Mô hình MVVM trong Swift

Có thể bạn cũng đã biết về mô hình MVC(Model-View-Controller) nhưng trong iOS có thể nó sẽ thành Massive-View-Controller vì nếu bạn đã làm một dự án lớn thì có thể các view controllers có thể sẽ hơn 1000 dòng code, đấy chính là vấn đề tại sao nó biến các view controllers ở lên hỗ độn và khi đọc rất khó chịu.

Dưới đấy là cấu trúc theo lý thuyết MVC:

i)

Nhưng trong thực tế đôi khi nó sẽ khác:

i)

Như các bạn thấy thành phần Controllers sẽ phình to ra khi dự án phát triển.

Trong bài này hôm nay chúng ta sẽ tìm hiểu về MVVM(Model-View-ViewModel). Cấu trúc nó sẽ như sau:

i)

Bây giờ thì thành phần Controller có nhiện vụ hiển thị những gì mà View Model muốn hiển thị và nó không quan tâm và cũng không biết về Model. Các thành phần như UILabel, UITextField,… nó chỉ hiển thị dữ liệu mà ViewModel cung cấp.

Vậy lợi ích ở đây là gì?

Ví dụ: Controller sẽ không phình to nữa. Nếu muốn chỉnh sửa lại Model, chúng ta đơn giản chỉ cần làm ở View Model. Ví dụ khi thay đổi và hiển thị các định dạng khác nhau với NSDate. Thường thì chúng ta sẽ chỉnh sửa nó ngay trong ViewController, có nghĩa là ViewController sẽ phình to ra. Thay vì thế chúng ta sẽ xử lý định dạng để hiển thị ngày tháng ở trong ViewModel nó sẽ chỉ rõ ra class/file nào chịu trách nhiệm cho việc chuyển đổi từ NSData sang String.

Lợi ích nữa khi sử dụng MVVM là ứng dụng sẽ dễ kiểm thử hơn, bạn có thể dễ dàng viết các tests cho các ViewModels vì các thành phần liên kết dưới loose coupling.

Ví dụ:

Model, ViewModel, Unit Test

Tôi sẽ tạo class Car

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Car {    var model: String?    var make: String?    var horsepower: Int?    var photoURL: String?      init(model: String, make: String, horsepower: Int, photoURL: String) {      self.model = model      self.make = make      self.horsepower = horsepower      self.photoURL = photoURL    }  } |

Và tiếp theo là ViewModel tôi đặt tên là CarViewModel:

|  |  |
| --- | --- |
| 2  3  4  5  6  7  8  9  10  11  12 | class CarViewModel {    private var car: Car?    var modelText: String?    var makeText: String?    var horsepowerText: String?    var titleText: String?    var photoURL: NSURL?      init(car: Car) {      self.car = car    }  } |

Bạn có thể thấy các thuộc tính của 2 classes này khá giống nhau, tiếp theo tôi sẽ viết TDD test(test driven development)

i)

Chúng ta sẽ khởi tạo đối tượng Ferrari F12 như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | func testCarViewModelWithFerrariF12() {    let ferrariF12 = Car(model: "F12", make: "Ferrari", horsepower: 730, photoURL: "http://auto.ferrari.com/en\_EN/wp-content/uploads/sites/5/2013/07/Ferrari-F12berlinetta.jpg")    let ferrariViewModel = CarViewModel(car: ferrariF12)    XCTAssertEqual(ferrariViewModel.modelText, "F12")    XCTAssertEqual(ferrariViewModel.makeText, "Ferrari")    XCTAssertEqual(ferrariViewModel.horsepowerText, "730 HP")    XCTAssertEqual(ferrariViewModel.photoURL, NSURL(string: ferrariF12.photoURL!))    XCTAssertEqual(ferrariViewModel.titleText, "Ferrari F12")  } |

Bây giờ bạn có thể ấn command+U để chạy thì nó sẽ báo lỗi, nhưng không sao chúng ta sẽ sửa nó.

class CarViewModel {

private var car: Car?

var modelText: String? {

return car?.model

}

var makeText: String? {

return car?.make

}

var horsepowerText: String? {

guard let horsepower = car?.horsepower else {

return nil

}

return "\(horsepower) HP"

}

var titleText: String? {

guard let make = car?.make, model = car?.model else {

return nil

}

return "\(make) \(model)"

}

var photoURL: NSURL? {

guard let photoURL = car?.photoURL else {

return nil

}

return NSURL(string: photoURL)

}

init(car: Car) {

self.car = car

}

}

Bây giờ chúng ta có thể thấy lợi ích của việc sử dụng MVVM. Class Car khá đơn giản, nhưng CarViewModel lại thêm một số các phần getter của các thuộc tính. Ví dụ với biến titleText, nó liên kết với thuộc tính của Car tương tự với horsepowerText,… Bây giờ bạn có thể ấn command+U để chạy test này.

Refactoring