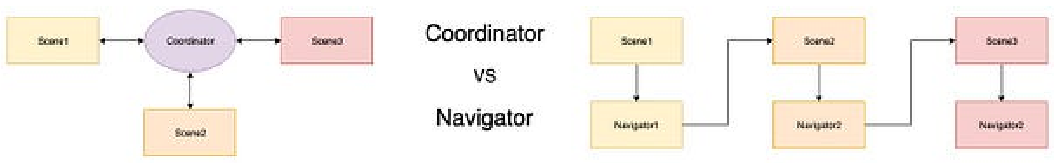
Freedium

< Go to the original



Small Navigators for Scene transitions in iOS

No Segue, no Coordinator, just simple Navigation

Sven Korset

Follow

Better Programming · ~10 min read ·
August 16, 2019 (Updated: December 11, 2021) · Free: No

Pieces of a scalable iOS app architecture

Are you on-trend and use the new coordinator pattern for transitions? Or is your heart still beating for segues and storyboards? Then let me destroy your illusions! 😈

Incidentally, this is a post from the "Pieces of a Scalable iOS App Architecture" series.

Segues

Who remembers the WWDC 2012? At that time, storyboards and segues were a hot topic. Finally, you could see the entire app flow in *InterfaceBuilder,* which screen leads to where, how does the screens look, and no coding needed anymore if you connect a segue transition directly in the storyboard with a button. Visual coding en excellence! 🤩

Segues and storyboards contain some issues that contradict a scalable architecture:

1. If all the screens are in a storyboard, parallel processing leads to very difficult to resolve **merge conflicts.**
2. Even if you use a single storyboard for each screen, it is still very difficult to perform meaningful **code reviews** of these storyboard views.
3. If you then assign colors or other constant values in storyboards, you will need to find all the **hard-coded values** when a single value needs to be changed.
4. And then, if values in the code are overwritten, it can be very time-consuming due to **fragmentation** to find out why a change in the storyboard doesn't work.
5. The storyboard defines a **hard-coded navigation** with segues, which makes the navigation inflexible.
6. Segues usually require identifiers that are **defined redundantly** once in the storyboard and again as a code constant, which is very error-prone.
7. **Dependency injection** and segues do not work very well together.
8. In addition, the **interface isn't very clean**.

You can certainly list a few more drawbacks here, but I just want to take a quick look at the code level, especially to clarify the last points a bit further.

As an example, starting with line 3, a typical *ViewController1* is defined here.

Usually, a segue is initiated via `performSegue` , as shown here in line 12. As an identifier, you have to pass a string which is hopefully, somewhere defined as constants.

Here it's done in line 4 as an example. As the second parameter, you then pass a `sender` of the type `Any` — two parameters and already two flaws.

First, using strings as constants for navigation are not type-safe. Nothing prevents me from passing the string "Foobar" and getting a runtime exception. As you have to define the same string again in the storyboard, this happens more often than you might think. 😨

The second criticism is the `sender: Any?` parameter. If an action is linked to a value, as demonstrated here with a `selectedData` object, you usually want to pass that to the next *ViewController* during the transition. In practice, it usually comes down to the fact that you pass this value here as a `sender` and then cast it back in the `prepare` method in line 21.

It is painful to see `sender` misused as a transfer parameter for the destination *ViewController*. The interface for "sender" actually says something totally different.

> *sender: The object that you want to use to initiate the segue.*

Does `selectedData` really initiate the segue?

And casting to *Any* and back would irk anyone that has even a little sense for "pretty code". 🤢

In line 25, the parameter required by *ViewController2* is assigned by *ViewController1* via property injection. However, this requires *ViewController1* to know the *ViewController2*, which means a direct dependency.

You can easily resolve this direct dependency by using a *ViewController2Interface* protocol, but what's the point if you cannot swap the actual *ViewController2* instance because it's hard-coded by the storyboard? 😑

If you look at *ViewController2*, you can see that this property injection also leads to a force unwrapped property in line 33.

For example, if you forget the assignment in line 25, you will notice the crash at runtime only if the value is also accessed. You can do that a bit earlier by doing a corresponding `precondition` test as in line 38 but it's still not pretty.

By the way, if you ask why you should be using a `precondition`, I highly recommend Paul Hudson's post "Picking the right way of failing in Swift".

However, using an optional as shown in *ViewController3* on line 43 would not be better because an optional in the interface means that

In practice, however, this only leads to workarounds for undefined states because the interface is simply wrong.

Do you see? There are plenty of issues with segues and storyboards, so just don't use them, please! 🙏 (Except if you are hacking a prototype.)

Coordinator

In 2015, Soroush Khanlou made the concept of "The Coordinator" en vogue.

Andrey Panov wrote in detail about the coordinator pattern in his article "Coordinators Essential tutorial". Another interesting blog post and video is "How to use the coordinator pattern in iOS apps" by Paul Hudson.

The central idea is that the *ViewController* is no longer responsible for the transition. It's now the responsibility of a new class, the *Coordinator*. This can be abstracted with a protocol so that there is no direct dependency anymore.

ViewController1 and *ViewController2* are instantiated via the *FirstCoordinator* and thus know nothing about each other. They also know nothing about *FirstCoordinator* because the dependency was abstracted via the *Coordinator* protocol.

yourself how the transition is ultimately implemented. This could even be done via storyboards and segues! 🤣 (But seriously, don't use them! 🤨)

After instantiation, the coordinator is always assigned to the *ViewController*. That way, further transitions are possible.

So, there's nothing wrong with the possibility to navigate to another *ViewController* in line 11. Or that navigation then can happen in *ViewController2*, too.

However, it gets interesting when *ViewController2* then calls `controller2` at the coordinator, just because the interface allows it. 😏

Of course, that sometimes makes sense, for example, in a folder-like content structure but then not for the files/leaves.

And does it really make sense that every other *ViewController* can call `controller2` from anywhere, too?

So, if the interface also provides these transitions for leaves and all the other cases where it doesn't make sense, the interface is simply wrong.

> Don't provide wrong interfaces!

It also might be interesting if different transitions or animations are to apply, e.g., from *ViewController1* to *ViewController3* via push, but from *ViewController2* to *ViewController3* modally.

it?

You might come up with something like `controller3(showModal: Bool)`, but then we are back at the bad interface again. Is it **really** intended that **every** controller including *ViewController3* can show *ViewController3* modally **and** via push?

Now you might ask: Why is that a problem? If it does not make sense to navigate from *ViewController1* to *ViewController3*, just don't call `controller3` in *ViewController1*! 🤔

Yeah, that's the same reason why you should mark everything as *private* and only open it as *public* if necessary. It's a core concept of encapsulation and good API design: **only provide what's really necessary**.

Sure, you can provide all 100 of your framework's methods as *public*, but that will overwhelm new users of your framework when essentially only three of them are really meant to be used.

By hiding the other 97 via *private* and *internal* access modifiers, you don't only make the visible API smaller and clearer, you also reduce the risk of misusing your framework.

The same applies to your navigation interface. If it's not intended to move from *ViewController1* to *ViewController3*, then simply don't provide this possibility.

If the design changes and it makes sense from a point, then just add this possibility but do not open Pandora's box in advance. 😉

provides those transitions that make sense for this *ViewController*. However, it brings us closer to a *Navigator*. 😁

The coordinator pattern seems to be very popular right now. So, who is happy with it should, of course, use it. Nevertheless, I would like to re-introduce an old approach.

Navigator

What is the task of a *Navigator* class? Of course, it's to navigate from one scene to another.

In my article "The perfect iOS app architecture", I wrote about the concept of a *Scene* and the specific role of the *Navigator*. In short: Scene = app screen (including View, ViewController, Navigator and more). In the following example, I use a *Scene* as a *ViewController* for simplicity.

Each scene has its own *Navigator* and tells you which scenes you can navigate to from this scene.

Depending on the environment (iPad/iPhone) or test scenario, the concrete implementation may lead to different *ViewControllers* or with different animations.

In line 3, a *BaseViewController* is first defined, from which all *Scene* classes derive, instead of directly from *UIViewController*. This is to

Lines 35 and 24 define *Scene1* and its associated *Scene1Navigator*. The *Navigator* requires a *UIViewController* to perform the transitions of the scene. This is assigned via property injection in line 43 in the `init` method of *Scene1*.

The navigation is then triggered in line 47 (for simplicity the *NavigatorInterface* is omitted). Normally, this would be done by a *Logic*, not a *ViewController*, but the principle would be the same.

The logic knows that the app should navigate to a new scene here. Any parameters must also be known and passed. However, how the transition actually works doesn't matter for the logic. That's what the *Navigator* takes care of.

How the concrete transition finally looks, only the *Navigator* knows with line 27.

Whether via push or modal or with special animation or even to a completely different *ViewController* for a test scenario, these are concrete implementation details of the current *Navigator* instance.

Of course dependency injection is the way to go, just as implied with *Scene2* in line 54.

The *Navigator* should not be directly dependent on other scenes, so you can use a *Factory* to solve this. How to do this is described in my article "Dependency management done manually in Swift", or you can investigate in my DemoApp Project (DAP). 🧐

Navigator. A simple interface provides all the transitions of the scene. That's all! 😁

The Difference

But what is the big difference to the Coordinator pattern?

1. Each scene has its own *Navigator*. With this, a developer immediately sees all transitions from this scene that are not only possible but also allowed. When looking at a *Coordinator*, the developer still needs to know which of all possible transitions also make sense and thus are allowed. In a *Navigator*, this is obvious. All provided transitions make sense.
2. There is no need for a `start` method. Somewhere, of course, the app must be started, but an *AppDelegate* does not actually perform a transition, it simply sets a `rootViewController`, and that's it. For that, you do not need a *Navigator/Coordinator*.
3. The *Navigator/Coordinator* does not need to be passed to other *ViewControllers*. Of course, other dependencies must be injected, and each scene has to create its own *Navigator*, but the *Coordinator*'s meta-level of "coordinating all *ViewControllers*" is dropped.
4. In some articles, they go more into detail of flows and returned values from flow to flow. At some point, they introduce child coordinators and special nesting that can make coordinators quite complex and difficult to understand. By contrast, the *Navigator* concept is simple, clear and easy to understand.

In essence, a *Coordinator* represents an n:n relation, while the *Navigator* represents a 1:m relation, where $m \leq n$ and n equals the number of *ViewControllers/Scenes* in your app or in a particular flow.

But wait, sometimes I want to have an *n*:1 relation where it should be possible to navigate to *Scene2* from every other scene in the app. With a *Coordinator*, I then only have to write this method once. With *Navigators*, I have to write it again and again or, even worse, use copy and paste. How would you solve that? 🤔

No problem, simply use **protocols and protocol extensions.**

In line 1, a *Scene2Navigation* protocol is defined. This protocol provides the transition to *Scene2* but needs the *ViewController* for performing the transition. So, this is not a Navigat**or**, but a Navigat**ion.** This protocol enables a class to **navigate to scene2**.

With line 6, the protocol is then extended by the implementation of `scene2` . This is the transition code that should be executed for all transitions to *Scene2* regardless from which scene. At least for all scenes that implement this protocol.

In line 15, we then have our *Navigator* again, but in this case, it doesn't provide an own implementation for the *scene2* transition. This is automatically added by extending the *Scene1Navigator* to apply to *Scene2Navigation* in line 19.

navigate from *Scene2* to *Scene2* or from other particular scenes, simply don't extend the corresponding *Navigator* and don't write such a line for that *Navigator*.

Now you have easy control, the interface remains clear & correct and still no duplicated code. 😊

Conclusion

Whether you use *Coordinator* or *Navigator* is up to you. The difference is not that big. I like it simple and clean rather than complicated and massive. Therefore, I prefer the *Navigator* concept, but everyone has their own opinions.

The main thing is that it is not done by the *ViewController* anymore, and whatever the transition looks like, please **do not use segues and storyboards**! 😉

#swift #navigator #coordinator #segue #programming