Maximum Blue Green

iOS Architecture Patterns

By Kevin Destin, ECE '20

Introduction

SmartBell aims to provide a smarter way for an athlete to store and analyze metrics associated with their workouts. We initially targeted a limited number of exercises performed with a barbell and provided an automated way to classify exercise and count repetitions. Users interact with that data through a mobile app. Therefore, a key aspect of building SmartBell was the development of an iOS Application.

This tech note aims to provide a concise introduction to common iOS architecture patterns, by first providing relevant background information which will lead to a discussion of their merits through the perspective of our design requirements for the SmartBell application.

Background

Architectural Patterns

An architectural pattern "provides a set of predefined subsystems, their responsibilities, and includes rules and guidelines for organizing the relationships between them" (Franchitti). In the context of iOS development, architectural patterns typically refer to the schema that describes how the user interface (UI), application data, and application logic intermingle. While there are countless of these patterns, we will introduce 2 for the sake of further discussion.

Model-View-Controller (MVC)

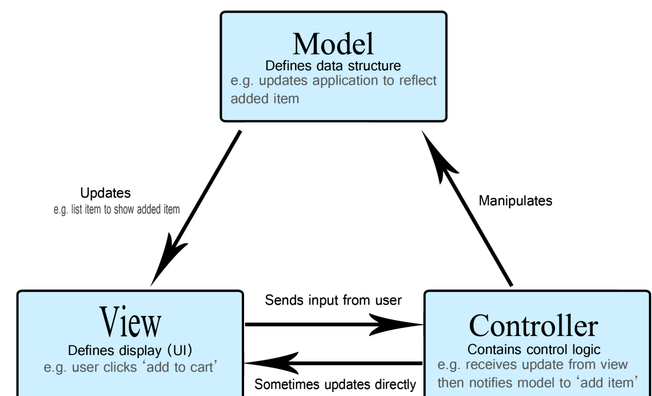

Figure 1. MVC Diagram (Mozilla)

- The Model is an abstraction that encapsulates data needed by the application, and methods to update it.
- The View is an element of the UI responsible for representing some data to the user. This can be a chart, a button, etc...
- The Controller contains control logic, and is responsible for updating both the Model and View as necessary

Model-View-ViewModel (MVVM)

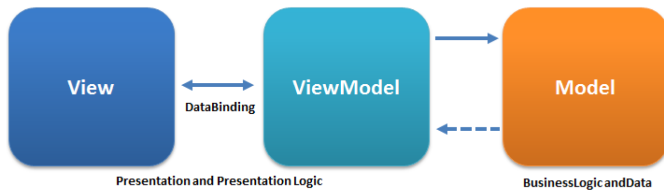

Figure 2. MVVM Diagram (Wikipedia)

- Model - Identical to MVC
- View - Identical to MVC
- The ViewModel is a middleman between the View and the Model. It exposes data that should be presented by the View, and interfaces with the Model to perform updates and retrieve data.

UI Frameworks

UI frameworks provide the required infrastructure to develop mobile applications. Specifically, this includes infrastructure to display UI elements (Views), handle input, and run applications, etc...

We can further subdivide frameworks by the type of application that uses them: hybrid or native.

Hybrid Application

Hybrid applications are applications built with UI frameworks that allow developers to use web technologies to develop native applications. This provides developers with the flexibility to write an app much like they would build a website and have a single code base that can be natively deployed to Android and iOS devices, along with the web. This can make hybrid development much quicker than separately developing native applications for each target platform. However, hybrid apps are not without their downsides: they are often slower than native applications and can lack functionality accessible to native code. React Native, Xamarin, Flutter are all frequently used UI Frameworks to develop hybrid applications.

Native Application

A native application is an application built using the specific programming language(s) intended for the target platform. On iOS, this would typically be done with UIKit and/or SwiftUI. UIKit has been around since the inception of iOS, being the default UI framework. SwiftUI is much newer, released in public beta during Apple's 2019 developer conference (WWDC). UIKit and SwiftUI are both designed to encourage the use of an architecture pattern: UIKit is based on the MVC, while SwiftUI facilitates the use of MVVM.

Discussion

Ideally, we wanted our code base to be easy to both test and maintain. Code that is both easy to test and to maintain has as little "coupling" as possible. Additionally, maintainable code should be readable and have clear intent.

Coupling

Coupling is a measure interdependence between units of code. Code that is highly coupled consists of modules or functions that depend strongly on each other. Code that has minimal coupling has as little interdependent modules as possible.

```
struct Foo {

    func operateOn(_ a: Bar, _ b: Baz){}
}

struct Bar {}

struct Baz {}
```

Figure 3. Example of coupling

In Figure 3, we can see a simple example of coupling: struct Foo is coupled with structs Bar and Baz, since the implementation of Foo depends on the implementations of Bar and Baz.

We can imagine how changing either structs, Bar and Baz, could mandate that the function operateOn

also be updated. It follows that high cohesion increases the cost of making changes to (i.e. refactoring) code, since the need to change one module often has a domino effect and requires that multiple other modules also be refactored.

Functionally, highly coupled code can work just as well as code that has minimal coupling. However, code bases are rarely static. They evolve as requirements change, which requires that code must be refactored and tested once again. Highly coupled code inflates the time necessary to make meaningful changes to a codebase.

Additionally, high degrees of coupling can also make testing much more cumbersome. This because all depended modules must be included and instantiated in tests.

Given the disadvantages associated with highly coupled code, we sought to pick an architecture pattern that minimized coupling

Model-View-Controller (MVC)

As shown in Figure 1, the View forwards user input to the controller. The controller can manipulate the Model and/or update the View. And the Model can also update the View. Each component in the triangle is coupled with at least one other component. With this comes all the disadvantages of high coupling. For example, any changes to the View may require changes to at least one of the Model or Controller. Writing tests for any of the three components will likely require that all three be present within the test code. This can quickly become cumbersome in large applications.

Model-View-ViewModel (MVVM)

MVVM tackles the coupling seen in MVC by shifting responsibility for data transfer. Rather than require that either the View or the ViewModel manually propagate data, it instead occurs through data binding. Data binding is a technique that synchronizes changes to data for all consumers of that data. This notably decouples the View from the ViewModel; both only need be aware of the data they are bound to. The ViewModel is however responsible for manipulating the Model and propagating any changes back to the View. But this means that the Model is not dependent on the View or the Model. Using MVVM, the mutual independence of the View and the Model provides the valuable opportunity to both develop and test UI code separately from business logic.

Readability

Readable code is easier to read, understand, and therefore to maintain. Given that we had the choice between SwiftUI and UIKit, examining whether any of the two had any stylistic advantages was also important to examine.

```
let buyBttn = UIButton(type: .custom)
buyBttn.setTitle("Buy", for: .normal)
buyBttn.addTarget(self, action: #selector(buyShip), for:
.touchUpInside)
buyBttn.backgroundColor = UIColor(red: 0.47, green: 0.32,
blue:0.70, alpha: 1)
buyBttn.setTitleColor(.white, for: .normal)
buyBttn.titleLabel?.font = UIFont.referredFont(forTextStyle:
.headline)
buyBttn.contentEdgeInsets = UIEdgeInsets(top: 20, left: 20,
bottom: 20, right: 20)
buyBttn.layer.cornerRadius = 30
```

Figure 4. UIKit Button (Hudson 2019)

```
Button(action: buyShip ) {
  Text("Buy")
}.padding()
 .background(Color(red:0.47, green:0.32, blue:0.7, opacity:1))
 .corderRadius(25)
 .foregroundColor(.white)
 .font(.headline)
```

Figure 5. SwiftUI Button (Hudson 2019)

Figures 4 and 5 are examples of the code required to make the same clickable button, along with the associated styling. Figure 4 is the corresponding UIKit code, while Figure 5 depicts the same using SwiftUI. We can see in this example that SwiftUI is significantly clearer, lacking much of the scaffolding required in UIKit. In practice, this observation generally holds true: SwiftUI is often capable of producing similar results with much less code than UIKit.

Design Choices made for SmartBell App

Given the advantages of MVVM over MVC, we chose to move forward with the MVVM architecture pattern. SwiftUI's increased readability and native support of the

MVVM pattern made it the clear choice for us to move forwards with MVVM.

This not to say that UIKit and MVC do not have their place. MVC is still an incredibly popular architecture pattern. UIKit is a more mature framework compared to SwiftUI, given its decade long head start, which would mean that one could expect more stability and a larger feature set from UIKit as of May 2020.

Conclusion

iOS architecture patterns play an important part in the resulting quality of the code. Both Model-View-Controller and Model-View-ViewModel patterns are incredibly popular and make of the core design philosophies of the UIKit and SwiftUI frameworks. We have seen that the MVVM reduces coupling between components compared to MVC. The ease of maintaining and testing code with MVVM, coupled with the conciseness of SwiftUI were deciding factors for implementing that pattern in the SmartBell application.

References

1. Hudson, P. (2019, June 17). *SwiftUI vs UIKit – Comparison of building the same app in each framework* [Video]. YouTube. https://www.youtube.com/watch?v=qk2y-TiLDZo

2. Mozilla. (2019, March 18). *MVC*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Glossary/MVC

3. Model–view–viewmodel. (2009, June 2). Wikipedia, the free encyclopedia. Retrieved May 1, 2020, from https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel

4. Franchitti, J. (n.d.). *Design Patterns, Architectural Patterns*. http://www.nyu.edu/classes/jcf/g22.2440-001_sp06/slides/session8/g22_2440_001_c82.pdf

5. Apple Developer. (n.d.). *About app development with UIKit | Apple developer documentation*. https://developer.apple.com/documentation/uikit/about_app_development_with_uikit