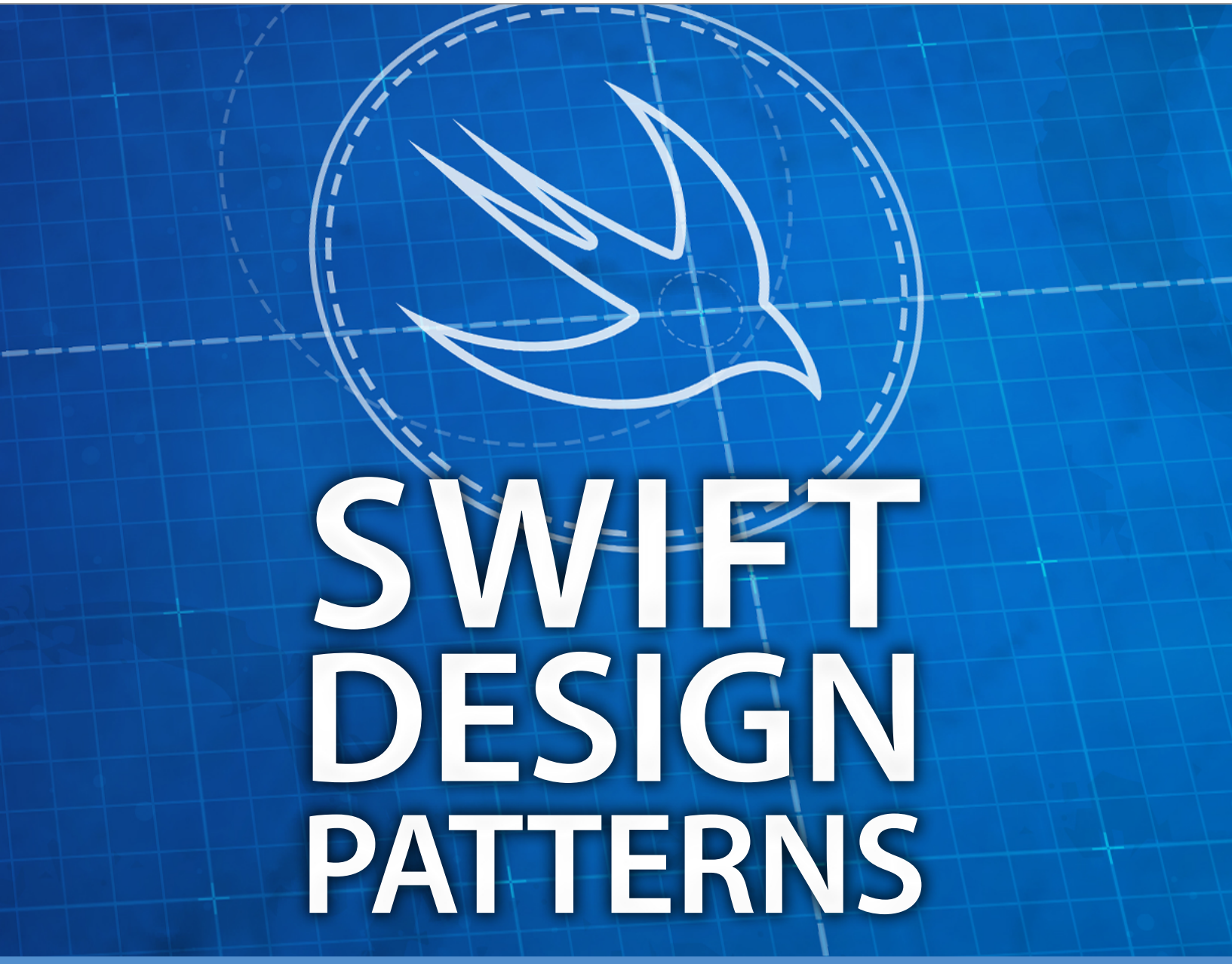
HACKING WITH SWIFT



SWIFT DESIGN PATTERNS

COMPLETE REFERENCE GUIDE

Learn smart, idiomatic ways to design great Swift apps

FREE SAMPLE

Paul Hudson

Chapter 1

MVC

Start with Apple's preferred software architecture.

Model-View-Controller

The Model-View-Controller architecture (MVC) was invented in the 70s by Smalltalk developers. However, because Smalltalk heavily influenced Objective-C at NeXTSTEP, and NeXTSTEP eventually got folded into Apple, MVC became the de facto standard for Apple platform development.

In this chapter I want to introduce you to the concepts behind MVC and how they are applied in Apple development, discuss the major advantages and disadvantages of MVC, then walk you through example code that demonstrates how to clean up common MVC mistakes.

I realize that history and naming can sound dull, but please trust me on this: coming in the world of software architecture with fresh eyes can really confuse you. When acronyms like MVC, MVP, MVVM, MVVMC, and VIPER get tossed around extensively, the distinction between them can start to get lost. So, we're going to start with the basics so you can understand how we got here, then move on.

The fundamentals of MVC

Like I said already, MVC is old, predating both Swift and even Objective-C. That doesn't mean it's necessarily *bad*, in fact quite the opposite: the fact that an architecture invented before iOS, before Windows, and even before MS-DOS is still useful and popular today shows you how good it is. MVC is used extensively on Apple platforms, but it's also hugely popular on Windows, the web, and beyond, which makes extensive knowledge of MVC a transferable asset as your career develops.

You don't need to look far to figure out *why* MVC is so popular, because there are only two reasons.

First, it separates the concerns of your architecture so that each component does only one thing: stores data (Model), displays that data (V), or responds to user and manipulates the model appropriately (C).

Second, you can understand it in seconds. You just read a single sentence above that explains

the core of MVC, which makes it easy to explain, easy to learn, and easy to apply.

These two things combine powerfully: separation of concerns in your code is both explicit and easy to do, and teams can work together on code without treading on each others' toes. As a result, MVC has lasted about 40 years already, and I think it's likely to last another 40 or more.

What my description *doesn't* include is how the three M-V-C components ought to talk to each other. A view is nothing without data, for example, but how should the two communicate?

It won't surprise you to learn that I didn't leave off such information by accident – how these three components communicate can easily descend into a religious war, because MVC doesn't actually tell us that part. Instead it's been left open to interpretation and reinterpretation over the years, to the point where what many people consider MVC is not at all MVC.

For now, let's stick with the classic definition of MVC, as outlined by its author Trygve Reenskaug in 1979.

First, models. Here's how Reenskaug defined them: "Models represent knowledge. A model could be a single object (rather uninteresting), or it could be some structure of objects." This is the easiest component to understand, because it has a one-to-one mapping with real-world concepts like **User**, **Recipe**, and **Book**.

Second, views. Here's how they are defined by Reenskaug: "A view is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a presentation filter. A view is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages." So, views are visual manifestations of the data behind them, and they both read from and write to their model directly.

Finally, controllers. From Reenskaug again: "A controller is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. The controller receives such user output,

translates it into the appropriate messages and pass these messages on to one or more of the views." So, controllers provide the user the ability to control their app, and translate those instructions into messages for the views.

To make things just a teensy bit muddy, Reenskaug outlines a specialized form of controller called the *editor* that "permits the user to modify the information that is presented by the view."

(All quotes taken from Reenskaug, Trygve, December 1979, *Models-Views-Controllers*.)

So models are data, views are representations of that data, and controllers exist to arrange views, shuttle user input back and forth, and edit data. In other words:

- The user uses the controller.
- The controller manipulates the model as an editor.
- The model sends its data to the view.
- The view renders itself to the user.

There might be a few other steps in there – for example your model might apply data validation such as constraining ranges – but that's more or less it. If done well, all three components are neatly decoupled: neither the view and the model have any sort of coupling on the controller, so you can switch them around as needed.

Does that sound like the MVC you use in your application? Almost certainly not. But now that you understand the fundamentals of MVC we can look at how it's employed on Apple platforms.

How Apple developers uses MVC

As soon as you create a new application for iOS, macOS, or tvOS, you're immediately faced with view controllers. The existence of these things – *their very name* – tells you that Apple has been Thinking Different about more than just their marketing.

Think about this for a moment: how often do you create your views for your apps? The

chances are you do it fairly rarely, if ever – we rely on **UIView** on iOS/tvOS and **NSView** on macOS. How often do you make models that send their data straight to whatever view is representing them? Again, the chances are you do it fairly rarely if ever.

Instead, Apple developers tend to write applications a bit like this:

- Models are structs or classes holding a handful of properties.
- Views are almost exclusively taken straight from UIKit or AppKit (WatchKit is quite different).
- Controllers contain everything else.

And that "everything else" is huge – view controllers are commonly responsible for manipulating models and precisely configuring views to display them, handling network code, running animations, acting as data sources, responding to user input, loading and saving, and much more.

As a result, MVC has become retroactively named "massive view controller" because that's often the end result: view controllers that are thousand of lines long.

Now to be fair Apple hasn't done much to correct this problem. While regular view controllers are confused enough, its popular subclasses like **UITableViewController** couple both the view and controller tightly together so that the controller is effectively juggling responsibilities.

Even with its tendency to balloon out of control, MVC the way it's used by Apple really is the standard architecture for iOS apps. This means three things:

1. Apple's sample code, including its Xcode templates, all assume you're using MVC.
2. Most other developers will use MVC. If you join a team and work on their existing iOS app, it's probably based around MVC. If you open a book about Apple development, it's probably based around MVC. If you find some sample code online and want to use it, it probably also assumes you're using MVC.
3. If you don't use MVC, you risk fighting Apple's built-in APIs that are all designed around

MVC.

Sometimes it doesn't matter how appealing alternative architectures are, sometimes MVC is the best choice. This is particularly true for smaller apps, where the work required to set up an alternative is almost as much work as building the app.

Model-View-Presenter

As you've seen, the way we use MVC on Apple platforms is distinctively different from the way MVC was designed. This shouldn't come as a shock: MVC was codified at a time when graphical user interfaces were still young, and using a language that is quite alien to Swift.

A similar-but-different alternative is called Model-View-Presenter (MVP), and looks like this:

- The Model stores data that will be displayed to the user.
- The View stores passive user interface elements that forward user interaction to the presenter so it can act on it.
- The Presenter reads data from the model, formats it, then displays it in views.

Sound familiar? Although it's not an exact match, I think you'll agree that Apple's implementation of MVC is much more similar to *MVP*.

Advantages and Disadvantages

Although MVC is the default choice for most developers, that doesn't mean you should use it uncritically. To help you decide what's right for your project, I've tried to put together some advantages and disadvantages to give you a better idea of what you're working with.

Advantages

The biggest advantage of MVC is that it's a well-known pattern implemented in quite literally millions of programs. Although the definition of MVC will always vary slightly from person to person and platform to platform, the core tenets remain the same. This gives MVC an enviable position as being one of the few software architectures in the world that are recognizable by everyone – if you say "we should use MVC", the other developers in the room will understand the meaning and implications of that. The same can rarely be said for MVVM, MVVMC, VIPER, Elm, or other architectures.

Second, MVC separates responsibilities so clearly that even a beginner can be sure what role something has. Put simply, if it draws to the screen it's a view, if it stores data then it's a model, and if it doesn't draw to the screen or store data then it's a controller. This is a clear separation of concerns that makes it easy for everyone to get started, and lets us subdivide larger applications into digestible chunks – at least at first.

Third, whether or not you use classic MVC or Apple's modified MVP, it has a clear data flow. In Apple development the controller updates the model and responds to state changes, while also updating the view and responding to user events. The controller becomes the ultimate arbiter of truth, with both views and models being thin receptacles.

Finally, MVC is how UIKit, AppKit, and WatchKit are designed. Even though we see weird hydras like "view controller", that's where view-related events such as **viewDidLoad()** and **viewDidAppear()** get sent so that's where they ought be dealt with. View controllers are the inevitable hub of all events in Apple's world, which is why adding code there is so easy.

Disadvantages

The biggest disadvantage of MVC is that our view controllers frequently end up huge, confusing balls of mud. Yes, it's easy to add code to the view controller, but that doesn't mean it's the best place for it. It's not uncommon to find view controllers conforming to three, four, five, or even more different protocols, which means one view controller is forced to act as a data source for a table view, a collection view, a navigation controller, a WebKit navigation delegate, and more.

One of my favorite speakers is Soroush Khanlou, who had this to say about large controllers: "When you call something a Controller, it absolves you of the need to separate your concerns. Nothing is out of scope, since its purpose is to control things. Your code quickly devolves into a procedure, reaching deep into other objects to query their state and manipulate them from afar. Boundless, it begins absorbing responsibilities."

Second, the separation of concerns can sometimes be blurred. Should code to format data before presentation go into the model, the view, or the controller? A particular culprit is the **cellForRowAt** method of table views – whose job should it be to create and prepare cells for display?

Third, view controllers are notoriously hard to test. Even without your application logic in place, testing a view controller means trying to interact with UIKit controls and from experience I can promise you that's an absolute nightmare. Even though you might try your best to keep things separate, any controller code that encapsulates any *knowledge* – anything more than sending a simple value back in a method – will be harder to test when it touches the user interface.

Fourth, MVC often leads to small views and models. The point of MVC is that it divides responsibilities over three independent components, but in practice there's usually a tiny M, an even smaller V, and a huge C – the division is hugely slanted towards the controller, to the point that you'll often see folks doing all their layout in code as part of their view controller's **viewDidLoad()**.

Finally, MVC doesn't adequately answer questions like "where do I put networking?" or "how

do I control flow in my app?" This isn't an MVC problem – the same problem exists in MVVM and others.

Fixing MVC

There is a logical fallacy known as the No True Scotsman fallacy, which is when you make a sweeping generalization, have it disproved by counter-examples, then adjust the generalization so that it excludes the counter-examples.

Wikipedia has an example that I find particular pleasing because my own Scottish father has tried it on me:

- Person A: “No Scotsman puts sugar on his porridge.”
- Person B: “But my uncle Angus likes sugar with his porridge.”
- Person A: “Ah yes, but no *true* Scotsman puts sugar on his porridge.”

(In case you were curious, the correct way to make porridge is with salt. Apparently.)

A version of this same fallacy is often seen when people defend MVC. It goes something like this:

- Person A: “MVC architecture neatly separate models, views, and controllers.”
- Person B: “But all these apps put view code into the controller.”
- Person A: “Ah yes, but that’s not a *true* MVC architecture.”

Those apps might not represent the pure definition of MVC architecture, but they certainly represent how it’s used in practice.

In this chapter I want to look at four ways to improve bad MVC code. My goal is *not* to make you write MVC in the way Trygve Reenskaug defined it 40 years ago, because we’ve moved on since then. Instead, I want to use Apple’s implementation of MVC as a target: what can we do to identify problematic code, then refactor it so it’s cleaner, clearer, and more in tune with what Apple intended?

One view controller to rule them all

The C in MVC stands for “controller”, but that doesn’t mean you need to have precisely one

controller in your app, or even one controller per screen. You can have as many controllers as you like, and they can work together, work independently, or a mix of the two. Dave DeLong, a long-time Apple engineer with extensive experience across multiple frameworks, said this: "the principle behind fixing Massive View Controller is to unlearn a concept that's inadvertently drilled in to new developers' heads: 1 View Controller does not equal 1 screen of content."

I should clarify three things. First, that concept is not inadvertently drilled in to new developers' heads – it's *actively* drilled in there. Second, I've personally taught this understanding to thousands of students, so in some respects I'm part of the problem. However, third is that this understanding is actually useful when you're just getting started – it's a helpful way to explain the concept to users when they are just starting out, because they can look at the Mail app on their phone and see view controllers being pushed on and off a stack.

So, I don't think the old "1 view controller equals 1 screen of content" is a bad thing to get started with, but it *does* become bad as your application grows. Apple has provided us with specific, powerful tools such as view controller containment to help break screens up into more manageable chunks, but these get remarkably low usage – developers continue to be attracted to the Ball of Mud methodology.

This gives us our first target for bad MVC apps: can extremely large view controllers be split into smaller controllers using view controller containment? The code required is almost trivial – adding a child view controller takes only four steps:

1. Call **addChildViewController()** on your parent view controller, passing in your child.
2. Set the child's frame to whatever you need, if you're using frames.
3. Add the child's view to your main view, along with any Auto Layout constraints if you're using them.
4. Call **didMove(toParentViewController:)** on the child, passing in your main view controller.

In Swift code it looks like this:

```
addChildViewController(child)
child.view.frame = frame
view.addSubview(child.view)
child.didMove(toParentViewController: self)
```

When you're finished with it, the steps are conceptually similar but in reverse:

1. Call **willMove(toParentViewController:)**, passing in **nil**.
2. Remove the child view from its parent.
3. Call **removeFromParentViewController()** on the child.

In code, it's just three lines:

```
willMove(toParentViewController: nil)
view.removeFromSuperview()
removeFromParentViewController()
```

Just for convenience you might want to consider adding a small, private extension to **UIViewController** to do these tasks for you – they do need to be run in a precise order, which is easily done incorrectly.

Something like this ought to do it:

```
@nonobjc extension UIViewController {
    func add(_ child: UIViewController, frame: CGRect? = nil) {
        addChildViewController(child)

        if let frame = frame {
            child.view.frame = frame
        }

        view.addSubview(child.view)
        child.didMove(toParentViewController: self)
```

```
    }

    func remove() {
        willMove(toParentViewController: nil)
        view.removeFromSuperview()
        removeFromParentViewController()
    }
}
```

That's marked **@nonobjc** so it won't conflict with any of Apple's own code, now or in the future.

The next step is to divide work across view controllers, and the easiest place to start there is by splitting your controllers up by functionality.

Back when iOS 5 was introduced it came with a feature called Newsstand – a dedicated shelf where users could buy magazine apps. At the time I was running a team of seven at a media company, and we took it on ourselves to launch all 60 magazine brands onto Newsstand ready for iOS 5's arrival – we had less than two months to develop everything from scratch, which was guaranteed to be hectic.

I had done most of the coding to read magazines, others were working on making the Newsstand APIs work and adding a back-issue library, and one person was working on the storefront. When the first draft of the storefront was ready to test, we did the usual "test your app like you hate your app" fuzz testing that all smart developers do, and it was pretty awful – if you tapped a Buy button five times quickly then five purchases would begin, and the app would crash.

Obviously we didn't want users to cause such problems, so I asked the storefront developer to dim the screen as soon as purchasing started, and show an activity indicator on top. This happened in time for the second test run and made things a lot better, but only later did I notice how it was achieved: everywhere the dimming was required a subview was being created with a spinner on top, then removed when the purchase either finished or failed.

This meant two things. First, we had a lot of code duplication on our hands, which is never a good thing. Second, our store's view controller grew huge as it tried to handle all possible success and failure situations in a single place.

This is a situation that's crying out for view controller containment. To be fair to the developer in question, view controller containment was only introduced in iOS 5 so there was no way he would have any experience of it, but developers today have no excuse.

In this situation – any time where you need to overlay one view controller temporarily over another – view controller containment becomes the easiest solution. You can even wrap the whole thing in a closure, like this:

```
guard let vc =
storyboard?.instantiateViewController(withIdentifier: "Second")
else { return }

add(vc)

performSlowWork(with: data) {
    vc.remove()
}
```

In so many cases this literally means you can cut code from one view controller and paste it into another. The grand total amount of code hasn't changed, but your responsibilities are clearer and I guarantee the code is easier to maintain.

You can take this further: two or three child view controllers can work side by side in the same parent view controller, similar to how **UISplitViewController** works on the iPad. If the top half of your controller is only loosely linked to the bottom half, why put them all in one?

(In case you were wondering, we did launch all 60 onto Newsstand in time for the iOS 5 launch, despite Apple coming to visit us and saying "we're so glad you're supporting iPhone, because most others aren't" – iPhone support was apparently promised by an executive. On

launch day over half of all Newsstand apps were from our company, and we won a national award for our efforts.)

Delegation of responsibilities

Why must your view controller be responsible for network requests? Why must your view controller be responsible for loading and saving data? Why must it be the data source for your table view controller and your picker?

The answer of course is that there is no good reason – at least for anything beyond simple apps. Yet it's so common to see code like this:

```
class ViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate, UIPickerViewDataSource,
UIPickerViewDelegate, UITextFieldDelegate,
WKNavigationDelegate, URLSessionDownloadDelegate {
```

What does that class *do*? Apple has already muddied the water enough by giving us "view controllers", but now this poor view controller is being asked to do almost everything by itself. This kind of code *works*, of course it does, but if you're making any given view controller conform to more than one or two protocols then you automatically lose the right to complain about any massive view controllers in that project.

The solution here is to create data source and delegate objects where it makes sense. Usually any protocol with "DataSource" in its name is a good target to be carved off, because you can create a dedicated object that reads your model and responds appropriately.

Yes, this makes your view controller smaller, and yes it helps you get towards the goal of each object having a single responsibility, but the biggest win here is testability. Once you have a dedicated object that consumes model data, adjusts it according to your application logic, then sends the data back as a data source, you can write tests that exclude UIKit – you can create some mock data, hand it to your data source class for processing, then write tests to make sure it's worked correctly.

Let's look at a practical example: you want to embed a **WKWebView** that enables access to only a handful of websites that have been deemed safe for kids. In a naïve implementation you would add **WKNavigationDelegate** to your view controller, give it a **childFriendlySites** array as a property, then write a delegate method something like this:

```
func webView(_ webView: WKWebView, decidePolicyFor
navigationAction: WKNavigationAction, decisionHandler:
@escaping (WKNavigationActionPolicy) -> Void) {
    if let host = navigationAction.request.url?.host {
        if childFriendlySites.contains(where: host.contains) {
            decisionHandler(.allow)
            return
        }
    }

    decisionHandler(.cancel)
}
```

(If you haven't used **contains(where:)** before, you should really read my book Pro Swift.)

To reiterate, that approach is perfectly fine when you're building a small app, because either you're just learning and need momentum, or because you're building a prototype and just want to see what works.

However, for any larger apps – particularly those suffering from massive view controllers – you should split this kind of code into its own type:

1. Create a new Swift class called **ChildFriendlyWebDelegate**. This needs to inherit from **NSObject** so it can work with WebKit, and conform to **WKNavigationDelegate**.
2. Add an import for WebKit to the file.
3. Place your **childFriendlySites** property and navigation delegate code in there.
4. Create an instance of **ChildFriendlyWebDelegate** in your view controller, and make it the navigation delegate of your web view.

Here's a simple implementation of just that:

```
import Foundation
import WebKit

class ChildFriendlyWebDelegate: NSObject, WKNavigationDelegate
{
   var childFriendlySites = ["apple.com", "google.com"]

   func webView(_ webView: WKWebView, decidePolicyFor
navigationAction: WKNavigationAction, decisionHandler:
@escaping (WKNavigationActionPolicy) -> Void) {
      if let host = navigationAction.request.url?.host {
         if childFriendlySites.contains(where: host.contains) {
            decisionHandler(.allow)
            return
         }
      }

      decisionHandler(.cancel)
   }
}
```

That solves the same problem, while neatly carving off a discrete chunk from our view controller. But you can – and should – go a step further, like this:

```
func isAllowed(url: URL?) -> Bool {
   guard let host = url?.host else { return false }

   if childFriendlySites.contains(where: host.contains) {
      return true
   }
```

```
    return false
}

func webView(_ webView: WKWebView, decidePolicyFor
navigationAction: WKNavigationAction, decisionHandler:
@escaping (WKNavigationActionPolicy) -> Void) {
    if isAllowed(url: navigationAction.request.url) {
        decisionHandler(.allow)
    } else {
        decisionHandler(.cancel)
    }
}
```

That separates your business logic ("is this website allowed?") from WebKit, which means you can now write tests without trying to mock up a **WKWebView**. I said it previously but it's worth repeating: any controller code that encapsulates any *knowledge* – anything more than sending a simple value back in a method – will be harder to test when it touches the user interface. In this refactored code, all the knowledge is stored in the **isAllowed()** method, so it's easy to test.

This change has introduced another, more subtle but no less important improvement to our app: if you want a child's guardian to enter their passcode to unlock the full web, you can now enable that just by setting **webView.navigationDelegate** to **nil** so that all sites are allowed.

The end result is a simpler view controller, more testable code, and more flexible functionality – why *wouldn't* you carve off functionality like this?

Coding your user interface

A second group who have lost all privileges to complain about massive view controllers are those people who insist on creating their views in code and doing so inside **viewDidLoad()**. I have no problem with people creating UI in code, and in fact it's the smart thing to do for a

great many projects, but just cast your eyes over this monstrosity:

```
backgroundColor = UIColor(white: 0.9, alpha: 1)

let stackView = UIStackView()
stackView.translatesAutoresizingMaskIntoConstraints = false
stackView.spacing = 10
view.addSubview(stackView)

stackView.topAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.topAnchor).isActive = true
stackView.leadingAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.leadingAnchor).isActive = true
stackView.trailingAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.trailingAnchor).isActive = true
stackView.axis = .vertical

let notice = UILabel()
notice.numberOfLines = 0
notice.text = "Your child has attempted to share the following
photo from the camera:"
stackView.addArrangedSubview(notice)

let imageView = UIImageView(image: shareImage)
stackView.addArrangedSubview(imageView)

let prompt = UILabel()
prompt.numberOfLines = 0
prompt.text = "What do you want to do?"
stackView.addArrangedSubview(prompt)

for option in ["Always Allow", "Allow Once", "Deny", "Manage
Settings"] {
```

```
    let button = UIButton(type: .system)
    button.setTitle(option, for: .normal)
    stackView.addArrangedSubview(button)
}
```

That's not even a complex user interface, but it's the kind of thing you'll see in **viewDidLoad()** even though that's a *terrible* place to put it.

All the code above – literally all of it – is *view* code, and needs to be treated as such. It is not controller code, and even with Apple's muddled definition it is not *view controller* code either. It's view code, and belongs in a subclass of **UIView**.

This change is trivial to make: you copy all that code, paste it into a new subclass of **UIView** called **SharePromptView**, then change the class of your view controller's view to your new subclass.

The final **SharePromptView** class should look something like this:

```
class SharePromptView: UIView {
    override init(frame: CGRect) {
        super.init(frame: frame)
        createSubviews()
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        createSubviews()
    }

    func createSubviews() {
        // all the layout code from above
    }
}
```

All **UIView** subclasses must implement **init(coder:)**, but as you're creating your UI in code you will also need to add **init(frame:)**. The **createSubviews()** method is there to support both.

Thanks to that custom **UIView** subclass you can now take a huge amount of code out of your view controller:

```
class ViewController: UIViewController {
    var shareView = SharePromptView()

    override func loadView() {
        view = shareView
    }
}
```

Having a dedicated **shareView** property allows you to access any properties you declare inside **SharePromptView** without having to keep casting **view**.

Avoiding the app delegate

Of all the places that get mightily abused, the **AppDelegate** class is king, queen, chief, and emperor. When you think to yourself, "I need to create this data and share it in all sorts of places," your **AppDelegate** class does not jump up and down shouting "pick me, pick me!"

In fact it's quite the opposite: this class is created to handle launching your app and responding to specific system events, and even if you handle only a few of those you're already talking about 100-200 lines of code.

This class is a particularly bad dumping ground for developers who create their user interface in code. This isn't restricted to developers who write *all* their user interfaces in code – just setting up a tab bar controller can force you to do this.

Here's a simple rule you can try following: if you're putting something into **AppDelegate** that isn't a simple implementation of the **UIApplicationDelegate** protocol, it should almost

certainly be its own controller.

Summary

Massive view controller syndrome is real, but it doesn't *need* to be. I've just outlined four common ways you can make your view controllers smaller, more testable, and closer to the goal of single responsibility:

- View controller containment lets us create simpler view controllers that get composed into one larger one.
- Creating dedicated classes to act as data sources and delegates helps isolate functionality and increase testability.
- Coding your user interface is perfectly fine, but do it in a **UIView** subclass.
- Don't dump code in the app delegate – it's almost never the right place for it.