IF YOU'RE SUBCLASSING, YOU'RE DOING IT WRONG.

July 13, 2015

by Hector Matos

THE HEART OF SWIFT

We can think of Swift through the transitive property of equality:

- At the heart of Swift is Protocol Oriented Programming.
- At the heart of Protocol Oriented Programming is abstraction & simplicity.

- Therefore, at the heart of Swift is abstraction & simplicity.

Now you may be wondering about my title. I'm not suggesting that subclasses don't have value. Especially in the use case of single inheritance, classes and subclasses could certainly be a powerful weapon. If you don't believe me, then I highly suggest reading my friend @cocoawithlove's post here. What I am suggesting though, is that it is possible to overuse inheritance and classes for everyday problems in iOS. We have a natural tendency to use reference types/classes as OO programmers to solve these problems, but I personally believe we should be the opposite and lean towards value types instead. Our need to write programs that are modular, scalable, and reusable is inevitable; Swift can give this to us with it's powerful value types without having to resort to relying heavily on reference types. I believe that not only can this be achieved through protocol-oriented programming, but it can also be achieved through two other types of programming that have abstraction & simplicity at their hearts: value-oriented programming & functional programming.

Now just to be clear, I am in no way an expert on these types of programming. I, like many of you, have been an object-oriented programmer since the MMM days (manual memory management). Being self taught, I've managed to value abstraction and simplicity from the start. Without knowing it, I was also an object-oriented programmer who leaned towards the functional and in many cases, value-oriented & protocol-oriented programming. This is probably why I jumped on the Swift bandwagon since Day One with gusto and vigor. That entire week at WWDC I was just floored with how much the heart of Swift aligned with my own beliefs on how we should all program. In this post, I hope to get you (the object oriented programmer) to open your mind to how you can solve everyday problems in a more Non-OOP way.

THE PROBLEM WITH OOP (AND WHY I'LL JUST HAVE TO SUCK IT UP)

I'll be the first to say it: It's hard to write iOS Applications without OOP. At the heart of Cocoa is OOP. Without OOP, you just can't write an iOS application. I sometimes wish it weren't true. If you think contrarily, feel free to prove me wrong. I need to be proven wrong. Pleeeease, prove me wrong!

Either way, at some point you're just going to have to use objects, deal with reference semantics, and be forced into using classes because Cocoa demands it. Some problems that you get with this are problems we all have come to know and love:

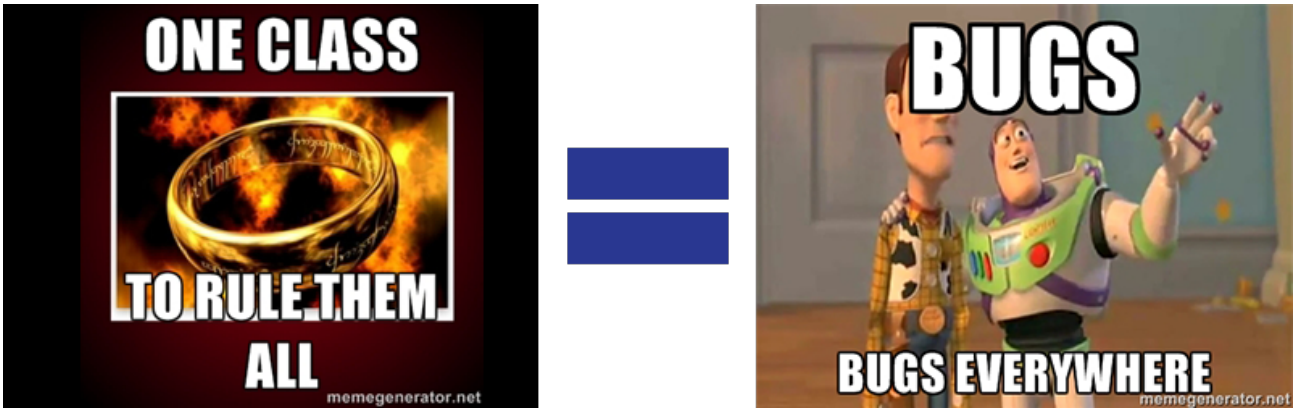
- Passing around class instances always seem to have the uncanny ability of not being in the exact state you expect it to be when you need it to be. (This is caused by mutable state where a shared owner of your object can mutate properties of that object when it feels like it.)
- Subclassing a cool class to get extended functionality prevents you from using other cool, related classes to add more functionality without resorting to multiple inheritance and added complexity. (Take for example, trying to combine two UITextField subclasses to create one super UITextField that has both features.)
- An added issue with the above bulletpoint is unexpected behavior. If you were to have a scenario like that point above, you fall into a dependency problem where one change in a superclass could adversely affect the other superclass you used to chain the two features together. AKA issues due to tight coupling between classes.
- Mocking in unit tests. Some classes are so tightly coupled with environmental state in the system that properly testing some classes require you to create a fake representation of that class. I don't need to tell you that this essentially means that you aren't REALLY testing that class. You're faking it. Not to mention, many Mocking libraries use runtime trickery to fake a class.
- Concurreny issues. This goes hand in hand with the mutable state issue above. There's really no need to tell you that you can get issues from multiple threads mutating a reference at the same time that can throw syncing between objects out of whack at runtime.
- Ease of falling into anti patterns like God classes (classes that hold all of the

responsibility for important high level code that many subclasses need.), Blobs(classes that have WAY too much responsibility), Lava Flow(classes that have so much undocumented code that everyone is too afraid to touch it), etc.

PROTOCOL ORIENTED PROGRAMMING

It's very easy to fall into OOP Anti-Patterns. Half the time, we're all just too lazy (myself included) to click File>New File. We end up just not wanting to build another class from scratch when just adding another function to an already existing class is SO much easier. If you do this enough and follow the same lazy cycle for subclassing one "very important" class, you get the God/Death Star) class.

I actually did this once to make every view controller in an app have the ability to present an error view that pointed to the `navigationController`'s `navigationBar`. Boy, was I a fool. When it finally came time to change the behavior for that Error God Class, I had to change begavior for the ENTIRE app. Not very smart. You should have SEEN the bugs.



Much of that Error God Class could have been easily abstracted and made better through the use of Protocol Oriented Programming. (I HIGHLY recommend watching that video by the way if you want to learn about POP). It's funny when you think about it because in that video, even Apple says,

> *""Instead of using a class, start with a protocol"."*
>
> — Dave Abrahams: Professor of Blowing Your Mind

Here's a small sample of how that atrocity looked:

```swift
class PresentErrorViewController: UIViewController {
    var errorViewIsShowing: Bool = false
    func presentError(_ message: String = "Error!", w
        //do complicated, fragile logic
    }
```

```
}

//Over 100 classes inherited from this class, by the
class EveryViewControllerInApp: PresentErrorViewContr
```

As the project went on, it was quickly apparent that not every UIViewController needed this error logic or every bit of functionality that class provided. Any one of the people on my team could have easily manipulated something in the superclass that would affect the entire application. This made the code fragile. This made it too polymorphic. The superclass here dictated the behavior of its children when the children should be dictating their own behavior. Here's how we can actually structure this better with Protocol-Oriented Programming in Swift 2.0:

swift

```
protocol ErrorPopoverRenderer {
    func presentError(message: String, withArrow shou
}

extension ErrorPopoverRenderer where Self: UIViewCont
    func presentError(message: String, withArrow shou
        //add default implementation of present error
    }
}

class KrakenViewController: UIViewController, ErrorPo
    func methodThatHasAnError() {
        //…
        //Throw error because the Kraken sucks at eat
        presentError(/*blah blah blah much parameters
    }
}
```

You see, we have something pretty cool happening here. Not only did we drop our God class, but we made a move towards more modular and extensible code. By creating an `ErrorPopoverRenderer`, we just gave any class the ability to

render an `ErrorView` as long as they conform to this new fancy protocol! And what's more, our `KrakenViewController` doesn't have to implement the `presentError` function because we extended `UIViewController` to provide a default implementation!

Ah but wait! There's a problem! Now we have to implement every parameter ***every*** time we want to present an `ErrorView`. This kind of sucks because we can't provide default values to protocol function declarations.

I liked those values! And what's worse, in the process of trying to make things more modular, we introduced complexity. Well let's go ahead and try to remedy that a bit with a neat trick Swift 2.0 introduced, Protocol Extensions:

swift

```swift
protocol ErrorPopoverRenderer {
    func presentError()
}

extension ErrorPopoverRenderer where Self: UIViewCont
    func presentError() {
        //Add default implementation here and provide
    }
}

class KrakenViewController: UIViewController, ErrorPo
    func methodThatHasAnError() {
        //…
        //Throw error because the Kraken sucks at eat
        presentError() //Woohoo! No more parameters!
    }
}
```

Ok this is already looking awesome. Not only did we get rid of all those nasty parameters, we also took advantage of Swift 2.0 to provide a default implementation of `presentError` at the protocol level where we made use of `Self`. Using `Self` here indicates that that extension will only ever take place ***if***

and only if the conformer inherits from `UIViewController`. This gives us the ability to assume that the ErrorPopoverRenderer is indeed a `UIViewController` without even extending `UIViewController`! What's even better than that is the fact that now, the Swift runtime calls the `presentError()` through static dispatch instead of through dynamic dispatch. This essentially means we just gave our `presentError()` function a little performance boost at the call site!

Ah, but there is still a problem. This is the end of our Protocol-Oriented Programming journey for now, but we still haven't stopped making this better. In fact, we still have a problem. What happens when we want to take advantage of some default parameter values and leave the rest alone? There isn't much we can do with Protocol-Oriented Programming to help that but there is another world we can go to for help. Now, let's take advantage of Value-Oriented Programming.

VALUE-ORIENTED PROGRAMMING

You see, `Protcol-Oriented Programming` and `Value-Oriented Programming` go hand in hand. In the WWDC video linked above, Crusty made some bold statements that we can do everything classes can do with value types such as structs and enums. For the most part, I agree with this, ***but in moderation***! In my opinion, I agree with Crusty that protocols are essentially the glue that holds value oriented programming together. In fact, since we are already talking about the heart of Swift and value types, I want to show you, my favorite readers, an awesome visual taken from Andy Matuschak's **wonderful** talk on value-oriented programming in Swift:

```
hectormatos at KrakenDev in ~/Downloads/swift_stdlib-master
$ grep -e "^struct" stdlib.swift | wc -l
      87
hectormatos at KrakenDev in ~/Downloads/swift_stdlib-master
$ grep -e "^enum" stdlib.swift | wc -l
       8
hectormatos at KrakenDev in ~/Downloads/swift_stdlib-master
$ grep -e "^class" stdlib.swift | wc -l
       4
```

As you can see, in the Swift Standard Library, there are only ***4*** classes and over ***95*** instances of structs and enums that make up the core of Swift's functionality! Already, we can see the influence of value types in Swift.

As Andy explains, we should think about having a **thin object layer**(classes) and a **thick value layer** when we code in Swift. Classes have their place, but I want to go as far as stating that I believe their place is only at a high level in the object layer where it ***manages*** actions though the pulling of logic that lives in the value layer.

> *"Separate Logic From Action"*
>
> — Andy Matuschak

Value types, as you well know, is a type whose value is copied when it is assigned to a variable or constant, or when it is passed to a function. This reduces complexity by only having ***one owner at any point in time***. This is as opposed to Reference types, who in comparison, are shared on assignment and can have many owners, some of which you may not even know about! At any point in time, using references can introduce side effects from an owner going rogue and mutating it behind the scenes. Classes=more complexity. Values=less complexity.

Using the simplicity of value types, let's use one to achieve the default parameter value implementation that we had before using Brian Gesiak's value options paradigm:

swift

```swift
struct Color {
    let red: Double
    let green: Double
    let blue: Double

    init(red: Double = 0.0, green: Double = 0.0, blue
```

```swift
        self.red = red
        self.green = green
        self.blue = blue
    }
}

struct ErrorOptions {
    let message: String
    let showArrow: Bool
    let backgroundColor: UIColor
    let size: CGSize
    let canDismissByTap: Bool

    init(message: String = "Error!", shouldShowArrow:
        self.message = message
        self.showArrow = shouldShowArrow
        self.backgroundColor = backgroundColor
        self.size = size
        self.canDismissByTap = canDismiss
    }
}
```

Using this options struct (a value type!), we can update our protocol oriented implementation to include some value oriented programming like so:

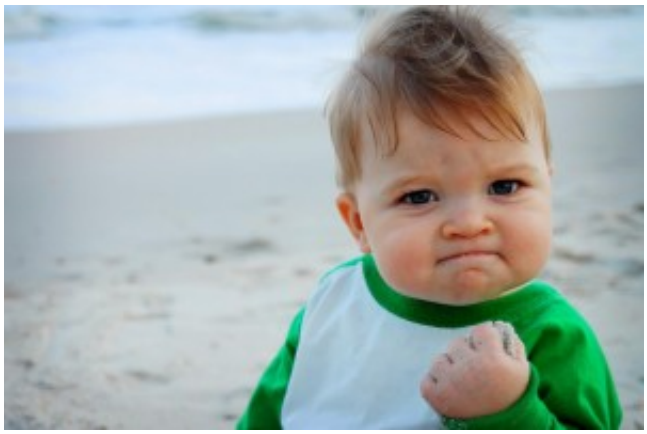
swift

```swift
protocol ErrorPopoverRenderer {
    func presentError(errorOptions: ErrorOptions)
}

extension ErrorPopoverRenderer where Self: UIViewCont
    func presentError(errorOptions = ErrorOptions())
```

```
class KrakenViewController: UIViewController, ErrorPo
    func failedToEatHuman() {
        //…
        //Throw error because the Kraken sucks at eat
        presentError(ErrorOptions(
            message: "Oh noes! I didn't get to eat th
            size: CGSize(width: 1000.0, height: 200.0
        ))
        //Woohoo! We can provide whatever parameters
    }
}
```

As you can see, we've created a completely abstract, scalable, and modular approach to error handling with view controllers without having all view controllers inherit from a God class. This becomes especially beneficial when you have a God class that has several different responsibilities. Not only that, but writing implementations like this error feature enables you to drop it in anywhere you want without much refactoring or structural changes!



FUNCTIONAL PROGRAMMING

So let's get this out of the way. Functional programming is still new to me. But I do know one thing: this paradigm demands an approach to programming that encourages the coder to avoid mutable data and changing state. Like mathematical functions, functional programs are composed of functions whose

output is solely dependent on it's parameter inputs and can't be affected by dependencies outside the scope of the function. This is known as, "data in, data out". This means that every time you pass a value in, you will always get the same value back out. THINK OF THE TESTS!

If we program with the functional in mind, we can take advantage of many benefits from value types coupled with Functional Programming including (but not limited to):

- completely thread safe code (variables are copied on assignment in concurrent code which means another thread can't modify a value in a parallel thread).
- more expressive unit tests.
- no need for mocks in unit tests (using values eliminate the need for recreating a world where you have to mock objects in order to test a bit of functionality. You can essentially recreate whatever you need for a unit test through initialization of a feature completely abstracted from any dependencies.)
- code that is more concise (and frankly, can look like sorcery).
- impressing your coding buddies.
- looking cool.
- getting the Kraken's mad respect.

WHEN TO SUBCLASS

When should you subclass? There are times where you just have no choice. Here are a few examples:

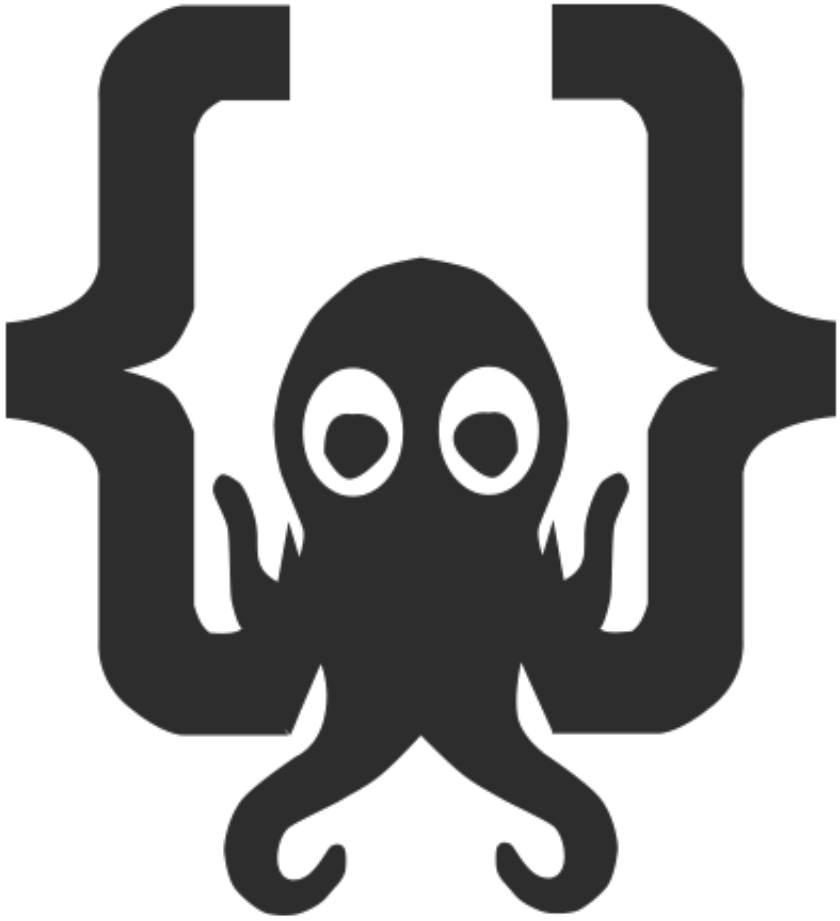
- When it's required by the system. Many Cocoa APIs require the use of classes and you shouldn't have to fight the system just to use value types. UIViewController's have to be subclassed or else you end up with an empty app. DON'T FIGHT THE SYSTEM!
- When you need something to manage and communicate your value types between other class instances. Andy Matuschak gave a great example of

using a class that takes calculated values from a value-type draw system to communicate to a Cocoa class to draw that draw system to a screen.
- When you need or want implicit sharing amongst many owners. One such example is Core Data. Persistence is a fickle thing and when using Core Data, it can be very useful to have synchronization across many owners who require that sync. But beware of concurrency issues! This is a tradeoff you have to make when dealing with these kinds of things.
- When you can't figure out what a copy means for a reference type. Would you copy a singleton? No. Would you copy a UIViewController? No. A window? Hell, no. (Unless you do, then it's your perogative.)
- When an instance's lifetime is tied to external effects or you just need a stable identity. Singleton's are a great example of this.

CONCLUSION

As object-oriented programmers, we have been trained to think of solving problems through classes. Over time, we've even developed patterns to offset issues we encounter with reference types. My thoughts are that we can effectively mitigate these workarounds by using a different form of thinking when we program. If we really do value scalability and reusability, then we have to accept that modular programming is the way to go. It's much easier to achieve this with value types and the new and improved protocols in Swift 2.0. Our previous ways of OOP made it hard to think in terms of value-oriented programming and protocol-oriented programming, but with Swift, these paradigms are starting to become second nature the more and more we write in it. It may take some programming of our own brains to achieve this, but together I believe the iOS community as a whole can accept these practices to greatly simplify how we solve problems on a daily basis. At the heart of Swift is a VERY strong value-type system and frankly, we should take advantage of it by using these practices that have kept value-oriented programming in mind from the start. Hopefully, this post can help you a little bit when it comes to writing more expressive, naturally safe code everyday.

Happy coding, fellow nerds!



7 Comments 20 Likes Share

Name

itstrueimryan • 6 months ago

I referenced your article here: https://stackoverflow.com/q... in a recent answer to a question this article addresses. However, it seems now that you CAN have default parameters for protocol extension default method implementations, so what would you say, now, is the major advantage of this solution vs just added a method to UIViewController itself via extension. Thanks!

Nam-Anh itstrueimryan • 2 months ago

I too would like to know the answer to this, Ryan. I read your StackOverflow article and as a Swift beginner would also like to code in best practices as much as possible. I've followed Hector's code above but am getting compile errors still. Sometimes I find it is hard to understand the underlying reasons why one method is better than another. Hope someone responds.

Alex • 7 months ago

"As you can see, in the Swift Standard Library, there are only 4 classes and over 95 instances of structs and enums that make up the core of Swift's functionality! Already, we can see the influence of value types in Swift."

That's pretty misleading.

For one thing, it's inaccurate. It's an Xcode-generated interface from an old beta release (maybe it was current when you wrote this). It's not up-to-date, or complete (no actual code), and even the grep expressions are wrong (they skip terms like "final class", for example). With a current copy of the Swift stdlib, and a correct grep expression, the struct:class:enum is lower than 4:1:1. There are still a lot of structs but nowhere near 96%.

For another, a ton of core functionality exists in Foundation (and its open-source version, for other platforms), which the Swift stdlib wraps. You're counting types which have a struct façade over a giant class cluster as being 100% struct. Even if you wanted to implement the Swift stdlib from scratch in pure Swift, the most natural way to do it, in many cases, would be to wrap a class. The presence of a struct is not an indictment of a class.

see more

Vikrant Dwivedi • a year ago

Which one is better from memory point of view.....In value type every time we need a new copy but in reference type we don't need to do that ....

Reply • Share ›

Tim ➚ Vikrant Dwivedi • 8 months ago

That's not really true. We've known for years how to efficiently implement persistent data structures, and lots of languages do this -- including, to an extent, Swift.

It'd be more accurate to phrase your question as: When you need to refer to something in memory twice (or more than twice), is it better for the computer to automatically figure out whether it should be shared or not (immutable data structures), or to require the programmer to manually work it out each time (mutable-by-default classes)?

Erlang-based systems are hitting an astonishing nine 9's in production, so I think the value of letting the computer figure it out automatically has been established. You don't translate HLL to opcodes by hand, or allocate registers by hand. Why would you track memory ownership by hand, if you didn't need to?

Reply • Share ›

Олександр Деундяк ➚ Vikrant Dwivedi • 10 months ago

Swift compiler has great optimizations, in most cases objects aren't copied fully if it's not needed. Check wwdc videos about value types and perfomance in swift

1 Reply • Share ›

enti • a year ago

What about your errorViewIsShowing: Bool ? Did you just ignore it? What if you have to have this state for every class?

1 Reply • Share ›

ALSO ON **KRAKENDEV**

Auto Layout Magic: Content Sizing Priorities

32 comments • 2 years ago

Hector Matos — I try my best to do things like this for you all. Calling me out on grammar in public doesn't quite …

Defeating the Anti-Pattern Bully 💥 Singletons

3 comments • 2 years ago

Emma Suzuki — Good article !! Some says: "don't say 'singleton', say 'globalton' instead". I like this saying …

4 Xcode Asset Catalog Secrets You Need to Know

Be Cool with CIFilter Animations 🔥

7 comments • 2 years ago

10 comments • 2 years ago

Pavel — The first and last secret is interesting, I might use it pretty often. Thanks!

Alejandro — For #6 you need to render it into an GLKView by creating a CIContext from the EAGLContext …

tagged with swift, protocol oriented programming, value oriented programming, value semantics, reference semantics, subclass, programming anti patterns, god class, scalable code, iOS, functional programming, inheritance, value type, reference type

Newer / Older

Email Address

DO IT!

I respect your privacy and will NEVER share your data with anyone. You can count on me.