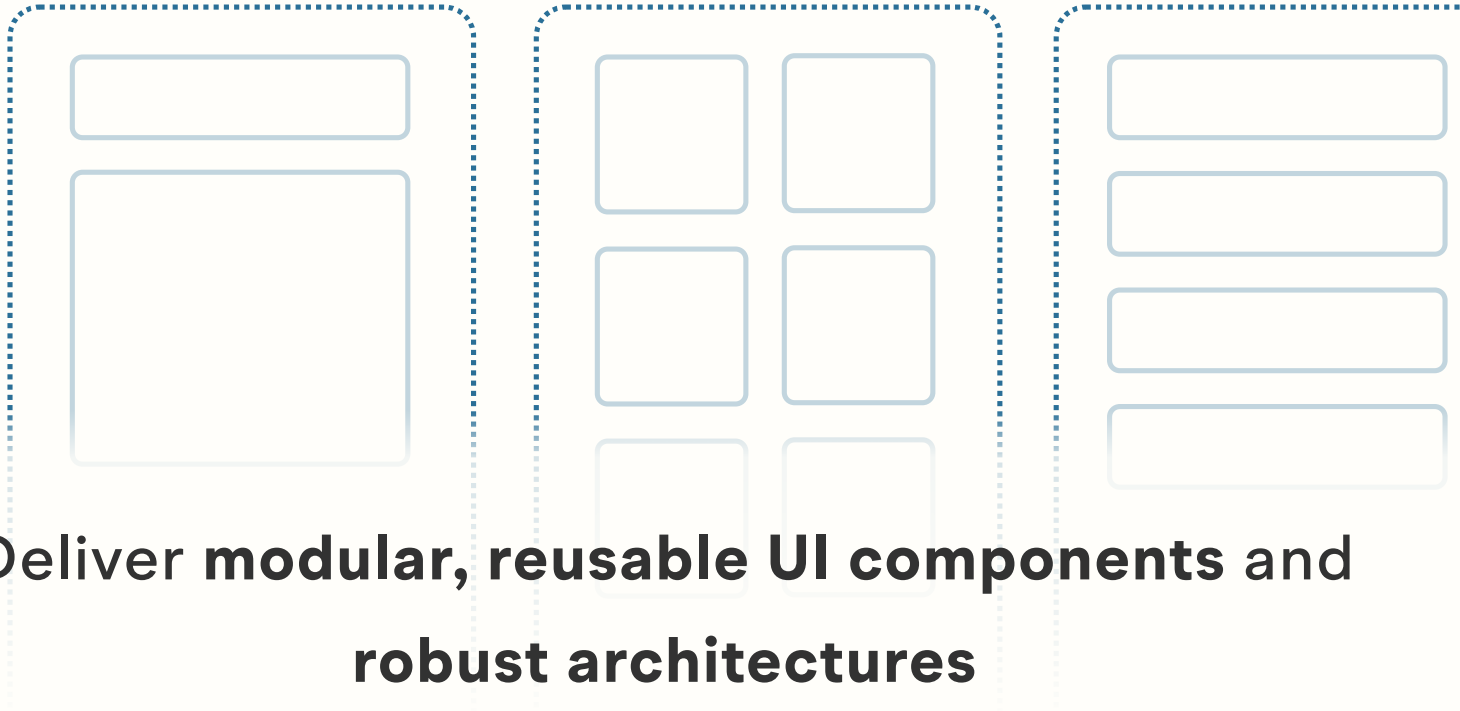
EARLY RELEASE EDITION

MOBILE SYSTEM DESIGN

1. MODULAR UI & UI ARCHITECTURES



Deliver **modular, reusable UI components** and **robust architectures**

TJEERD IN 'T VEEN

9 Reusing Views Across Flows

In this chapter

- Understanding what makes a view reusable across flows
- Common problems that occur in practice
- Why the navigation-idiom you use isn't critical for reusable views
- Decoupling a view from flows by using various implementations:
  - Using closures and actions
  - Using standalone types or interfaces
- Designing a declarative API to offer a configurable view
- How to handle optional elements when reusing views

Views are like meerkats. If you see one, there are more nearby, and they need each other to function.

Once we deliver a view, it's rare they work alone. Often we need to place it in some sort of navigation-flow; A set up where a user goes from screen to screen to finish a task.

Even if a view does work completely in isolation; It still needs to be inserted, pushed, presented, or embedded, which is done by another view, window, scene, or screen.

Therefore, placing views forces us to make choices about navigation. However, a common problem when delivering a screen or view is that it can become tricky to reuse it in different locations.

Now, you might think, *"I don't need my view to be reused in multiple flows"*.

I think that's fair. Views often start out as part of a single flow. They rarely need to be reusable from the start.

Unfortunately, requirements can often change, and so might your navigation-flow. In that case, I'd argue it's good to consider having a view that you can easily move around in flows.

However, what if I told you that, with a few simple ideas, you can spend the *same amount* of effort on a view, *and* you get reusability for free?

Because it's more about naming and roles than fancy architectures to make a view reusable, which we'll explore in this chapter.

We already covered how to make self-sufficient views that we can cut and paste from and to different locations in the app. Continuing with that idea, we'll cover how to set views up in such a way, so that it's easy to place them in various navigation-flows.

NOTE: With this approach, it will even be painless to embed them in other views, which we'll cover in the chapter 'Taming Complex UI'.

This chapter is solely focused on making views reusable across any flow, whether they're pushed on the stack, presented on top of other views, or even embedded inside larger views. This chapter is *not* focused on building a navigation flow itself. We'll save that for a later chapter.

We start this chapter by looking at the common problems we run into that prevent us from reusing a view in different flows.

After understanding the problems on a deeper level, we'll move on to three solutions.

Using the first approach, we'll decouple views from navigation flows using closures, which we'll sometimes call *actions* as a convention for declarative UI. As we'll discover, closures are good to get up and running, but don't always scale. Especially when dealing with nested views, which are more common in declarative frameworks.

After covering the trade-offs of the closure solution, we'll look at a solution when we pass an interface around. However, merely creating an interface isn't always enough. We'll understand why the roles and names of types become crucial to make this solution work properly.

To finish up the chapter, we'll deal with optionality. Because reusing a view across flows means that maybe we want to modify views, such as hiding or changing certain buttons or UI elements. For this, we're going to get fancy and design a declarative API. Not only because it's fun, but because this offers an elegant way to keep adding functionality to a view while we build up its complexity.

Before we implement solutions, let's begin by looking at two common problems that prevent us from reusing views across flows.

9.1 Common problems

We may see or write code inside a view that manages navigation. But certain problems can sneak in.

As an example, a user might press a *close* button on a view. Then, depending on how the parent displays the view, the view may either dismiss itself if it's presented modally (on top of another view). Or it might pop itself from the navigation stack if it's displayed sequentially.

In the next example, we can see how this looks; We have an iOS viewcontroller which represents a screen. Once a user taps a close button, `didPressCloseButton()` is called, in which the view

inspects the view stack to decide whether to dismiss its own presentation, or pop itself off the view stack.

WARNING: This is an anti-pattern!

```
class CustomersLoveHybridViewController: UIViewController {

    func didPressCloseButton() {
        // Inspect whether this screen is presented.
        if self.isBeingPresented || self.presentingViewController != nil {
            // If presented, dismiss the screen, revealing the screen below.
            self.dismiss(animated: true)
        } else {
            // If it's not presented, remove it from the navigation stack.
            self.navigationController?.viewControllers.popLast()
        }
    }

    // ... snip

}
```

The idea behind this approach is that this viewcontroller "can take care of itself" regarding navigation, so that it becomes more self-sufficient, unburdening the parent.

This viewcontroller or "screen" may appear to be more self-sufficient at first. But, this backfires: This viewcontroller basically becomes a captain of the navigation-ship since it now interferes with the navigation stack. It becomes too knowledgeable of where and how it is presented.

By having a screen or view mutate the navigation stack, it will most likely interfere with its parent. Because if a parent is in charge of the navigation stack, this view will now take over, and causes that parent to be out-of-sync with its own state.

Imagine if all views in a flow mutate the navigation stack independently. Then it will be hard to reason about a flow, since there is no single "source of truth" for navigation. This makes it very difficult for a parent flow to keep track of the navigation state and is a surefire way to having bugs.

Not to mention, what if we ever need to embed this view into another? Then it shouldn't do anything *at all* with the navigation stack. Perhaps there shouldn't be a close button, or perhaps the view should only be hidden when pressing the close button.

Embedding this view will not work properly in its current state, and will require refactoring to make this work.

9.1.1 A smarter, yet still problematic, approach

Let's take a smarter, common approach and offload navigation functionality to its own type, such as a coordinator.

NOTE: A Coordinator is a commonly used class that handles the navigation of a flow or app. It is in charge of maintaining the state of navigation by handling the respective navigation-stack.

In the next example, we pass a coordinator to a view. This view has a button that renders a gear icon (also known as a *cog*) for settings. When a user presses on this button, we trigger a specific coordinator action, such as opening the settings screen via `coordinator.openSettings()`.

WARNING: This example still prevents reusable views!

```
struct StealingYourDataPermissionView {

    // This view has a dependency on a Coordinator
    let coordinator: Coordinator

    var body: some View {
        Button {
            // By pressing this button, we tell the coordinator to go
               to the settings page.
            coordinator.openSettings()
        } label: {
            // Renders a gear symbol
            Image(systemName: "gear")
        }

        // ... snip
    }

    // ... snip
}
```

This solution is better than before; This view does not know whether it's presented, and therefore it doesn't mutate the view stack.

Unlike before, this view is not trying to be the captain of the navigation ship. It offloads navigation to a coordinator that has a specific role to handle navigation.

From the view's perspective, it doesn't really matter what the coordinator is. It only needs to know it has navigation-related methods, such as `openSettings()`.

But there are still some problems with this common approach. Let's review them.

9.1.2 The view decides what the parent should be

This view still dictates that it requires a coordinator to function, which is a big assumption to make.

Because, what if we want to use it in a flow that doesn't use coordinators in the first place? What if the parent flow uses routers? Or just glues things together with method calls? Or what if the parent uses unsafe string matching to decide the next view to present? Then forcing a parent flow to use, or be, a coordinator makes little sense.

We *could* increase the flexibility by making `Coordinator` an interface, then a parent can figure out how to implement `openSettings()`.

This is a fair approach. Because usually we can solve tight-coupling by just sticking some interfaces in our code.

But, somehow, it's often not enough when this concerns navigation. Because this view would still limit the flow to coordinators only.

That this view explicitly requires a coordinator creates a situation where a child (the view) decides things for a parent (such as a flow). It's like letting a toddler plan the family vacation — chaos guaranteed.

But that's not the only problem. Let's understand another issue with this approach, after which we'll look at some solutions.

9.1.3 The child orders a parent around

The second problem is that this view now assumes that the gear button should always open the settings screen. Since it calls `openSettings()` method directly, the child view is aware of that parent's method.

But, imagine if we want to place this view in an entirely different flow, such as an onboarding flow. A flow where the user is not logged in yet.

Now, this gear button should not open the settings screen, because the user isn't registered and therefore can't even access this screen.

That's a problem, because this view assumes that the button should always open settings. It's again a situation where the child (view) is telling its parent (a flow) what to do.

Maybe in the onboarding-flow, tapping this gear icon should open the device's location permissions, instead. But that means that the `Coordinator` dependency now must support (or be) a variation where it does something else in this flow.

So either we are introducing a `Coordinator` interface and we pass different coordinators per flow. Or, if we have a single coordinator, we'll be adding an `if` statement somewhere that says *"During*

onboarding, `openSettings()` opens location permissions. But, after registration when the user is logged in, `openSettings()` should open the app's settings page".

Alternatively, maybe there should not be a gear icon during the onboarding flow. Then what would we do?

We could hide the button. But, even if we hid the button, what does this mean for the onboarding flow? Because now this view has a dependency on a coordinator, despite not needing it. Should it pass a dummy coordinator that ignores `openSettings()`? That's a shoddy solution, too.

And what if, in the future, this view needs to be embedded? Then there shouldn't be a coordinator here at all.

Because of the way it works, this view doesn't "just work" when we move it around across flows.

That this view explicitly calls `openSettings()` hinders the reusability, since that creates all these hurdles for parent flows.

In this chapter, we'll cover some solutions, including one for this scenario, where we defer navigation actions to a type, such as a `Coordinator`.

9.2 Preparing for a solution

Now we'll cover some approaches to make views reusable across flow.

In the previous chapters, we worked on `CourseUI` and `CourseView`, and we intentionally didn't support navigation back then.

NOTE: If you might recall, `CourseView` only concerns itself with loading courses, whereas `CourseUI` renders the actual UI and handles the user interaction.

In this chapter, we'll continue improving these views and make them reusable for navigation flows using various methods.

We'll do it in such a way that both `CourseUI` *and* `CourseView` are *unaware* of the navigation they're used in. This is to keep this feature self-sufficient and flexible to use in any flow.

Ironically, the navigation is irrelevant for this feature. Because after we're done, we can theoretically place `CourseView` in tons of navigations and even embedded it in other views. *Spoiler alert: We will.*

9.2.1 What we need to support

Before we come up with some solutions, let's look at what we need to support.

We'll focus on `CourseUI`. Because we left it in an unfinished state regarding navigation.

`CourseUI` has four navigation-related actions, namely: `joinCall()`, `openScheduler()`, `openMessageInbox()`, and `openDetails(todoItem:)`.

Looking at the implementation of `CourseUI`, we can see that we implemented these four methods with placeholder messages, ready to be implemented. Once a user taps the related buttons, these methods will print debug statements with some information about where to implement it.

```
struct CourseUI: view {
    // ... snip


    var body: some View {
        // ... snip

        // Views such as ScheduleView, refer to the methods such as
           joinCall or openSCheduler.
        ScheduleView(date: course.calendarEvent?.date, joinCall:
           joinCall, openScheduler: openScheduler)
    }

    // Navigation-specific. Called when user taps on buttons.
    private func joinCall() {
        print("Implement \(#function) on \(#file), line: \(#line)")
    }

    private func openScheduler() {
        print("Implement \(#function) on \(#file), line: \(#line)")
    }

    private func openMessageInbox() {
        print("Implement \(#function) on \(#file), line: \(#line)")
    }

    private func openDetails(todoItem: todoItem) {
        print("Implement \(#function) on \(#file), line: \(#line)")
    }
}
```

But, notice a problem (besides the methods not being implemented)? Try to figure it out before reading on.

The answer is that this view *assumes* that pressing on a button opens a specific flow. Even if we haven't implemented the methods yet, the view calls methods such as `joinCall()` or `openScheduler()`.

To remedy this, the view should only relay the action that happens, so that a parent – such as a coordinator, or flow, or router – can decide for themselves what to do.

To solve this, let's rename the methods: `joinCall()` becomes `didPressJoinCall()`, `openScheduler()` becomes `didPressSchedule()`, and so on.

```
struct CourseUI: view {
    // ... snip

    // Previously joinCall()
    private func didPressJoinCall() {
        print("Implement \(#function) on \(#file), line: \(#line)")
    }

    // Previously openScheduler()
    private func didPressSchedule() {
        print("Implement \(#function) on \(#file), line: \(#line)")
    }

    // Previously openMessageInbox()
    private func didPressMessageButton() {
        print("Implement \(#function) on \(#file), line: \(#line)")
    }

    // Previously openDetails(todoItem:)
    private func didPressTODOItem(_ todoItem: todoItem) {
        print("Implement \(#function) on \(#file), line: \(#line)")
    }
}
```

We are essentially giving the names a lower abstraction. For example, we move from `openMessageInbox()` (higher abstraction) to `didPressMessageButton()` (a lower abstraction). This is the key.

This sets the proper expectations, even when we haven't implemented the methods. Now, this view is oblivious to *any* navigation-aspect, it only focuses on user interaction. This allows a parent to decide what to do after we finish the implementation.

We now know which actions to support, but we are still printing debug statements. Before we implement these methods, let's first continue by designing a solution.

9.3 Considering various solutions

Let's design how we would implement `CourseView` from the call-site, or parent's perspective, then we can worry about the actual implementation afterwards.

This parent could be anything. Today, it could be a master-detail screen, a tab bar, or `MarketplaceView`, but tomorrow it could be a `CoursePresenter`. Which is why we should

design `CourseView` as a *standalone* view that works *regardless* of the parent, that makes no assumptions about navigation.

We can take various approaches to implement navigation triggers; As an example, we could use closures as actions, or pass around an interface to respond to actions, or declaratively build up a type with a mix of methods and actions. These approaches are what we'll cover in this chapter.

But there are more ways; Maybe you want to pass messages, use a central dispatch, or implement a router, and so forth.

Whatever approach you prefer, **making a view oblivious to navigation is *key***. This ensures that your views remain loosely coupled and gives you flexibility to reuse views in various multiple flows, whether they are the current flows or upcoming flows that may come in the future.

The parent decides the actions – or triggers – when a user presses navigation-specific buttons. All that `CourseView` and `CourseUI` will communicate in this case is *"The user pressed on the messages button"* or *"The user pressed on the phone button"*.

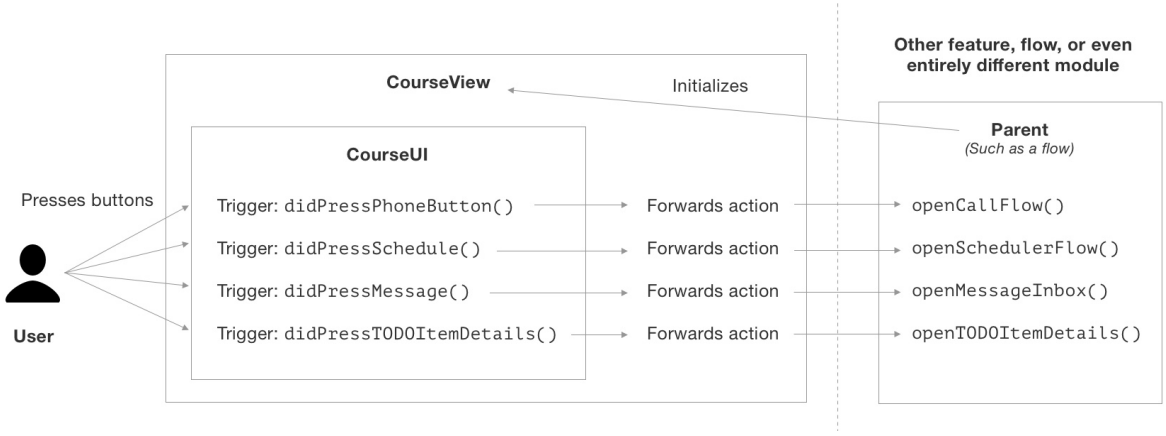
First, let's design our solution using closures. Since it's quite common and fits the declarative approach well.

9.4 Using closures

One approach is to work with closures. This allows us to decouple the view from its parent, such as a navigation flow.

The way it works, is that a user will press buttons on the UI of `CourseUI`, triggering a closure, which will forward this to `CourseView` (since `CourseUI` is nested).

Then, `CourseView` will forward these actions to its parent. Again, this parent could be anything, such as a coordinator or router, or even a feature in an entirely different module.



NOTE: We can safely assume that `CourseUI` is hidden from the parent to keep the Course API smaller and easier to implement. `CourseUI` is only to be used by `CourseView`.

Notice that the parent doesn't have to share the same action names as `CourseUI`. It's likely that the parent will have higher-abstraction names (such as `openCallFlow()`) as opposed to lower-abstraction names (such as `didPressJoinCall()` in `CourseUI`).

9.5 A closure implementation

Following the Holistic-Driven Development approach, let's design this API from a parent's perspective. After that, we'll dive deeper and update both `CourseView` and `CourseUI`.

Looking at how a parent initializes `CourseView`, we can see how we would pass the closures. There are four actions to implement, and we could pass them in the initializer, for instance.

To keep momentum, we'll use placeholders in this first step.

```
CourseView(courseService: courseService, ID: ID,
           didPressJoinCall: {
               print("Implement the action on \(#line)")
           }, didPressSchedule: {
               print("Implement the action on \(#line)")
           }, didPressMessageButton: {
               print("Implement the action on \(#line)")
           }, didPressTODOItem: { todoitem in
               print("Implement the action on \(#line)")
           })
```

NOTE: How we pass the actions it isn't too important, and is prone to subjective debates. As long as the parent handles the navigation, and the view is oblivious, you're going in the right direction.

By pressing on the buttons, the parent would print out the related line, so that it's ready to be implemented.

Before we connect these closures to `CourseUI`, let's first understand how we would connect a parent flow to `CourseView` by removing the placeholders.

9.5.1 When a parent is a coordinator

Often, a parent is a coordinator: A type that glues views together into a specific flow.

For instance, let's assume we have a `CourseCoordinator` that sets up a `CourseView` instance. By doing so, it glues the `CourseView` actions to its own methods.

```
class CourseCoordinator {

    // This method makes a CourseView instance
    // We can imagine that it's triggered from a flow
    func makeCourseView(courseService: CourseService, ID: UUID) ->
        CourseView {
         CourseView(courseservice: courseservice, ID: ID,
                    // We connect the actions to the methods inside this
                        coordinator
                    didPressJoinCall: {
                        openJoinCall()
                    }, didPressSchedule: {
                        openScheduler()
                    }, didPressMessageButton: {
                        openMessageInbox()
                    }, didPressTODOItem: { todoItem in
                        openTODOItemDetails(todoItem)
                    })
    }


    // These methods open other flows. Triggered by CourseView's
        actions.

    func openJoinCall() {
        // ... implementation omitted
    }

    func openScheduler() {
        // ... implementation omitted
    }

    func openMessageInbox() {
        // ... implementation omitted
    }

    func openTODOItemDetails(_ todoItem: TODOItem) {
        // ... implementation omitted
    }

    // ... snip
}
```

Notice how, unlike `CourseView`, this coordinator *is* aware of the actions that happen after a user presses a button. For instance, when a user triggers `didPressSchedule()`, this coordinator knows it should call `openScheduler()` and update the navigation stack.

How `CourseCoordinator` opens a scheduler (or other views) isn't relevant to `CourseView` and isn't relevant to us in this example. This was one example, but remember, the parent doesn't *have* to be a

coordinator. It can be something else entirely.

Notice that with this approach, we can connect anything to `CourseView`. Today it's a coordinator calling methods such as `openJoinCall()`, but tomorrow it could be a view that embeds `CourseView` that does something else entirely.

Before we move on, let's clean up the code a bit.

9.5.2 Cleaning up the call-site

By wrapping these methods in a closure, we are adding a layer of indirection for little value. Instead, we can pass the methods directly without wrapping them in a closure.

Now, instead of calling the methods from the closures, we pass the method *names*.

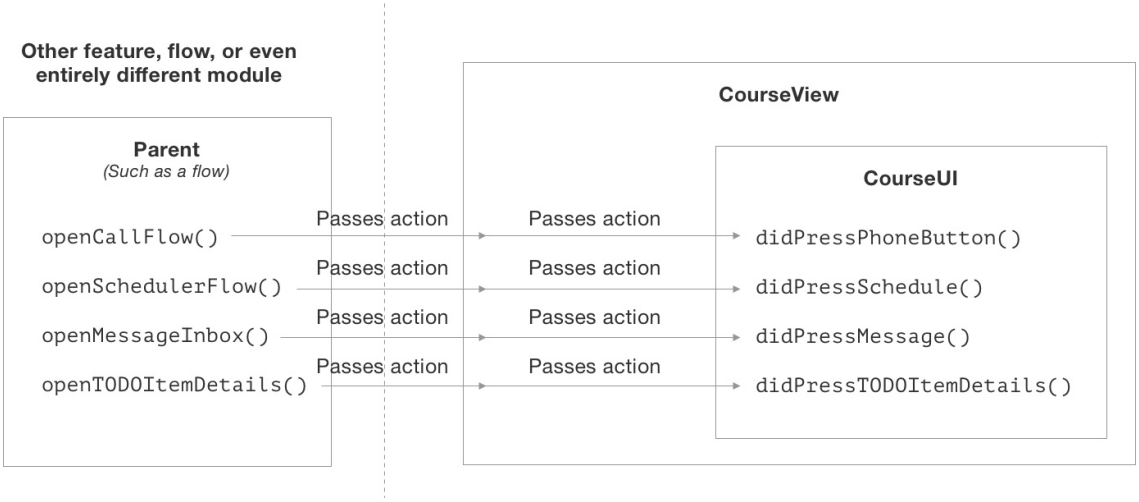
```
1  class CourseCoordinator {
2
3      // This method makes a CourseView instance
4      // We can imagine that it's triggered from a flow
5      func makeCourseView(courseService: CourseService, ID: UUID) ->
           CourseView {
6          CourseView(courseservice: courseservice, ID: ID,
7                     didPressJoinCall: openJoinCall,
8                     didPressSchedule: openScheduler,
9                     didPressMessageButton: openMessageInbox,
10                    didPressTODOItem: openTODOItemDetails)
11     }
12
13     // ... snip
14 }
```

Before we consider the downsides of the closure-approach, let's finish this implementation by looking at how `CourseView` would connect this to `CourseUI`.

9.5.3 Gluing everything together

We've seen how `CourseUI` triggers an action (closure), forwards it to `CourseView`, and then forwards that to the parent.

To set it up, we flip the diagram. The parent passes an action closure to `CourseView`, which passes this closure to `CourseUI`, which will then use that when a user taps on buttons.



This is how we'll implement the solution; `CourseView` will accept closures and forward them to `CourseUI`. Then, `CourseUI` will connect these closures to its UI elements.

9.5.4 Inside CourseView

Inside `CourseView`, we would offer these actions as closure properties, which it will pass on to `CourseUI`. That means that `CourseUI` will now also accept these four actions (closures) to be passed in.

```
struct CourseView: View {

    // We'll add new actions
    var didPressJoinCall: () -> Void
    var didPressSchedule: () -> Void
    var didPressMessageButton: () -> Void
    var didPressTODOItem: (TODOItem) -> Void

    // ... snip

    var body: some View {
        // We pass the actions to CourseUI.
        CourseUI(course: courseBinding,
                 didPressJoinCallAction: didPressJoinCall,
                 didPressScheduleAction: didPressSchedule,
                 didPressMessageButtonAction: didPressMessageButton,
                 didPressTODOItemAction: didPressTODOItem,
        )

        // ... snip
    }

}
```

9.5.5 Inside CourseUI

Inside `CourseUI`, we have four placeholders ready to be implemented.

We can replace the placeholders with the action implementation since it can now call the action closures.

Notice that we suffix these closures with `Action` so their names don't clash with the same-named methods.

```
struct CourseUI: View {
    // ... snip

    // We'll add new actions
    var didPressJoinCallAction: () -> Void
    var didPressScheduleAction: () -> Void
    var didPressMessageButtonAction: () -> Void
    var didPressTODOItemAction: (TODOItem) -> Void

    var body: some View {
        // ... snip

        // Views such as ScheduleView, refer to the methods such as
            didPressJoinCall
        ScheduleView(date: course.calendarEvent?.date, didPressJoinCall
            : didPressJoinCall, didPressSchedule: didPressSchedule)
    }


    // We trigger these actions by replacing the placeholders with the
        actions.
    private func didPressJoinCall() {
        didPressJoinCallAction()
    }

    private func didPressSchedule() {
        didPressScheduleAction()
    }

    private func didPressMessageButton() {
        didPressMessageButtonAction()
    }

    private func didPressTODOItem(todoItem: TODOItem) {
        didPressTODOItem(todoItem)
    }
}
```

Now, the closures that are passed from a parent – such as `CourseCoordinator` – are now connected to the UI.

9.5.6 Cleaning up CourseUI

Currently, the methods only call the actions. But there is no added benefit of having these methods, and all they do is add more indirection, making it harder to follow our code.

Let's clean this up a bit, and remove this one layer of indirection, like we did before when connecting methods from the coordinator to `CourseView`.

First, we can remove the methods, such as `didPressSchedule()` and `didPressJoinCall()`.

Next, we'll remove the `Action` suffix from the closures, since we don't need to make a distinction anymore between the action closures and the private methods.

Originally we would pass methods to the views, but now we pass the action closures instead. Notice that since the closures use the same name as the deleted methods, we don't need to change how we pass these closures to the related views.

```
struct CourseUI: View {
    var body: some View {
        // ... snip

        // Views still refer to the same actions, this isn't changed
        ScheduleView(date: course.calendarEvent?.date, didPressJoinCall
            : didPressJoinCall, didPressSchedule: didPressSchedule)
    }

    // We can delete these now:
    // private func didPressJoinCall() {
        // didPressJoinCallAction()
    // }

    // private func didPressSchedule() {
        // didPressScheduleAction()
    // }

    // private func didPressMessageButton() {
        // didPressMessageButtonAction()
    // }

    // private func didPressTODOItem(todoItem: TODOItem) {
        // didPressTODOItem(todoItem)
    // }

    // ... snip
}
```

9.5.7 Trade-offs when using closures

Declarative UI invites us to nest views. but when forwarding closures, it can create more boilerplate as we experienced.

Closures are great for a few actions. But the problem of using closures is that it doesn't really scale. We would have to introduce some boilerplate. Four actions are *doable*, but imagine having to pass eight or 12 methods, it can become quite messy.

Another problem is that, the more nested views become, the more we would pass these closures around. Passing closures from `CourseView` to `CourseUI` is one layer deep, but imagine if we went one level deeper, we would multiply the boilerplate from 8 (2x4) to 12 (3x4) closures.

The more actions we need to pass, and/or the more views become nested, the more closures we have to pass around. Once the depth *or* number of actions grows, the more we're inclined to use a different solution.

Let's go over some alternative solutions by first introducing a separate type. After that, we'll cover a declarative approach.

9.6 Passing a navigation type

Often, if we're juggling multiple closures, it might be easier to introduce a separate type to handle the navigation for us.

This could be many things, such as a class, a struct containing closures, or an interface – known as a *protocol* in Swift terms.

However, earlier in this chapter, we covered how passing a coordinator to a view harms its reusability.

Let's look at this approach in more detail to see how we can still use a separate type to handle our navigation for us *while* we keep the ability to reuse a view across flows.

9.6.1 A navigation interface

For instance, instead of a `CourseCoordinator` class, we may introduce a `CourseCoordinator` *interface* that implements the four navigation methods.

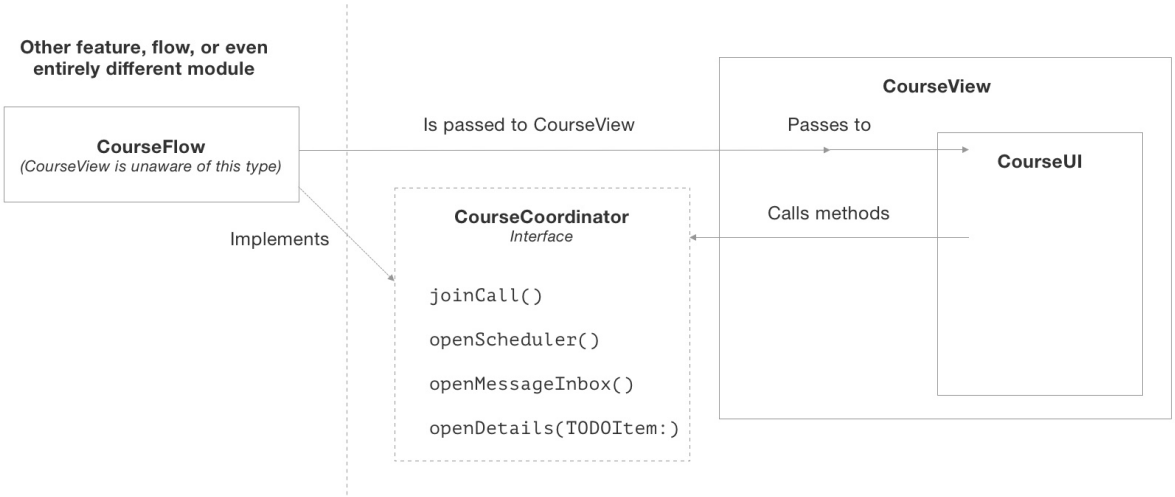
NOTE: In Swift, we declare interfaces using the protocol keyword.

```
1 protocol CourseCoordinator {
2     func joinCall()
3     func openScheduler()
```

```
4       func openMessageInbox()
5       func openDetails(TODOItem:)
6   }
```

Then we could pass a type conforming to this interface to `CourseView`.

For example, we may have a `CourseFlow` class that implements `CourseCoordinator`. This gets passed to `CourseView`. Underwater, `CourseView` would pass this type to `CourseUI`, after which `CourseUI` would call its methods when a user presses buttons.



From the call-site, it would look as simple as passing `CourseFlow` via an initializer, which I'd argue is more readable than passing four action closures.

```
1  CourseView(courseService: courseService,
2             ID: ID,
3             // We now pass a courseFlow instance to CourseView.
4             coordinator: courseFlow)
```

Note that `CourseView` and `CourseUI` actions are oblivious to `CourseFlow`, they only know that they receive a type conforming to `CourseActions`.

By making this an interface, it opens up a lot of flexibility. Now anything that implements this `CourseCoordinator` interface can glue together these actions however they wish.

Passing a type is easier than connecting closures together. Let's say we want to add three more actions, then we don't have to keep adding closures to `CourseView` and `CourseUI`. We merely have to define them on the interface – and ultimately the type that implements it. Which, I would argue, is easier than passing three more closures to `CourseView` and `CourseUI`.

Especially if `CourseView` changes a lot internally, it's easier to pass this type around, as opposed to connecting many closures together, which is as enjoyable as untangling earbud wires.

9.6.2 The problem of passing a navigation type

One problem we covered is that by using a navigation type, such as a coordinator, a view will make too many assumptions about how it's used in a flow.

`CourseView` now dictates that, in order to use it, we *must* pass it some kind of coordinator. Even if it's a flexible interface, it's still a coordinator.

But *another* problem is that `CourseUI` is now aware of navigation-specific methods.

Because to use this coordinator, `CourseUI` would call methods such as `coordinator.joinCall()` or `coordinator.openScheduler()`, which are navigation-specific.

The problem lies mostly in the name and responsibilities. Let's continue by giving the interface and its method a better name. This will give us a more flexible view.

9.6.3 The name sets expectations

First, by *not* calling the interface `CourseCoordinator`, we can ease that this type "must be a coordinator".

We could rename it to something more generic, such as `CourseNavigation`.

It's better, but the problem remains. The view still assumes that all actions (triggers) are used for navigation. But that's not always the case.

What if tapping on *'join call'* should not navigate anywhere?

For instance, if a user presses the 'join call' button, a parent flow may decide to show an alert *"Warning: This will start a phone call"* instead. As opposed to directly triggering a phone call, scaring millennials.

But it could even be something else; Maybe a parent coordinator may trigger a deep link to Google Meet or open a contacts screen.

Remember, the goal is that the view has *no idea* about navigation. So when we pass a navigation object, coordinator or otherwise, views become aware of navigation.

To solve this, let's rethink the name. Perhaps we can rename this type related to what a user does, or the actions. Let's try `CourseActions`.

That's better. We took the navigation-aspect out of it.

But, there is a second problem to solve: The view should not be calling methods such as `openScheduler()` or `openTODOItemDetails()`. Because, as you might recall, the view then wrongly assumes it triggers navigation-related actions.

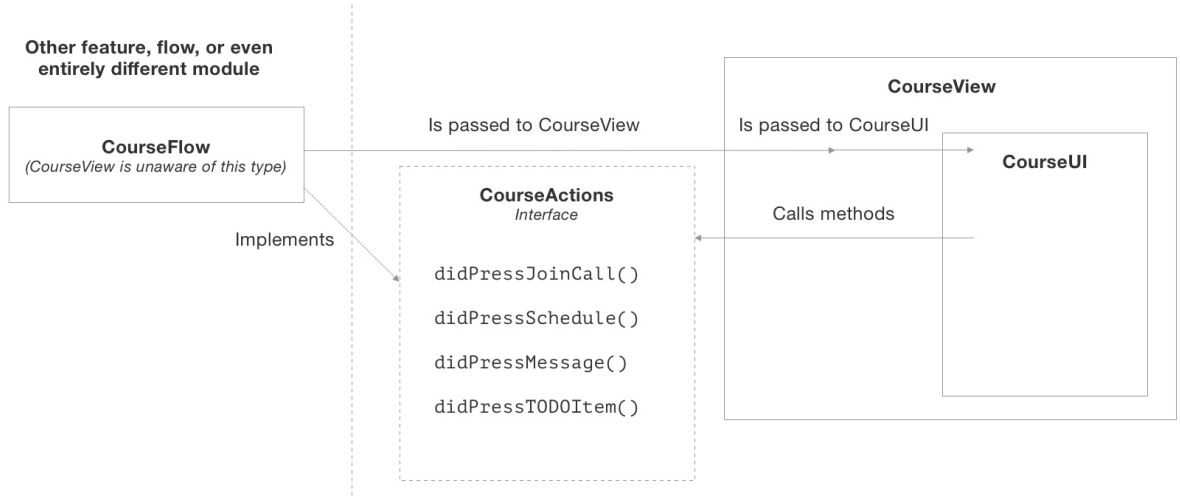
Instead, the method names should reflect something on a lower abstraction. Let's try something related to actions.

These actions could be `didPressTODOItem()` or something like `didPressSchedule()`, then a parent can freely decide what to do as a response.

Using this knowledge, let's update the interface name and its methods.

```
1 // The interface (protocol) is renamed to something related to actions
2 protocol CourseActions {
3     // The method names only infer what a user did. It doesn't dictate
          what should open.
4     func didPressJoinCall()
5     func didPressSchedule()
6     func didPressMessageButton()
7     func didPressTODOItem todoItem:)
8 }
```

Now a parent type, such as a `CourseFlow`, would implement the `CourseActions` interface, and implement methods such as `didPressJoinCall()`, as opposed to implementing a `CourseCoordinator` interface and implementing methods such as `openScheduler()`.



With our solution, `CourseUI` is completely oblivious again to navigation. Using proper naming solves the problem we covered in the beginning of this chapter, because it sets the proper expectations and fully decouples a view from a parent.

9.7 Dealing with optionality

We have some excellent tools on our belt to decouple a view from its navigation.

If you want to stop the chapter here, that's absolutely fine. But if you want to go deeper down the rabbit hole with me, then let's consider more approaches to design the API of `CourseView`.

Let's cover what it would look like if methods become *optional*. Because when we move a view to a new flow, we might have to tweak or alter a view. Such as hiding or showing specific buttons because they may not fit the flow.

For fun, let's get fancy and design a declarative approach. Because a declarative approach embraces optionality, allowing us to define actions we may or may not implement.

Following the Holistic-Driven Development tradition, we'll start by designing the API from the call-site. Then we don't have to worry about the implementation just yet.

9.7.1 Designing a declarative solution

Notice how in the upcoming example that we can call methods one after another using a declarative approach.

These methods could be `didPressSchedule(action:)` or `didPressJoinCall(action:)`. The actions inside these methods are closures or methods that we pass.

Also notice that we keep chaining, as opposed to passing closures, in an initializer.

This way, we declaratively "build up" and modify the `CourseView` type.

```
1 // We can chain method calls. We are not passing actions to the
     initializer anymore.
2 CourseView(courseService: courseService, ID: ID)
3   .didPressJoinCall(action: openCallFlow)
4   .didPressSchedule(action: openCalendarSchedule)
5   .didPressMessageButton(action: openChat)
6   .didPressTODOItem(action: openTODOItemDetails)
```

Chaining methods fits the declarative paradigm of "building up" a type. This is like chaining modifiers in Jetpack Compose.

Before thinking about how to implement this solution, let's consider the implications.

9.7.2 Reasoning about optionality

One key component to be aware of is that with a declarative style, all these actions are now *optional*.

For instance, we can decide to only implement the `didPressJoinCall(action:)` method, but now we can leave the other methods alone.

```
CourseView(courseService: courseService, ID: ID)
  .didPressJoinCall(action: openCallFlow)
  // We decide *not* to implement the other actions
```

Before, when we pass all actions to the initializer, it's mandated that we implement all methods. In fact, the compiler would yell at us if we forgot one, giving us compile-time safety (assuming we don't offer default values in the initializer).

With the declarative API, it's easy to add (or forget) actions. Setting up actions isn't mandatory anymore.

We are trading compile-time safety for ergonomics. Because the compiler won't warn us if we forget an action. The good news is that we *do* get flexibility in return.

There are some solutions to deal with optionality:

- We can embrace optionality by omitting view elements for methods that aren't called. For instance, if we don't implement the `didPressMessageButton()` method, then we can make sure that `CourseView` does *not* render the message button; This makes `CourseView` more configurable if needed.
- We can add default actions when we don't implement these actions, such as triggering a `/messages/` URL for deep-linking once a user presses the message button. We can choose to do this if the parent didn't configure the `didPressMessageButton()` method.
- Once a user triggers an action, we can show a debug message telling the developer that something isn't implemented yet.
- We can call an assertion that will crash debug builds to make it explicit that something isn't implemented yet.

These are just suggestions to deal with optionality. But the point is, it's good to know that with a declarative approach, we are building up a type's complexity. Meaning that we would support both a basic and more complex or "decorated" variations.

9.7.3 Embracing optionality

If you're taking a declarative approach, I would recommend to embrace the optionality, as opposed to throwing errors or warnings when we haven't implemented an action yet. Because when modifying a

declarative element, it's the convention that each individual step is a valid variation.

For instance, if we define a `Text("I like bagels")`, we expect it to be a valid `Text` view. Then, if we modify this declaratively, we still consider `Text` valid after each step.

We can keep decorating the `Text` element, and in-between each step, it's convention that the `Text` remains a valid `Text` element or something similar.

```
Text("I like bagels")
  // Still valid
  .frame(maxWidth: .infinity, alignment: .leading)
  // Still valid
  .padding(.leading, 20)
  // Still valid
  .font(.footnote)
  // Still valid
  .padding(.bottom, 2)
```

We should consider the same for the views we offer, where each individual step or modification is a valid variant. This makes the implementation more configurable and thus more complex, but this is well-suited if we want that flexibility.

For instance, in each step of the next code listing, `CourseView` remains a valid version, even if we don't configure all actions.

```
// This is a valid CourseView
CourseView(courseService: courseService, ID: ID)
   // Still valid (may now display a phone button)
  .didPressJoinCall(action: openCallFlow)
   // Still valid (may now display the schedule buttons)
  .didPressSchedule(action: openCalendarSchedule)
   // Still valid (may now display a message button)
  .didPressMessageButton(action: openChat)
   // Still valid (may now offer buttons on TODO items)
  .didPressTODOItem(action: openTODOItemDetails)
```

That concludes the API design. Let's go deeper down the rabbit hole once again and look at a way to implement this.

9.8 Implementing a declarative API

We can enable chaining on the same type by returning the same type again after calling a method on it.

In other words, when we call a declarative method on `CourseView`, we need to ensure that it returns a new `CourseView`.

Let's implement `didPressJoinCall()` as the first declarative method. We need to ensure it returns an updated copy of `CourseView` containing the relevant action.

NOTE: The reason we need to make copies is that SwiftUI views are structs (value types) and not classes (reference types).

Before we deal with actions, let's first make an empty implementation to make the compiler happy.

```
struct CourseView: View {

    // This declarative method returns a CourseView instance.
    func didPressJoinCall(/* The action is undefined for now */) ->
        CourseView {
        // We make a copy
        var copy = self

        // TODO: We need to modify this copy so it accepts a new action

        // We return a copy
        return copy
    }

    // ... rest omitted
}
```

Since this method returns a `CourseView`, we can already start chaining this method if we wanted. Even if it does nothing yet (because we aren't passing any actions).

```
CourseView(courseService: courseService, ID: ID)
   // This doesn't do anything, but at least we can chain the method.
   // Notice that we aren't passing actions yet.
  .didPressJoinCall()
  .didPressJoinCall()
  .didPressJoinCall()
```

9.8.1 Glueing the action to the new instances

Now it's time to connect the action to the copy.

First, we make sure that `didPressJoinCall` accepts an action as a parameter. Then, to connect the action, we can set the action to a property of the copy.

Like before, we exit the method by returning the copy. But, this time, the copy contains the action.

```
struct CourseView: View {

    // Now we accept an action.
    func didPressJoinCall(action: @escaping () -> Void) -> CourseView {
```

```
        var copy = self
        // NEW: We set the new action on this copy.
        copy.didPressJoinCall = action
        return copy
    }

    // ... snip
}
```

NOTE: A Swift detail: We need to mark this action as `@escaping` to show that the closure is stored and will outlive the method call.

We just set an action on a `didPressJoinCall` property. To achieve this for all four actions, `CourseView` must support these actions as properties. This is very similar to the closures solution from earlier in this chapter.

However, this time, these actions need to be optional because we might not set all actions. To make these closures optional in Swift, we need to wrap them in another pair of parenthesis and use the `?` keyword.

```
struct CourseView: View {

    // These actions are now optional.
    // We achieve this by wrapping them inside parenthesis, and
       suffixing it with the '?'
    var didPressJoinCall: (() -> Void)?
    var didPressSchedule: (() -> Void)?
    var didPressMessageButton: (() -> Void)?
    var didPressTODOItem: ((TODOItem) -> Void)?

    // ... snip
}
```

This allows us to set actions as properties. In fact, we can forego the chaining approach and use properties instead. This will save us from the boilerplate.

```
// Inside some method
let courseView = CourseView(courseService: courseService, ID: ID)
// We pass the actions as properties
courseView.didPressJoinCall = openCallFlow
courseView.didPressSchedule = openCalendarSchedule
courseView.didPressMessageButton = openChat
courseView.didPressTODOItem = openTODOItemDetails
// But now, depending on the location, you may have to explicitly
   return the view.
return courseView
```

This is still declarative in the sense that we state what we want, and letting `CourseView` handle the rest. But this just doesn't chain the way SwiftUI does.

It might not be as fancy, but achieves a similar result.

Assigning values to properties are statements, not expressions. Assigning a property returns nothing. In contrast, the chaining methods always return a new instance of `CourseView`.

Because of this, we now need to return the `CourseView` instance explicitly on the last line.

9.8.2 Implementing all declarative methods

Let's finish up the declarative implementation. We define the four declarative methods and connect the respective actions to the copies.

```
1  struct CourseView: View {
2
3      // These actions are now optional.
4      var didPressJoinCall: (() -> Void)?
5      var didPressSchedule: (() -> Void)?
6      var didPressMessageButton: (() -> Void)?
7      var didPressTODOItem: ((TODOItem) -> Void)?
8
9      // These methods now accept the action to call once they are
           triggered.
10     func didPressJoinCall(action: @escaping () -> Void) -> CourseView {
11         // We make a copy
12         var copy = self
13         // We wire the new action to this method.
14         copy.didPressJoinCall = action
15         // We return a copy
16         return copy
17     }
18
19     func didPressSchedule(action: @escaping () -> Void) -> CourseView {
20         var copy = self
21         // We set the didPressSchedule action on the copy
22         copy.didPressSchedule = action
23         return copy
24     }
25
26     func didPressMessageButton(action: @escaping () -> Void) ->
           CourseView {
27         var copy = self
28         // We set the didPressMessageButton action on the copy
29         copy.didPressMessageButton = action
30         return copy
31     }
32     func didPressTODOItem(action: @escaping (TODOItem) -> Void) ->
           CourseView {
33         var copy = self
34         // We set the didPressTODOItem action on the copy
```

```
35          copy.didPressTODOItem = action
36          return copy
37      }
38
39      // ... rest omitted
40  }
```

This is enough to connect the actions to `CourseView`. It contains boilerplate, but it's a good starting point. Perhaps with some code generators or macro tools, we could improve this solution even further.

The last piece of the puzzle is to connect this to `CourseUI`. Let's look at that next, after which we'll be done.

9.8.3 Connecting to CourseUI

Similar to the closure solution, CourseView still passes the action closures to CourseUI as it did before. That *does* mean that `CourseUI` now must accept *optional* closures and react such as omitting UI elements.

```
1  import SwiftUI
2
3  struct CourseView: View {
4
5      var body: some View {
6          // We pass the actions to CourseUI, the same as before.
7          // But now, CourseUI accepts optional closures, too.
8          CourseUI(course: courseBinding,
9                   didPressJoinCallAction: joinCall,
10                  didPressScheduleAction: openScheduler,
11                  didPressMessageButtonAction: openMessageInbox,
12                  didPressTODOItemAction: didPressTODOItem,
13          )
14
15          // ... snip
16      }
17
18      // ... snip
19  }
```

Since `CourseUI` isn't part of the Course API, it doesn't need declarative methods that return itself.

But because `CourseView` now passes optional closures, `CourseUI` is forced to act on that. With a little imagination, we can decide to omit certain views if these methods aren't called. For instance, if `didPressMessageButton` is nil, we can make sure that `CourseUI` omits the message button.

9.8.4 Chaining forever

Now everything works as intended. We can call `didPressJoinCall(action:)` on `CourseView`, it will return a new instance with that specific action coupled to the `CourseView` instance. Then, once a user presses a button, such as the 'join call' button, it will trigger the proper action.

Because the methods return a new `CourseView`, we can keep chaining them forever, if we so desire.

Silly enough, we can even call the same methods every time, since the type returned is always `CourseView`. But it showcases how `CourseView` is now composable.

For example, we can keep calling `didPressSchedule()` with four different actions.

```
1 CourseView(courseService: courseService, ID: ID)
2   .didPressSchedule(action: openScheduler1)
3   .didPressSchedule(action: openScheduler2)
4   .didPressSchedule(action: openScheduler3)
5   .didPressSchedule(action: openScheduler4)
```

This isn't different from repeatedly calling modifiers on other declarative types. Such as repeatedly calling `bold()` on a single `Text` element, hoping it becomes **super-duper bold text**.

```
1 Text(verbatim: "Why am I doing this?")
2   .bold()
3   .bold()
4   .bold()
```

But, since `CourseView` is within our control, we can decide what repeated actions will cause. You can choose that calling `didPressSchedule` on `CourseView` only calls the last invocation, such as `openScheduler4`. Or you might choose to trigger all actions.

If you want, you can add more protection measures to prevent misuse. For example, you can print or throw errors when the same method is called multiple times.

9.9 Conclusion

In this chapter, we took a close look at how to decouple a view from its navigation. We've covered common problems and used three different solutions to ensure a view is reusable across navigation flows.

NOTE: This chapter is the end of the series where we handle single views, `CourseView` in particular. Interesting how many lessons we can derive from a single view!

We've seen how we can use closures, interfaces, and even a declarative approach to design a view's API's.

We haven't even created a mature navigation flow. But that's the point. Because the key takeaway is that a view must be completely oblivious to navigation. It merely forwards messages such as *"The user tapped on the settings icon"*.

Forwarding messages such as `didTapGearIcon()` is quite a low abstraction, compared to making views call more high-level methods such as `openSettings()`. But, that's precisely what allows a view to be reused, so that a parent flow can focus on the navigation, instead.

A mental exercise is to pretend that you're a vendor. Meaning, you deliver your views to teams in other companies. Then you might not know how an external team implements your views.

With this perspective, a view directly calling `openPaymentFlow()` is an unreasonable assumption to make for other companies.

That means decoupling becomes key. But even when decoupling, we've seen how closures are hard to scale, and when using interface types, naming is crucial. Because if a view still calls `openMail()` on an interface, it's still assuming too much for a navigation flow.

Whenever you see a view call specific methods related to navigation, it might be good to pay extra attention and see if it doesn't assume too much.

Now that know how to decouple views from navigation, let's continue to see how we can build adaptable navigation flows.

9.10 What we covered

In this chapter, we covered:

- When a view can close or dismiss itself, it likely interferes with a parent flow.
- When a view calls methods related to navigation, it harms its reusability.
- When a view is too aware of navigation, it makes it harder to embed it.
- The action names are important for when a user presses. This removes any assumptions that a view has about its navigation.
- Instead of focusing on a specific navigation action, make the method names reflect the button presses.
  - Ensure that a view has a lower abstraction of button names, e.g. `didPressCloseButton`.
  - Ensure that a parent (flow) has a higher abstraction for the specific action, e.g. `dismissView`.

Closures as actions:

- We can use closures as actions to decouple a view from its navigation.
- Closures as actions work well on a smaller scale.
- The more closures we pass, the more unwieldy a view becomes to set up.
- The more we nest a view, the more boilerplate we have to pass action closures.

Passing around a navigation type:

- When the number of actions grows, it can be easier to pass a dedicated type related to navigation.
- An interface can be a good way to decouple a view from its navigation.
- Naming is crucial when passing a type, even when using interface, because a view might still be too aware of its navigation.
  - Ensure that the interface reflects nothing about navigation.

Optional actions:

- When making a view reusable, you often want more configurability. E.g. hiding buttons in a different flow.
- Optional views can increase complexity.
- One approach to handle optionality is to use a declarative approach. This allows you to build up a view.
- When using a declarative approach:
  - You need to return the same type to enable chaining.
  - It's a convention that each modifier results in a valid view.

Want to read more?

Enjoyed this sample? There's a lot more to discover in Mobile System Design! The book dives deep into turning real-world challenges into robust mobile solutions, covering everything from planning and testing to UI architecture, reusable components, and scaling up an app, including modular design and design systems.

Use code `SAMPLIST` for an exclusive 10% discount on the full book.

What you'll learn:

- How to transform briefs into actionable development plans
- Strategies for writing testable, maintainable code with minimal effort
- Practical dependency injection without unnecessary complexity
- Decomposing designs into reusable, scalable UI components
- The art of building self-sufficient features that stay nimble
- Managing complex navigation and crafting resilient UI flows
- When and how to implement a UI library or design system
- Scaling mobile architectures while maintaining quality

Buy the book at **www.mobilesystemdesign.com** and level up your mobile system design skills!