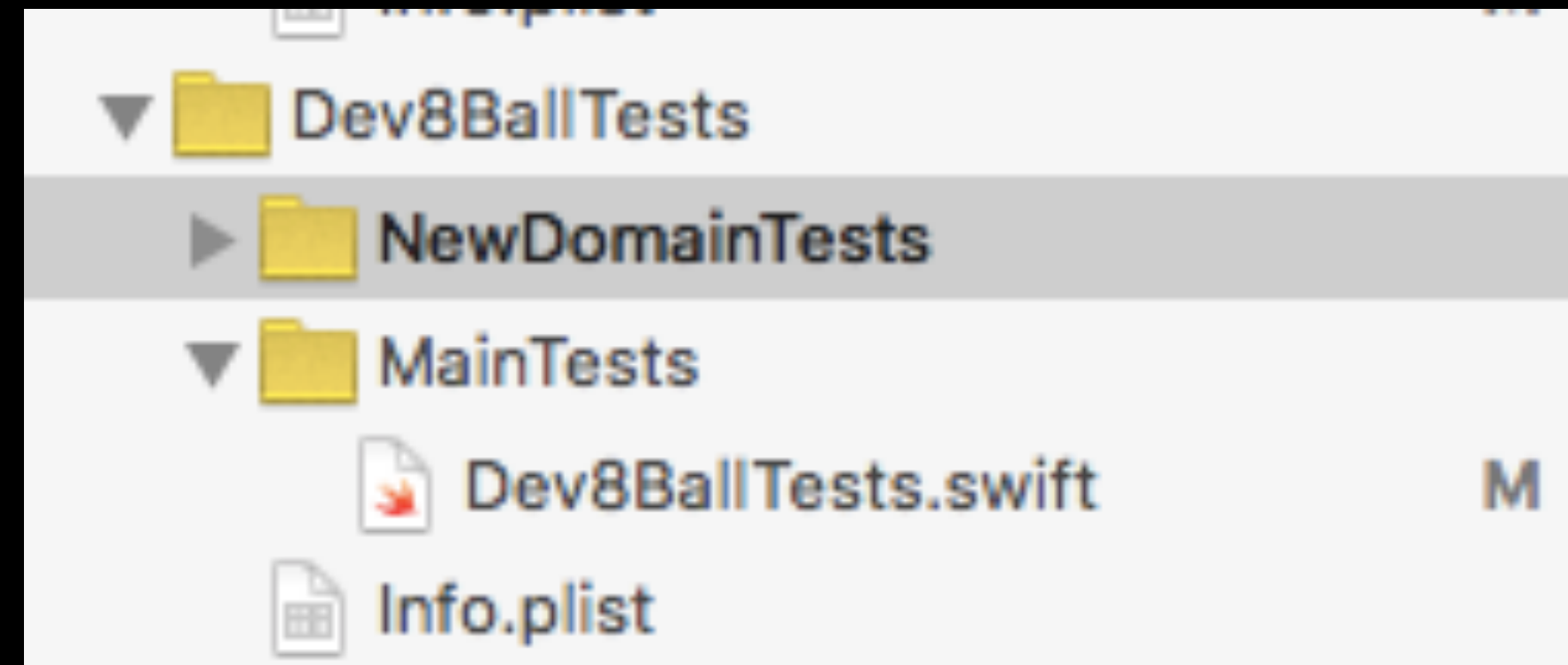
# Organizing Your Tests



# XCTestCase encapsulates unit tests

- One XCTestCase class per related group of tests
- Test methods are run in **parallel**; don't share state among the tests
- setUp and tearDown methods are called before and after each test method

# Don't be afraid to use groups



- Put groups of related XCTestCase classes in Xcode project groups
- Also put in any resources you need - *JSON* files, images, static database files.