**Unit test**

(short cut chạy unit test: ⌘ + U)

Test case thường bao gồm setup và teardown functions, chúng được gọi trước và sau các test function. Nó cho phép set up cái điều kiện trước (prerequisite) trước khi chạy test và clean nhưng thứ có trước đó sau khi test chạy xong.

Đặt tên class và function theo chức năng của unit test đó.

**Asserting**

XCAssert

XCTAssertTrue

XCTAssertFalse

XCTAssertNotEqual

XCTAssertNil

XCTAssertNotNil

Mặc định thì target muốn test sẽ chưa được thêm vào test target, ta phải import main target qua @testable. Nó cho phép truy cập đến các internal của main target.

|  |
| --- |
| struct Student { |
|  | /// The name of the student. |
|  | let name: String |
|  | /// The list of all grades for the student. |
|  | let grades: [Float] |
|  | /// The current grade point average of the student. |
|  | var gpa: Float? { |
|  | guard !grades.isEmpty else { return nil } |
|  | return grades.reduce(0, +) / Float(grades.count) |
|  | } |
|  | } |

|  |
| --- |
| import XCTest |
|  | @testable import MediumArticle |
|  |  |
|  | class StudentTests: XCTestCase { |
|  |  |
|  | // MARK: - gpa (computed property) |
|  |  |
|  | /// Students with no grades should not have a GPA. |
|  | func testGPAWithNoGrades() { |
|  | let student = Student(name: "test", grades: []) |
|  | XCTAssertNil(student.gpa) |
|  | } |
|  |  |
|  | /// If there is only one grade, the GPA should be equal to the grade. |
|  | func testGPAWithOneGrade() { |
|  | let student = Student(name: "test", grades: [4]) |
|  | XCTAssertEqual(student.gpa, 4) |
|  | } |
|  |  |
|  | /// The GPA should correctly average with ten grades. |
|  | func testGPAWithTenGrades() { |
|  | let student = Student(name: "test", grades: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) |
|  | XCTAssertEqual(student.gpa, 5.5) |
|  | } |
|  |  |
|  | /// The GPA should correctly average - even with extreme grades. |
|  | func testGPAWithExtremeNumbers() { |
|  | let student = Student(name: "test", grades: [Float(Int.min), Float(Int.max)]) |
|  | XCTAssertEqual(student.gpa, 0) |
|  | } |
|  | } |

**Expectation (Test Async)**

|  |
| --- |
| func retrieveAlumni(\_ completion: @escaping ((Result<[Student], NetworkError>) -> Void)) { |
|  | Network.shared.performRequest("alumni/all") { response in |
|  | guard let alumni = response as? [Student] else { |
|  | completion(.failure(.invalidResponse)) |
|  | return |
|  | } |
|  | completion(.success(alumni)) |
|  | } |
|  | } |

Test:

|  |
| --- |
| import XCTest |
|  | @testable import MediumArticle |
|  |  |
|  | class SchoolTests: XCTestCase { |
|  |  |
|  | /\* Other tests omitted \*/ |
|  |  |
|  | // MARK: - retrieveAlumni(\_:) |
|  |  |
|  | /// Retrieving the alumni should fetch the two alumni for the school. |
|  | func testRetrieveAlumni() { |
|  | let school = School(students: []) |
|  |  |
|  | school.retrieveAlumni { result in |
|  | switch result { |
|  | case .success(let alumni): |
|  | XCTAssertEqual(alumni.count, 2) |
|  | case .failure: |
|  | XCTFail() |
|  | } |
|  | } |
|  | } |
|  | } |

ở đây test case sẽ luôn đúng do mặc định nếu không có Assert nào fail thì testcase sẽ success. Do các assert được xử lý ở luồng khác sau khi test case chạy xong.

TCTestExpectation: dùng để báo cho unit test đợi cho công việc xử lý xong thì chạy unit test tiếp. Ta có thể gọi expectation(description:) để tạo 1 expectation, sau đó ta phải báo cho test function đợi đến khi expectation được fulfill bằng function waitForExpectation(timeout: handler:)

|  |
| --- |
| func testRetrieveAlumni() { |
|  | let school = School(students: []) |
|  | var result: Result<[Student], NetworkError>? |
|  |  |
|  | // Create the expectation. |
|  | let expectation = self.expectation(description: "Waiting for the retrieveAlumni call to complete.") |
|  |  |
|  | // Perform the asynchronous call. |
|  | school.retrieveAlumni { response in |
|  | // Save the response. |
|  | result = response |
|  | // Fulfill the expectation. |
|  | expectation.fulfill() |
|  | } |
|  |  |
|  | // Wait for expectations for a maximum of 2 seconds. |
|  | waitForExpectations(timeout: 2) { error in |
|  | XCTAssertNil(error) |
|  | switch result { |
|  | case .success(let alumni): |
|  | XCTAssertEqual(alumni.count, 2) |
|  | case .failure, |
|  | .none: |
|  | XCTFail() |
|  | } |
|  | } |
|  | } |

**Inverted Expectations**

Nó có chức năng ngược lại với expectation. Nếu như expectation được fullfil thì sẽ fail. Nghĩa là test function sẽ hết time out thì sẽ pass.

Để tạo inverted expectation ta gắn biến isInverted = true.

|  |
| --- |
| func testRetrieveAlumniNoFailure() { |
|  | let school = School(students: []) |
|  |  |
|  | // Create the expectation. |
|  | let expectation = self.expectation(description: "Expecting the retrieveAlumni call not to fail.") |
|  | expectation.isInverted = true |
|  |  |
|  | // Perform the asynchronous call. |
|  | school.retrieveAlumni(success: { \_ in /\* ignore success \*/ }, |
|  | failure: { \_ in expectation.fulfill() }) |
|  |  |
|  | // Wait for expectations for a maximum of 2 seconds. |
|  | waitForExpectations(timeout: 2, handler: nil) |
|  | } |

Để Test các FAF functions không có completion block ta phải khoá thread của test function lại đến khi công việc xử lý xong.

|  |
| --- |
| func doSomethingAsynchronous() { |
|  | DispatchQueue(label: "some queue").async { [weak self] in |
|  | sleep(1) |
|  | self?.didComplete = true |
|  | } |
|  | } |

Với func trên thì ta không thể inject queue vào function được

|  |
| --- |
| func doSomethingAsynchronous(queue: DispatchQueue = DispatchQueue(label: "some queue")) { queue.async { [weak self] in |
|  |  |
| sleep(1) |
| self?.didComplete = true |
| } |
| } |

Test

func testDoSomethingAsynhronous() {

let school = School(students: [])

// Create our test queue

let queue = DispatchQueue(label: "test-queue")

// didComplete is initially false.

XCTAssertFalse(school.didComplete)

// Inject our test queue.

school.doSomethingAsynchronous(queue: queue)

// Synchronize the queue to wait for it to complete.

queue.sync { /\* Do nothing, just synchronize \*/ }

// didComplete should now be true.

XCTAssertTrue(school.didComplete)

}

**Mocking**

Tạo ra giả định cho 1 lớp đối tượng, cho phép thay đổi giả định ban đầu khi đối tượng hoạt động như mong muốn.

Nhằm mục đích giả định các thuộc tính của 1 đói tượng mà class muốn test bị phụ thuộc vào nó. (Vd: class APIManager bị phụ thuộc vào URLSession thì ta mock URLSession để test API manager)

Có 2 loại mock là complete và partial

**Partial**

Mock trực tiếp từ class muốn mock để implement những thuộc tính mong muốn.

Ví dụ: muốn mock API request thì phải viết 1 class mock URLSessionDataTask để gỉa định việc khởi tạo 1 data task.

// We create a partial mock by subclassing the original class

class URLSessionDataTaskMock: URLSessionDataTask {

private let closure: () -> Void

init(closure: @escaping () -> Void) {

self.closure = closure

}

// We override the 'resume' method and simply call our closure

// instead of actually resuming any task.

override func resume() {

closure()

}

}

Sau đó mock URLSession để giả định việc respose data và error từ API

class URLSessionMock: URLSession {

typealias CompletionHandler = (Data?, URLResponse?, Error?) -> Void

// Properties that enable us to set exactly what data or error

// we want our mocked URLSession to return for any request.

var data: Data?

var error: Error?

override func dataTask(

with url: URL,

completionHandler: @escaping CompletionHandler

) -> URLSessionDataTask {

let data = self.data

let error = self.error

return URLSessionDataTaskMock {

completionHandler(data, nil, error)

}

}

}

Sau đó phải đảm bảo rằng ta có thể inject mock class vào class muốn test

class NetworkManager {

private let session: URLSession

// By using a default argument (in this case .shared) we can add dependency

// injection without making our app code more complicated.

init(session: URLSession = .shared) {

self.session = session

}

func loadData(from url: URL,

completionHandler: @escaping (NetworkResult) -> Void) {

let task = session.dataTask(with: url) { data, \_, error in

let result = data.map(NetworkResult.success) ?? .failure(error)

completionHandler(result)

}

task.resume()

}

}

Cuối cùng là ta viết test case cho class đó.

class NetworkManagerTests: XCTestCase {

func testSuccessfulResponse() {

// Setup our objects

let session = URLSessionMock()

let manager = NetworkManager(session: session)

// Create data and tell the session to always return it

let data = Data(bytes: [0, 1, 0, 1])

session.data = data

// Create a URL (using the file path API to avoid optionals)

let url = URL(fileURLWithPath: "url")

// Perform the request and verify the result

var result: NetworkResult?

manager.loadData(from: url) { result = $0 }

XCTAssertEqual(result, .success(data))

}

}

**Complete**

Mock qua các abstract, điều kiện cần là class test chỉ phụ thuộc vào abstract. Khi đó ta sẽ viết các class mock implement từ abstract để inject vào test class.

Ưu điểm complete của nó so với partial là nó sẽ không phải implement những thuộc tính không mong muốn từ class muốn mock.

Vd: như ví dụ trên

Đầu tiên ta phải abstract func loaddata

protocol NetworkSession {

func loadData(from url: URL,

completionHandler: @escaping (Data?, Error?) -> Void)

}

Sau đó cho URLSession implement abstract trên hoặc viết 1 class implement abstract trên.

extension URLSession: NetworkSession {

func loadData(from url: URL,

completionHandler: @escaping (Data?, Error?) -> Void) {

let task = dataTask(with: url) { (data, \_, error) in

completionHandler(data, error)

}

task.resume()

}

}

Viết class mock implement abstract

class NetworkSessionMock: NetworkSession {

var data: Data?

var error: Error?

func loadData(from url: URL,

completionHandler: @escaping (Data?, Error?) -> Void) {

completionHandler(data, error)

}

}

Sau đó làm cho class muốn test chỉ phụ thuộc vào abstract

class NetworkManager {

private let session: NetworkSession

init(session: NetworkSession = URLSession.shared) {

self.session = session

}

func loadData(from url: URL,

completionHandler: @escaping (NetworkResult) -> Void) {

session.loadData(from: url) { data, error in

let result = data.map(NetworkResult.success) ?? .failure(error)

completionHandler(result)

}

}

}

**Fake**

Cũng giống như mock, fake cũng implement từ protocol mà test class phụ thuộc nhưng sử dụng in-memory để lưu dữ liệu thay cho database (vd: Array, dictionary).

**Stub**

Trứa responses và answers được định nghĩa trước khi truyền vào object muốn test.

Vd: Sử dụng closure để khởi tạo logic mong muốn của class mà class muốn test bị phụ thuộc.

|  |
| --- |
| protocol Database { |
|  | func addUser(name: String) -> Bool |
|  | } |
|  | class DatabaseStub { |
|  | var addUserAction: ((String)) -> Bool)? |
|  | func addUser(name: String) -> Bool { |
|  | return addUserAction?(name) ?? false |
|  | } |
|  | } |

**Sky**

Giỗng stub nhưng ghi lại 1 số thông tin sử dụng để test. Vd: số lần method được gọi,…

**SwiftyMocky**

Pod ‘SwiftyMocky’

Dùng ở bản 3.4.0 do bản 4.0.1 đang gặp lỗi.

**Mockfile**

SwiftyMocky dùng file: mocky.yml để config (cấu trúc mới: <https://cdn.rawgit.com/MakeAWishFoundation/SwiftyMocky/4.0.1/docs/mockfile.html> )

Cấu trúc:

sources: # 2.1.

include: # 2.1.1

- ./Cloudy

templates:

- ./Pods/SwiftyMocky/Sources/Templates

output: # 2.2

./CloudyTests/Test Cases/Mock.generated.swift

targets: # 2.3 (optional)

- CloudyTests

* Sources: định nghĩa files/directories để scan
* Include: list các included files/directories
* Output: vị trí của file mock sau khi sinh ra.
* Targets: target của file mock

Cách giúp SwiftMocky nhận biết protocol nào để mock gán “//sourcery: AutoMockable” trước protocol:

1. AutoMockable annotation

*//sourcery: AutoMockable*

**protocol** ToBeMocked {

*// ...*

}

Với các objc protocol:

*//sourcery: AutoMockable*

*//sourcery: ObjcProtocol*

**@objc** **protocol** NonSwiftProtocol {

*// ...*

}

1. AutoMockable protocol

Tạo 1 dummy protocol

**protocol** AutoMockable { }

Sau đó cho các protocol muốn tạo mock kế thừa trực tiếp từ protocol trên

**protocol** ToBeMocked: AutoMockable {

*// ...*

}

1. Manual annotation

Trong 1 số trường hợp mà ta k muốn sử dụng Mock.generated.swift, or need to add some additional code to generated mock. Ta có thể tạo 1 base cho mock implementation.

**import** Foundation

**import** SwiftyMocky

**import** XCTest

**@testable** **import** TestedApp

*// sourcery: mock = "ToBeMocked"*

**class** SomeCustomMock: ToBeMocked, Mock {

Your custom code can go here

*// sourcery:inline:auto:ToBeMocked.autoMocked*

Generated code goes here**...**

*// sourcery:end*

}

Tiến hành tạo file mock:

Cách tiện nhất để sinh ra file mock là tạo 1 file Rakefile.rb với nội dung sau:

*# Rakefile*

task :mock **do**

sh "Pods/Sourcery/bin/Sourcery.app/Contents/MacOS/Sourcery --config mocky.yml"

**end**

task :mock\_watcher **do**

sh "Pods/Sourcery/bin/Sourcery.app/Contents/MacOS/Sourcery --config mocky.yml --watch"

**end**

sau đó sử dụng terminal: rake mock

sau khi

.Value() để làm gì?

Link:

* Part 1: <https://medium.com/swlh/unit-testing-in-swift-the-fundamentals-eec805766349>
* Stub: https://medium.com/@londeix/stubbing-in-pair-with-swift-compiler-c951770a295b