Architecture of SwiftUI components

Motivation

The motivation behind this proposal is to separate the SwiftUI components' implementations into layers with distinct responsibilities. This will make our system more scalable, maintainable, consistent and resilient. It's important to ensure that the design of our library is easy to reason about in order to allow for fast, effective iteration.

Maintainability

Our current SwiftUI implementations are quite rigid when it comes to support of different visual layouts or styles. Their API, their data transformation logic, and their visual appearance are all placed in a single layer. To be prepared for major visual updates, like the inputs redesign or visual expression work, the system needs to support multiple styles in parallel during development and migration. This would let us ship all the changes at once as well as allow to gradually roll out the new layouts to catch any issues early before mass release.

Scalability

Having a single layer with all sort of responsibilities is not just a problem for future improvements. The current structure makes it difficult to reuse certain components inside more compound ones, especially where slight visual changes are necessary. For example, to compose Money Input from Text Input and Select, both of them needs to have alternative layouts while their API and functionality should remain unmodified. We could either build those differences into that single layer with bunch of `if-else` branches, making our implementation more complex than it should be. Or we re-architecture our components based on responsibilities. This document addresses how to do the latter one efficiently.

Usability

Our current SwiftUI implementations are heavily influenced by their UIKit counterparts. However, while UIKit is an imperative framework driven by events, SwiftUI is a declarative one, driven by states. As we mature with SwiftUI, we should embrace the difference in paradigm on our implementations. If we don't, it's going to block us leveraging its full potentials as well as forcing us to make shady workarounds to implement the functionality we want. Last but not least, shipping APIs that are alien to the system can harm its usability as devs needs to mix different paradigms and learning uncommon patterns.

Proposal

Inspiration

The way how Apple designed SwiftUI components is really flexible, which is no surprise, as it's a general purpose UI library. Even though we don't have to make our APIs as open, the underlying layers are great example of how to divide responsibilities. The built in SwiftUI components have 3 different layers:

- **API** that represents the component and collects all the content to display
- **Style** that specifies the layout and appearance of the component
- **Style Configuration** that converts the raw content data into renderable views connecting the **API** and the **Style**

Below I will give more details about each, talk about the supporting types required, user interactions and accessibility concerns.

Layers overview

The following figure show how the information is transformed when travelling through the layers.

1. The developer passes in content data to the API layer, like Strings, Bools, etc.
2. The API layer aggregates the content with internal states if it has any, for example focus state.
3. The aggregated data is passed to the Style Configuration which converts them into functioning renderable views, like Label, Toggle, Select, etc to abstract the raw data from the Style.
4. The Style apply the proper appearance on those views and lays them out according to the UI design.
5. The final layout then presented on the user interface to the user.

To see how these layers help us solve the previously mentioned issues (scalability, reusability and usability) we first need to take a more detailed look at each of them separately.

Public API layer

The API is a very thin public facing layer that the developers interact with. It has a publicly available initialiser that defines what data the component displays. This `public init` should only accept content related raw data (`String`, `Decimal`, `Bool`, etc) specific to the instance that is being initialised. Other non content related data, such as the style, content size, disabled state, etc should be injected through the `Environment` via modifiers. The only `public` parts of this layer are the `init` method and the `body` property that renders the view, everything else should remain `private`.

The responsibility of the layer is to aggregate all the information that is essential to display by the component (content, internal states, environment, etc). Then it creates a Style Configuration from the collected data, and pass it to the Style to render the layout. Since assembling the layout or applying any visual styles is **not** considered as the responsibility of this layer, the `body` property is fairly simple as well.

```
public struct Select {
    public init(…) { … }

    public var body: some View {
        style.makeBody(configuration)
    }
}
```

Style Configuration

The Style Configuration is the glue between the API and the Style implementation. Its key responsibility is to prepare the data as renderable views for the Style. This way the Style ~~don't need to~~ is not able to manipulate the underlaying data and retains the purity of its purpose.

Every Configuration should be component specific without reusing other components' Configurations directly. If we were to reuse Configurations across components, it would introduce direct dependencies between those components. Tightly coupling them together would cause maintainability issues when any of those component changes. However, having fully dedicated Configurations doesn't mean they can't share an underlying view when it makes sense.

Another important role of the configuration is to handle access control. When all the layers are declared *publicly** we can still prevent developers from implementing custom styles by declaring the properties of the configuration with `internal` access level. That way, only the Styles implemented inside of Neptune can access them, blocking developers to add custom implementations outside of the system.

** The Style and Configuration don't necessarily need to be declared `public`, they can remain `internal`. However, there are cases when we support different styles (e.g. regular vs compact money input style) which means that both the style and the configuration needs to be `public`. In that case, the access modifier of the Configuration's property the ones one in charge to prevent custom style implementations.*

Style

The Style of the component is defined as a `protocol` which allow us to implement different visuals for a single component and substitute them with each other whenever we need it. Having the possibility doesn't mean we're gonna end up with endless custom implementations outside of the system. We can control that by either declaring the whole Style or the properties of the Configuration internally.

The Style's single requirement (and responsibility) is to implement a method that renders the layout and appearances based on the Style Configuration.

```
protocol SelectStyle {
    typealias Configuration = SelectStyleConfiguration
    associatedtype Body: View

    func makeBody(configuration: Configuration) -> Body
}
```

We can spot another influence of UIKit on how we injected the Styles into our components. Until now, we simply passed in the Style as a parameter in the `init` method of the API. A better way would be to use the `Environment` for this purpose. First of all, it helps the API to be agnostic of the visuals. This means we only pass in data through the `init` method that actually identifies the instance. By not coupling the Style to specific instances we make it more evident that the content and its visual appearance are independent from each other. Second of all, it lets us set a Style for all the components inside of a view hierarchy without setting it to all, one by one. It's also in line with the default styling behaviour of SwiftUI, which the developers are already familiar with.

Supporting types

In order to integrate this architecture seamlessly into SwiftUI we need to add some additional types and extensions under the hood.

Style wrapper

SwiftUI made it really simple and convenient to handle styles within a view hierarchy. We can invoke any styling method for any component on any kind of view anywhere in the view hierarchy and it will propagate downstream to apply it on all the corresponding children. Luckily this mechanism is *easily* available for us as well, by using the `Environment`.

However, the strongly typed nature of Swift and the associated body type of the Style make it impossible to inject it into the `Environment` as is. In order to do so the body type needs to be erased. One way would be to declare the Style protocol's `makeBody` method to return an `AnyView`, eliminating the generic type, but then we had to do it for every style implementation manually. Instead, we define a wrapper object that does this for us.

```
struct Select {
    …
    private struct Style {
        private var _makeBody: (Configuration) -> AnyView

        init<S: SelectStyle>(_ style: S) {
            _makeBody = { configuration in
                AnyView(style.makeBody(configuration: configuration))
            }
        }

        func makeBody(configuration: Configuration) -> some View {
            _makeBody(configuration)
        }
    }
    …
}
```

Injecting Style to the Environment

To follow the convenience of SwiftUI, we define the styling method as an extension on the `View` protocol. That allow us to apply the Style anywhere in the view hierarchy, styling multiple elements with a single call, instead of having to set it one by one on each children. This method wraps the actual Style implementation into the private type erased one and injects it into the `Environment`.

In order to inject our own objects into the environment we also have to declare our custom keys for the associated Style type and set a default value. Using a dedicated styling method lets us declare these keys and default values `private`, preventing developers to outsmart the system and inject custom implementations at will for the same key.

```
public extension View {
    func selectStyle<S: SelectStyle>(_ style: S) -> some View {
        environment(\.selectStyle, Style(style))
    }
}

private struct SelectStyleEnvironmentKey: EnvironmentKey {
    static let defaultValue = Style(BorderedSelectStyle())
}

private extension EnvironmentValues {
    var selectStyle: Style {
        get { self[SelectStyleEnvironmentKey.self] }
        set { self[SelectStyleEnvironmentKey.self] = newValue }
    }
}
```

Obscuring types

As it was mentioned earlier that Style Configuration hides the underlying data from the Style. It does so by providing renderable views that encapsulate the data with functionality without assembling the different parts together into a cohesive layout. Since it's the Configuration's responsibility to instantiate these views with the containing data, they're most commonly be declared as embedded types with `private` initialiser.

However, in some cases different components' Configuration might share some of their obfuscating views, but only when their use-cases are identical. The updated Input components can all share the implementation of `Label` and `Information` (see `InputLabel` example below) since they're fulfilling the exact same purpose in those components.

These obscuring views are never optional, so let's take a look at what happens when the underlying data is. How the Style is going to know whether it should present a view or not, when it doesn't have access to the underlying value, but only to a non-optional view? Luckily SwiftUI's `EmptyView` provides the right behaviour for us. When you define conditional views without a "default" branch, the system will implicitly add an `EmptyView` as a fallback. (See `Thumbnail` example below). The good thing about `EmptyView`, is that no matter what modifier you apply to it, it won't have any effect on it, hence it won't affect the layout either. So you can confidently add borders or padding to any obfuscated views, they're only going to appear if the underlying value is not an `EmptyView`.

StyleConfiguration code example

```
/// Embedded declaration
extension SelectStyleConfiguration {
    struct Thumbnail: View {
        private let image: Image?

        fileprivate init(_ image: Image?) {
            self.image = image
        }
```

```
        var body: some View {
            if let image = image {
                image
            }
            // an `EmptyView` in the `else` branch is added implicitly
        }
    }
}

/// Shared declaration
struct InputLabel: View {
    private let value: String?

    init(_ label: String?, optional: Bool = false) {
        value = label?.appending(optional ? " (Optional)" : "")
    }

    var body: some View {
        if let value = value {
            Text(value)
        }
    }
}
```

User interactions

How we handle user interactions has also have an effect on the components' public API as well as on the layers of the architecture.

States vs events

The way we implement user interactions is one of the most tangible evidence of the influence UIKit is having on our SwiftUI APIs. UIKit is driven by events which, when it comes to interaction handling, usually represented by closures passed in through the API. And that's the only concern of the API, it doesn't care about the outcome of the side effect, it only cares about the execution. Until now, we followed this same approach with our SwiftUI APIs, expecting closures to handle interactions.

However, the declarative nature of SwiftUI follows a more semantic approach, where the purpose and so the outcome of the interactions are well defined. This means, the API expects an external state in the form of a `Binding`, which is going to be altered by the interaction. So API basically defines the outcome, but not how the change is going to be executed.

The `MoneyInput` component for example, has a `currencySelectButtonTouched: () -> Void` closure parameter on its current public API. This event driven approach basically tells the consumer of the component that the currency button was touched and they can do whatever next. When we shift our paradigm to the state driven approach, the above closure parameter turns into an externally stored state, that the component will change upon the tap event. For example `showCurrencySelector: Binding<Bool>`. The win here is not that the developers are restricted to pass in any side effect. Because, let's be honest, they can still do anything when the state changes. But this way the API gives clear guidance on what is supposed to happen next and how the component was design to be used.

Separation of concerns

One grey area of the separation of concerns is handling user interactions. We discussed that the Configuration is the one who converts the underlying data (including content and states) into **functioning** views. That means, ideally those views already handle any kind of interaction they need to, which actually comes in handy, especially when they are shared between components. Not assigning that responsibility to the Style removes unnecessary code duplications in case of having multiple Style implementation.

Although, every so often, the layout might have an affect on the interaction, for example it can increase the tappable area of an element to make it more accessible for the users. Since the order of the modifiers applied to a view matter in SwiftUI, assigning the tap handler inside of an obfuscated view of the Configuration then enlarging the size of the view in the Style will result bigger space consumption on screen, yet the tappable area would remain the same.

In such scenarios, the actual interaction handling can be implemented in the Style, but the underlying effect of the interaction should still be hidden by Configuration. Instead of revealing an internal or externally bonded state, the Configuration will only reveal an action to call upon the interaction.

A good example of this from the SwiftUI library is the `PrimitiveButtonStyleConfiguration` which provides a `trigger()` method for the Style to invoke when the interaction happens. Another example from Neptune is Select (a.k.a. Dropdown) where not only the value label, but the thumbnail, chevron the surrounding padding and border needs to be tappable as well. Since that layout is assembled by the Style, the Configuration can't apply the interaction handling in the subviews individually, but the state change triggered by the tap should be buried inside.

Accessibility

The two main accessibility technologies we aim to support are DynamicType and VoiceOver. Handling DynamicType is fairly simple. It only affects the sizes of elements and their relative arrangements in the layout. So it's quite straightforward that the Style is the one in charge here as it's responsible for the layout.

However, VoiceOver needs to express not just the readable content, but all the implicit informations that are conveyed by different text styles, colours, sizes and layout arrangements. We don't have a single layer that unites all those responsibilities any more. Luckily, from a coding perspective, VoiceOver is not at all different from the screen of the device. Both of them are just user interfaces of different kinds, helping consumers to use the app. One uses visuals, processed by eyes, the other uses sounds, processed by ears. Since we already split the responsibilities of our layers for the visual user interface, we can just apply the same separation for VoiceOver as well. That means, the information represented by the content, such as `accessibilityLabel` or `accessibilityValue`, is built into the obscured views of the Configuration Whereas, the information represented by visual styles and the layout, such as the `accessibilityHidden`, `sortPriority` (a.k.a. order of the elements) or how to handle children elements, goes into the Style implementation.

Case study with Inputs

For practical application please read through the related case study article: Case Study: Architecture of SwiftUI components

Further improvements

As always we do have some exceptions when we need to be a bit lenient with the responsibilities of the layers. It could either be a conscious design decision or a limitation rooted somewhere in our system.

Styling texts

Just like all the styles of our component, or any styling in SwiftUI for that matter, our `Emphasis` and `TextStyle` could/should use also use `Environment` to be propagated through the view hierarchy. That would mean, we'd to implement our own `Text`(maybe `StyledText`) and `MarkupText` SwiftUI components. Besides being consistent with the styling approach, that would resolve one of the biggest issue we have currently with our `TextStyle`. When we apply it to the built in SwiftUI `Text` type, the resulting type won't be `Text` anymore. This means we lose some powerful features of `Text` like concatenating. With our own wrapping `StyledText` type we would be able to overcome these limitations we built into our token system.