Learn the Four Swift Patterns I Swear By

Written by Bart Jacobs

Cocoacasts

Learn the Four Swift Patterns I Swear By

Bart Jacobs

This book is for sale at
http://leanpub.com/learn-the-four-swift-patterns-i-swear-by

This version was published on 2017-11-19

Contents

About Cocoacasts

My name is Bart Jacobs[1] and I run a mobile development company, Code Foundry[2]. I've been programming for more than fifteen years, focusing on Cocoa development soon after the introduction of the iPhone in 2007.

Over the years, I've taught thousands of people about Swift and Cocoa development. Through my experience teaching, I've discovered and learned about the main problems people struggle with.

[1] https://twitter.com/_bartjacobs
[2] https://codefoundry.be

I created Cocoacasts to offer a roadmap for anyone interested in learning Swift and Cocoa development. Through Cocoacasts, I provide a clear path to learn the tools, the language, and the frameworks you need to master Swift and Cocoa development.

I currently work as a freelance developer and teach people about Swift and Cocoa development. While I primarily focus on developing software for Apple's platforms, I consider myself a full stack developer with a love and interest for Swift and Ruby development.

You can find me on Twitter[3]. Follow me and say hi. You can also follow Cocoacasts[4] on Twitter if you're interested in what I teach on Cocoacasts.

[3] https://twitter.com/_bartjacobs
[4] https://twitter.com/_cocoacasts

Welcome

Swift is still very, very young and many developers are still figuring out how to best use the language. There are countless tutorials about patterns and best practices, which makes it hard to see the forest for the trees.

In this book, you learn the four patterns I use in every Swift project I work on. You learn how easy it is to integrate these patterns in any Swift project. I promise that they are easy to understand and implement.

1 Namespaces in Swift

The first Swift pattern I use in every project that has any complexity to it is namespacing with enums and structs. Swift modules make the need for type prefixes obsolete. In Objective-C, it's a best practice to use a type prefix to avoid naming collisions with other libraries and frameworks, including Apple's.

Even though modules are an important step forward, they're not as flexible as many of us would want them to be. Swift currently doesn't offer a solution to namespace types and constants within modules.

2 Dependency Injection in Swift

Dependency injection is a bit more daunting. Or that's what you're made to believe. Does dependency injection sound too complex or too fancy for your needs. The truth is that dependency injection is a fundamental pattern that's very easy to adopt.

My favorite quote about dependency injection is a quote by James Shore. It summarizes much of the confusion that surrounds dependency injection.

> Dependency Injection is a 25-dollar term for a 5-cent concept.
> â€” James Shore

When I first heard about dependency injection, I also figured it was a technique too advanced for my needs at that time. I could do without dependency injection, whatever it was.

3 Value Types and Reference Types

When talking about object-oriented programming, most of us intuitively think about classes. In Swift, however, things are a bit different. While you can continue to use classes, Swift has a few other tricks up its sleeve that can change the way you think about software development.

This is probably the most important mindset shift when working with Swift, especially if you're coming from a more traditional object-oriented programming language such as Ruby, Java, or Objective-C.

4 Model-View-ViewModel in Swift

Model-View-Controller, or MVC for short, is a widely used design pattern for architecting software applications. Cocoa applications are centered around MVC and many of Apple's frameworks are impregnated by the pattern.

But there's an alternative that resolves many of the issues MVC suffers from, **Model-View-ViewModel**. You've probably heard of MVVM. But why is it becoming increasingly popular in the Swift community? And why have other languages and frameworks embraced Model-View-ViewModel for years?

I hope that my book helps you in some way, big or small. If it does, then let me know. I'd love to hear from you.

Enjoy,

Bart

1 Namespaces

The first Swift pattern I use in every project that has any complexity to it is **namespacing** with enums and structs. Swift modules make the need for class prefixes obsolete. In Objective-C, it's a best practice to use a class prefix to avoid naming collisions with other libraries and frameworks, including Apple's.

Even though modules are an important step forward, they are not as flexible as many of us would want them to be. Swift currently doesn't offer a solution to namespace types and constants within modules.

A very common problem I run into when working with Swift is defining constants in such a way that they are easy to understand by anyone working on the project. In Objective-C, this would look something like this.

```
1 NSString * const CCAPIBaseURL = @"https://example.com/v1";
2 NSString * const CCAPIToken = @"sdfiug8186qf68qsdf18389qsh4niuy1";
```

This works fine, but it isn't pretty and easy to read. Even though Swift doesn't support namespaces within modules, there are several viable solutions to this problem.

Using Structures

The option I commonly adopt in Swift projects uses structures to create namespaces. The solution looks something like this.

```
struct API {

    static let BaseURL = "https://example.com/v1/"
    static let Token = "sdfiug8186qf68qsdf18389qsh4niuy1"

}
```

I believe this neat trick was first coined by Jesse Squires[5]. It works great and I love the syntax to access the constants.

```
import Foundation

struct API {

    static let BaseURL = "https://example.com/v1/"
    static let Token = "sdfiug8186qf68qsdf18389qsh4niuy1"

}

if let url = URL(string: API.BaseURL) {
    print(url)
}
```

There is one unwanted side effect. The API structure in the previous examples can be instantiated. While this isn't a problem in itself, it may lead to confusion among fellow developers.

You could make the initializer of the API structure private, making it inaccessible to other parts of the project. Note that this only works if the API structure lives in its own file. That is the result of how access control works in Swift.

[5] https://www.jessesquires.com/blog/swift-namespaced-constants/

```
1  struct API {
2
3      private init() {}
4
5      static let BaseURL = "https://example.com/v1/"
6      static let Token = "sdfiug8186qf68qsdf18389qsh4niuy1"
7
8  }
9
10 let _ = API()    'API' initializer is inaccessible due to 'private' protection level
```

Using Enumerations

Another solution was proposed by Joseph Lord[6] in a discussion on Twitter[7] and written about by Natasha Murashev[8]. Instead of using structures, Joseph proposes to use enums without cases. The result is that the enumeration, acting as a namespace, can't be instantiated.

```
1  enum API {
2
3      static let BaseURL = "https://example.com/v1/"
4      static let Token = "sdfiug8186qf68qsdf18389qsh4niuy1"
5
6  }
```

Using this solution is almost identical to using structures. The main difference is the advantage highlighted by Joseph Lord, you cannot create an instance of the API enum without having an error thrown at you.

[6] https://twitter.com/jl_hfl
[7] https://twitter.com/NatashaTheRobot/status/714798388345110528
[8] https://www.natashatherobot.com/swift-enum-no-cases/

```
enum API {

    static let BaseURL = "https://example.com/v1/"
    static let Token = "sdfiug8186qf68qsdf18389qsh4niuy1"

}

let _ = API() 'API' cannot be constructed because it has no accessible initial...
```

The above solutions, using structs or enums, work very well for me. You don't have to put the constants of your project in one struct or enum. Take a look at the following more advanced example. You have to admit that his look very appealing, especially when compared to Objective-C.

```
import Foundation

enum API {

    static let BaseURL = "https://example.com/v1/"
    static let Token = "sdfiug8186qf68qsdf18389qsh4niuy1"

}

extension UserDefaults {

    enum Keys {

        static let CurrentVersion = "currentVersion"
        static let DarkModeEnabled = "darkModeEnabled"

    }

}
```

We create an extension for the UserDefaults class and define a nested enum, Keys. The resulting API is easy to read and clearly describes the meaning of the CurrentVersion constant static property.

```
// Update User Defaults
let userDefaults = UserDefaults.standard
userDefaults.set(1.0, forKey: UserDefaults.Keys.CurrentVersion)
```

Conclusion

This pattern may look a bit odd at first, but it works very well. Not only does it make working with constants easier, it's a simple pattern that's very easy to pick up.

2 Dependency Injection

Many developers cringe when they hear the words dependency injection. It's a difficult pattern and it's not meant for beginners. That's what you are made to believe. The truth is that dependency injection is a fundamental pattern that is easy to adopt.

My favorite quote about dependency injection is a quote by James Shore[9]. It summarizes much of the confusion that surrounds dependency injection.

> Dependency Injection is a 25-dollar term for a 5-cent concept.
> - James Shore[10]

When I first heard about dependency injection, I also figured it was a technique too advanced for my needs at the time. I could do without dependency injection, whatever it was.

What Is Dependency Injection

I later learned that, if reduced to its bare essentials, dependency injection is a simple concept. James Shore offers a succinct and straightforward definition of dependency injection.

> Dependency injection means giving an object its instance variables. Really. That's it. - James Shore[11]

[9] http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html
[10] http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html
[11] http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html

For developers new to dependency injection, it's important to learn the basics before relying on a framework or a library. Start simple. Chances are that you already use dependency injection without realizing it.

Dependency injection is nothing more than injecting dependencies into an object instead of tasking the object with the responsibility of creating its dependencies. Or, as James Shore puts it, you give an object its instance variables instead of creating them in the object. Let me show you what that means with an example.

An Example

In this example, we define a `UIViewController` subclass that declares a property, `requestManager`, of type `RequestManager?`.

```
1 import UIKit
2
3 class ViewController: UIViewController {
4
5     var requestManager: RequestManager?
6
7 }
```

We can set the value of the `requestManager` property one of two ways.

Without Dependency Injection

The first option is to task the `ViewController` class with the instantiation of the `RequestManager` instance. We can make the property lazy or initialize the request manager in the view controller's initializer. That's not the point, though. The point is that the view controller is in charge of creating the `RequestManager` instance.

```
import UIKit

class ViewController: UIViewController {

    lazy var requestManager: RequestManager? = RequestManager()

}
```

This means that the `ViewController` class not only knows about the behavior of the `RequestManager` class. It also knows about its instantiation. That's a subtle but important detail.

With Dependency Injection

But there's another option. We can inject the `RequestManager` instance into the `ViewController` instance. Even though the end result may appear identical, it definitely isn't. By injecting the request manager, the view controller doesn't know how to instantiate the request manager.

```
import UIKit

class ViewController: UIViewController {

    var requestManager: RequestManager?

}

// Initialize View Controller
let viewController = ViewController()

// Configure View Controller
viewController.requestManager = RequestManager()
```

Many developers immediately discard this option because it's cumbersome and unnecessarily complex. But if you consider the benefits, dependency injection becomes quite appealing.

Another Example

I'd like to show you another example to emphasize the point I made earlier. Take a look at the following example.

```
1  import Foundation
2
3  protocol Serializer {
4
5      func serialize(data: Any) -> Data?
6
7  }
8
9  class RequestSerializer: Serializer {
10
11     func serialize(data: Any) -> Data? {
12         return nil
13     }
14
15 }
16
17 class DataManager {
18
19     var serializer: Serializer? = RequestSerializer()
20
21 }
```

The `DataManager` class has a property, `serializer`, of type `Serializer?`. In this example, `Serializer` is a protocol. The `DataManager` class is in charge of instantiating an instance of a type that conforms to the `Serializer` protocol, the `RequestSerializer` class in this example.

Should the `DataManager` class know how to instantiate an object of type `Serializer`? Take a look at the next example. It shows you the power of protocols and dependency injection.

```
import Foundation

protocol Serializer {

    func serialize(data: Any) -> Data?

}

class RequestSerializer: Serializer {

    func serialize(data: Any) -> Data? {
        return nil
    }

}

class DataManager {

    var serializer: Serializer?

}

// Initialize Data Manager
let dataManager = DataManager()

// Configure Data Manager
dataManager.serializer = RequestSerializer()
```

The `DataManager` class is no longer in charge of instantiating the `RequestSerializer` class. It no longer assigns a value to its `serializer` property. In fact, we can replace `RequestSerializer` with another type as long as it conforms to the `Serializer` protocol. The `DataManager` no longer knows or cares about these details.

What Do You Gain

I hope that the examples I showed you have at least captured your attention. Let me list a few additional benefits of dependency injection.

Transparency

By injecting the dependencies of an object, the responsibilities and requirements of a class or structure become more clear and more transparent. By injecting a request manager into a view controller, we understand that the view controller depends on the request manager and we can assume that the view controller is responsible for request managing and/or handling.

Testing

Unit testing is so much easier with dependency injection. Dependency injection makes it very easy to replace an object's dependencies with mock objects, making unit tests easier to set up and isolate behavior.

In the next example, we define a class, `MockSerializer`. Because it conforms to the `Serializer` protocol, we can assign it to the data manager's `serializer` property.

```
import Foundation

protocol Serializer {

    func serialize(data: Any) -> Data?

}

class MockSerializer: Serializer {

    func serialize(data: Any) -> Data? {
        return nil
    }

}
```

```
// Initialize Data Manager
let dataManager = DataManager()

// Configure Data Manager
dataManager.serializer = MockSerializer()
```

Separation of Concerns

As I mentioned and illustrated earlier, another subtle benefit of dependency injection is a stricter separation of concerns. The `DataManager` class in the previous example isn't responsible for instantiating the `RequestSerializer` instance. It doesn't need to know how to do this.

Even though the `DataManager` class is concerned with the behavior of its serializer, it isn't, and shouldn't be, concerned with its instantiation. What if the `RequestManager` of the first example also has a number of dependencies. Should the `ViewController` instance be aware of those dependencies too? This can become very messy very quickly.

Coupling

The example with the `DataManager` class illustrated how the use of protocols and dependency injection can reduce coupling in a project. Protocols are incredibly useful and versatile in Swift. This is one scenario in which protocols really shine.

Types

Most developers consider three forms or types of dependency injection:

- initializer injection
- property injection
- method injection

These types shouldn't be considered equal, though. Let me list the pros and cons of each type.

Initializer Injection

I personally prefer to pass dependencies during the initialization phase of an object because this has several key benefits. The most important benefit is that dependencies passed in during initialization can be made immutable. This is very easy to do in Swift by declaring the properties for the dependencies as constants. Take a look at this example.

```
class DataManager {

    private let serializer: Serializer

    init(serializer: Serializer) {
        self.serializer = serializer
    }

}

// Initialize Request Serializer
let serializer = RequestSerializer()

// Initialize Data Manager
let dataManager = DataManager(serializer: serializer)
```

The only way to set the `serializer` property is by passing it as an argument during initialization. The `init(serializer:)` method is the designated initializer and guarantees that the `DataManager` instance is correctly configured. Another benefit is that the `serializer` property cannot be mutated.

Because we are required to pass the serializer as an argument during initialization, the designated initializer clearly shows what the dependencies of the `DataManager` class are.

Property Injection

Dependencies can also be injected by declaring an internal or public property on the class or structure that requires the dependency. This may seem convenient, but it adds a loophole in that the dependency

can be modified or replaced. In other words, the dependency isn't immutable.

```
import UIKit

class ViewController: UIViewController {

    var requestManager: RequestManager?

}

// Initialize View Controller
let viewController = ViewController()

// Configure View Controller
viewController.requestManager = RequestManager()
```

Property injection is sometimes the only option you have. If you use storyboards, for example, you cannot implement a custom initializer and use initializer injection. Property injection is then your next best option.

Method Injection

Dependencies can also be injected whenever they're needed. This is easy to do by defining a method that accepts the dependency as a parameter. In this example, the serializer isn't a property on the `DataManager` class. Instead, the serializer is injected as an argument of the `serializeRequest(_:with:)` method.

```
class DataManager {

    func serializeRequest(request: Request,
                          with serializer: Serializer) -> Data? {
        return nil
    }

}
```

Even though the `DataManager` class loses some control over the dependency, the serializer, this type of dependency injection introduces flexi-

bility. Depending on the use case, we can choose what type of serializer to pass into `serializeRequest(_:with:)`.

Which Type to Choose

It's important to emphasize that each type of dependency injection has its use cases. While initializer injection is a great option in many scenarios, that doesn't make it best or *preferred* type. Consider the use case and then decide which type of dependency injection is the best fit.

A Word About Singletons

Dependency injection is a pattern that can be used to eliminate the need for singletons in a project. I'm not a fan of the singleton pattern and I avoid it whenever possible. Even though I don't consider the singleton pattern an anti-pattern, I believe they should be used very, very sparingly. The singleton pattern increases coupling whereas dependency injection reduces coupling.

Too often, developers use the singleton pattern because it's an easy solution to a, often trivial, problem. Dependency injection, however, adds clarity to a project. By injecting dependencies during the initialization of an object, it becomes clear what dependencies the target class or structure has and it also reveals some of the object's responsibilities.

Dependency injection is one of my favorite patterns because it helps me stay on top of complex projects. This pattern has so many benefits. The only drawback I can think of is the need for a few more lines of code.

3 Value Types and Reference Types

When talking about object-oriented programming, most of us intuitively think about classes. In Swift, however, things are a bit different. While you can continue to use classes, Swift has a few other tricks up its sleeve that can change the way you think about software development. This is probably the most important shift in mindset when working with Swift, especially if you're coming from a more traditional object-oriented programming language like Ruby or Objective-C.

Value Types & Reference Types

The concept we tackle in this chapter is the differences between **value types** and **reference types** or, put differently, passing by value and passing by reference.

In Swift, instances of classes are **passed by reference**. This is similar to how classes are implemented in Ruby and Objective-C. It implies that an instance of a class can have several owners that share a copy.

Instances of structures and enumerations are **passed by value**. Every instance of a struct or enum has its own unique copy of data. And the same applies to tuples.

In the remainder of this chapter, I refer to instances of classes as objects and instances of structs and enums as values. This avoids unnecessary complexity.

An Example

It's important that you understand the above concept so let me explain this with an example.

```
class Employee {

    var name = ""

}

var employee1 = Employee()
employee1.name = "Tom"

var employee2 = employee1
employee2.name = "Fred"

print(employee1.name) // Fred
print(employee2.name) // Fred
```

```
Employee
Employee

Employee
Employee

"Fred\n"
"Fred\n"
```

We declare a class, `Employee`, and create an instance, `employee1`. We set the `name` property of `employee1` to `Tom`. We then declare another variable, `employee2`, and assign `employee1` to it. We set the `name` property of `employee2` to `Fred` and print the names of both `Employee` instances.

Because instances of classes are passed by reference, the value of the `name` property of both instances is equal to `Fred`. Does that make sense? Or does this result surprise you?

```
struct Employee {

    var name = ""

}

var employee1 = Employee()
employee1.name = "Tom"

var employee2 = employee1
employee2.name = "Fred"

print(employee1.name) // Tom
print(employee2.name) // Fred
```

```
Employee
Employee

Employee
Employee

"Tom\n"
"Fred\n"
```

In the second example, we declare a structure, `Employee`, and repeat the steps of the first example. The results of the print statements is different,

though.

By replacing `class` with `struct`, the outcome of the example changes in a significant way. An instance of a class, an object, can have multiple owners. An instance of a struct has one owner. When an instance of a struct is assigned or passed to a function, its value is copied. A unique instance of the struct is passed instead of a reference to the instance of the struct.

The moment `employee1` is assigned to the `employee2` variable, a copy of `employee1` is made and assigned to `employee2`. The values of `employee1` and `employee2` have no relation to one another apart from the fact that they are copies.

Benefits of Value Types

Swift uses value types extensively. Strings, arrays, dictionaries, and numbers are value types in Swift. That's not a coincidence. If you understand the benefits of value types, you automatically understand why these types are defined as value types in the Swift standard library. What are some of the benefits of value types?

Storing Immutable Data

Value types are great for storing data. And they're even better suited for storing immutable data. Assume you have a struct that stores the balance of the user's bank account. If you pass the balance to a function, that function doesn't expect the balance to change while it's using it. By passing the balance as a value type, it receives a unique copy of the balance, regardless of where it came from.

Knowing that a value won't change is a powerful concept in software development. It's a bold promise and, if used correctly, one that value types can keep.

Manipulating Data

Value types usually don't manipulate the data they store. While it's fine and useful to provide an interface for performing computations on the data, it's the owner of the value that manipulates the data stored by the value.

This results in a control flow that is transparent and predictable. Value types are easy to work with and, another great benefit, they behave great in multithreaded environments.

Why is that? If you fetch data from a remote API on a background thread and pass the data, as a value, from the background thread to the main thread, a copy is made and sent off to the main thread. Modifying the original value on the background thread won't affect the value that's being used on the main thread. Clean, simple, and transparent.

When to Use Value Types

While I hope I've convinced you to start using value types more often, they're not a good fit for every scenario. Value types are great for storing data. If you pass around the balance of someone's account in your application, you should not be using a class instance. What you're doing is passing around a copy of the current balance of the account.

But if you need to pass around the account itself, then you want to make sure the objects that have a reference to the account are working with the same account, the same instance. If the name of the account holder changes, you want the other objects that have a reference to the account to know about that change.

Value types and reference types each have their value and use. It would be unwise to choose one over the other. Andy Matuschak[12] has a clear stance on the benefits of value types and reference types in software development and has given several presentations on the topic. Andy

[12] https://twitter.com/andy_matuschak

describes the objects of an application as a layer that operate on the value layer. That's a great way to put it.

> Think of objects as a thin, imperative layer above the predictable, pure value layer. â€" Andy Matuschak[13]

[13] https://twitter.com/andy_matuschak

4 Model-View-ViewModel

Model-View-Controller, or **MVC** for short, is a widely used design pattern for architecting software applications. Cocoa applications are centered around MVC and many of Apple's frameworks are impregnated by the pattern.

A few months ago, I was working on the next major release of Samsara, a meditation application I've been developing for the past few years. The settings view is an important aspect of the application.

From the perspective of the user, the settings view is nothing more than a collection of controls, labels, and buttons. Under the hood, however, lives a fat view controller. This view controller is responsible for managing the content of the table view and the data that is fed to the table view.

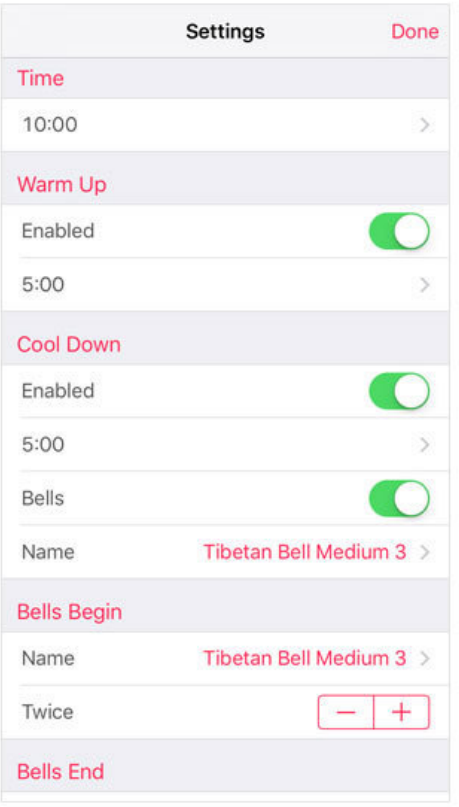


Table views are flexible and cleverly designed. A table view asks its data source for the data it needs to present and it delegates user interaction to its delegate. That makes them incredibly reusable. Unfortunately, the more the table view gains in complexity, the more unwieldy the data source becomes.

Table views are a fine example of the MVC pattern in action. The model layer hands the data source (mostly a view controller) the data the view layer (the table view) needs to display. But table views also illustrate how the MVC pattern can fall short. Before we take a closer look at the problem, I'd like to take a brief look at the MVC pattern. What is it? And what makes it so popular?

What Is It

The MVC pattern breaks an application up into three components or layers, **model**, **view**, and **controller**.

Model

The model layer is responsible for the business logic of the application. It manages the application state. This also includes reading and writing data, persisting application state, and it may even include tasks related to data management, such as networking and data validation.

View

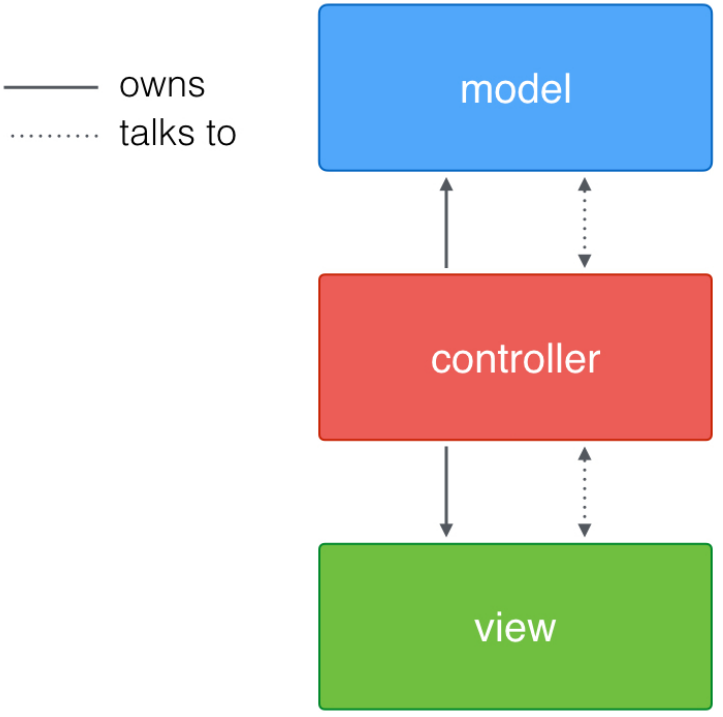
The view layer has two important tasks, presenting data to the user and handling user interaction.

A core principle of the MVC pattern is the view layer's ignorance with respect to the model layer. Views are dumb objects. They only know how to present data to the user. They don't know or understand what they're presenting.

Controller

The view and model layer are glued together by one or more controllers. In iOS applications, for example, that glue is a view controller, an instance of the `UIViewController` class or a subclass thereof.

A controller knows about the view layer as well as the model layer. This often results in tight coupling, making controllers the least reusable components of an MVC application. The view and model layer don't know about the controller. The controller owns the views and the models it interacts with.



Advantages

Separation of Concerns

The advantage of the MVC pattern is a clear separation of concerns. Each layer of the MVC pattern is responsible for a clearly defined aspect of the application. In most applications, there's no confusion about what belongs in the view layer and what belongs the model layer.

What goes into controllers is often less clear. The result is that controllers are frequently used for everything that doesn't clearly belong in the view layer or the model layer.

Reusability

While controllers are often not reusable, view and model objects are easy to reuse. If the MVC pattern is correctly implemented, the view layer and model layers should be composed of reusable components.

Problems

If you've spent any amount of time reading books or tutorials about iOS or macOS development, then you've probably come across people complaining about the MVC pattern. Why is that? What's wrong with the MVC pattern?

A clear separation of concerns is great. It makes your life as a developer easier. Projects are easier to architect and structure. But that is only part of the story. A lot of the code you write doesn't belong in the view or model layer. No problem. Dump it in the controller. Problem solved. Right? Not really.

Data formatting is a common task. Imagine that you're developing an invoicing application. Each invoice has a creation date. Depending on the locale of the user, the date of an invoice needs to be formatted differently.

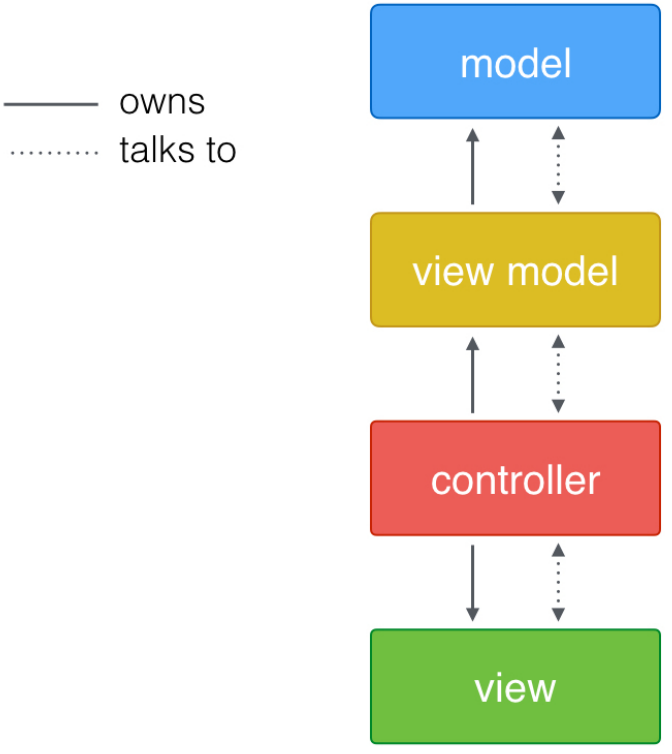
The creation date of an invoice is stored in the model layer and the view displays the formatted date. That's obvious. But who's responsible for formatting the date? The model? Maybe. The view? Remember that the view shouldn't need to understand what it's presenting to the user. But why should the model be responsible for a task that's related to the user interface?

Wait a minute. What about our good old controller? Sure. Dump it in the controller. After thousands of lines of code, you end up with a bunch of overweight controllers, ready to burst and impossible to test. Isn't MVC the best thing ever?

How Can We Solve This

In recent years, another pattern has been gaining traction in the Cocoa community. It's commonly referred to as the **Model-View- ViewModel** pattern, **MVVM** for short. The origins of the MVVM pattern lead back to Microsoft and it continues to be used in modern Windows development.

The MVVM pattern introduces a fourth component, the **view model**. The view model is responsible for managing the model and funneling the model's data to the view via the controller. This is what that looks like.



Despite its name, the MVVM pattern includes four major components, **model**, **view**, **view model**, and **controller**.

The implementation of a view model is usually straightforward. All it

does is translate data from the model to values the view(s) can display. The controller is no longer responsible for this ungrateful task.

Because view models have a close relationship with the models they consume, they're considered more model than view.

MVVM in Practice

I have to warn you, though. MVVM will change the way you write and think about software development. But that's a good thing. The following tutorials take a closer look at MVVM in practice.

- Swift and Model-View-ViewModel in Practice[14]
- More Swift and Model-View-ViewModel in Practice[15]

In these tutorials, I use Samsara as an example to apply the MVVM pattern. The goal is to refactor the settings view of Samsara using MVVM. With the help of MVVM, I end up with a skinny view controller and several view models.

Along the way, you learn more about the MVVM pattern and how it applies to Cocoa development. You also learn how Swift can help us elegantly apply the MVVM pattern thanks to protocols, protocol extensions, and enumerations.

[14] https://cocoacasts.com/swift-and-model-view-viewmodel-in-practice/
[15] https://cocoacasts.com/more-swift-and-model-view-viewmodel-in-practice/