SWIFT BEST PRACTICES

MVC – DIFFICULTY: EASY

INTRODUCTION

The MVC architecture has developers split their code into three groups that don't overlap: Model, View, and Controller. There is no one single definition of what MVC means, but on Apple platforms its common to see developers write controllers that read model data and send it back and forth between views.

THE PROBLEM

Apple gave us **UIViewController** on iOS and **NSViewController** on macOS, which mixes together the concepts of view and controller in one. Worse, Apple's sample code tends to lump functionality directly into view controllers, meaning that they become overloaded with work: delegates, data sources, animations, navigation, networking, model handling, and more.

These massive view controllers are hard to read, hard to maintain, hard to re-use, and hard to test, and yet are all too common on our platform. They no longer can really be said to use “MVC”, but instead us “Just C” because there is little to no M or V.

THE SOLUTION

View controllers should not contain view or model code – any such code should be moved to **UIView** subclasses or your model layer. For example, it's common to see views being created and configured inside **viewDidLoad()**, despite that method specifically meaning that the view has *finished* loading.

A better solution is to take any view code from there and put it into a **UIView** subclass, then override **loadView()** on your view controller like this:

```
class MyViewController: UIViewController {
    let myView = MyView()

    func loadView() {
        view = myView
    }
}
```

This helps build up the view part of your architecture so it has a meaningful part.

Similarly, you can and should transfer model-related work to your model layer. It's common to see view controllers have code like this:

```
func printData(index: Int) {
   let item = items[index]
   let message = "#\(item.number). \(item.title): \(item.subtitle)"
   print(message)
}
```

If you're transforming model data into some other piece of model data, that's best done either directly on the model itself or using a view model that handles formatting. For example:

```
struct Item {
   var number: Int
   var title: String
   var subtitle: String

   var formattedTitle: String {
      return "#\(number). \(title): \(subtitle)"
   }
}
```

THE CHALLENGE

Our app suffers from a very poor split of MVC: the **Project** struct is almost empty, and **DetailViewController** class does all sorts of layout inside **viewDidLoad()**. So, lots of work happens in view controllers, where it's harder to test.

You can fix this by moving work out of view controllers:

1. The **makeAttributedString()** method of **ViewController** transforms model data (strings) into different model data (an attributed string) – move that so it exists on the model itself, or wrap **Project** in a view model that does something similar.
2. Move all the view code from the **viewDidLoad()** method of **DetailViewController** into a dedicated **UIView** subclass, then make the view controller use that for its view.

Tip: give your **UIView** subclass a custom initializer that accepts a **Project** instance so it can configure itself.