Case Study: Architecture of SwiftUI components

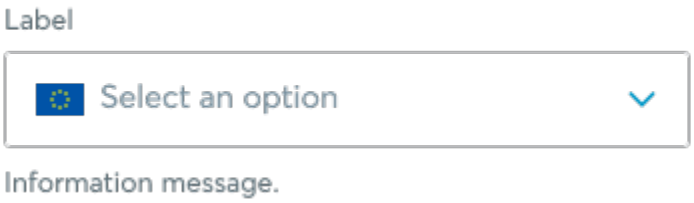
This is a complementer article for the Architecture of SwiftUI components. If you haven't yet read that, please do so before proceeding.

While the previous article discusses the *whats* and *whys,* this one focuses on the *hows.* It aims to reveal insights on *how* this approach helps us achieve our main goals (scalability, maintainability and usability), as well as to provide a practical guidance on *how* to apply it to our components' implementations.

- Implementing a component
  - Public API
  - Style Configuration
  - Style
- Maintainability
- Scalability
  - API
  - Style Configuration
- Usability
- Related Pull Requests

Implementing a component

The following screenshot shows the neutral empty state of our Select component. We're gonna use this as an example on how to apply the layers to our implementation.



Public API

The API layer is quite straightforward, its main concerns is to collect all the content that needs to be displayed and utilise the `Configuration` and `Style` to render the final view.

The contents we need to instantiate a Select are: label, thumbnail image (optional), placeholder, selected value (if there's any), the nature of the message (a.k.a. emphasis), a bool flag to indicate whether this input field is optional or not, and an externally held state that represents whether the option list is displayed or not. All these information are passed in as data types, such as `String`, `Bool`, `MarkupTextModel` etc.

Other than the content, the Style is also required to render the layout, but it's injected through the `Environment`.

API layer of Select

```
public struct Select: View {
    @Environment(\.selectStyle)
    private var style

    private let label: String?
    private let placeholder: String
    private let thumbnail: Image?
    private let value: String?
    private let information: MarkupTextModel?
    private let emphasis: InputEmphasis
    private let presentValues: Binding<Bool>?
    private let optional: Bool

    public init(
        label: String? = nil,
        placeholder: String,
        value: String?,
        thumbnail: Image?,
        information: MarkupTextModel? = nil,
        emphasis: InputEmphasis = .neutral,
        presentValues: Binding<Bool>?,
        optional: Bool = false
    ) { ... }

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

    private var configuration: Configuration {
        Configuration(
            label: label,
            placeholder: placeholder,
            value: value,
            thumbnail: thumbnail,
            information: information,
            emphasis: emphasis,
            presentValues: presentValues,
            optional: optional
        )
    }
}
```

Note that only the type declaration, the `init` and the `body` property are `public`, everything else is `private`. As we discussed, we still need a publicly available method and some hidden helper types that make it possible to inject the Style, while retaining the control to prevent unwanted injections.

Style handling supporting types

```
// MARK: - Style
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)
    }
}

// MARK: - Environment
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 }
    }
}
```

Style Configuration

As it was seen in the API implementation, the Configuration is created by the API. Select is a fairly simple component, where the API doesn't contain any internally maintained state. But in cases when a component has internal state(s), for example focus state, they're also going to be passed to the Configuration.

Configuration layer of Select

```
struct SelectStyleConfiguration {
    let label: InputLabel
    let value: Value
    let thumbnail: Thumbnail
    let information: InputInformation

    let emphasis: InputEmphasis
    let focused: Bool
```

```
    let tapHandler: Action?

    init(
        label: String?,
        placeholder: String,
        value: String?,
        thumbnail: Image?,
        information: MarkupTextModel?,
        emphasis: InputEmphasis,
        presentValues: Binding<Bool>?,
        optional: Bool
    ) {
        self.label = Label(label, optional: optional)
        self.value = Value(
            content: value ?? placeholder,
            kind: (value?.isEmpty ?? true) ? .placeholder : .value
        )
        self.thumbnail = Thumbnail(thumbnail)
        self.information = Information(information)
        self.emphasis = emphasis
        self.focused = presentValues?.wrappedValue ?? false
        if let presentValues = presentValues {
            tapHandler = Action(title: accessibilityActionTitle,
handler: { _ in
                presentValues.wrappedValue = true
            })
        } else {
            tapHandler = nil
        }
    }
}

extension SelectStyleConfiguration {
    struct Thumbnail: View {
        private let image: Image?
        init(_ image: Image?) {
            self.image = image
        }

        var body: some View {
            if let image = image {
                image
            }
        }
    }

    ...

    struct Value: View {
        enum Kind {
```

```
            case placeholder
            case value
        }

        let content: String
        let kind: Kind

        init(content: String, kind: Kind) {
            self.content = content
            self.kind = kind
        }

        var body: some View {
            Text(content)
        }
    }
}
```

The views representing the underlying data in the Configuration are completely agnostic of the style that's going to be applied on them. However, the `Value` view has a `Kind` property which gonna indicate the nature of its content to the Style whether it's an actual value or just a placeholder. This way the Style can apply the proper appearances to it. This might be interpreted as the Configuration is aware of the actual visual design, assuming that none of the styles would render both the placeholder and the value at the same time. But that is not an assumption of the visual design, rather it's a deliberate constraint based on the purpose of the content. As the name suggest, placeholder is only needed as a substitute when no value is present and no style should be able to render both simultaneously.

We could have added two separate view types, e.g. `Placeholder` and `Value`. Then we also have to add an `isEmpty` property to the latter one, so the Style knows which one to display. Although, this would add extra complexity to the Style and leave room for error such as displaying both of them at the same time.

In the case of Select the user interaction depends on the layout, so it's going to be handled by the Style. However, the Style's only concern is to execute what the Configuration provides, without being aware of the underlying logic. So we expose an `Action` which includes the accessibility information as well.

Style

The Style protocol is looks almost exactly the same for every component. The only difference is the type of the Configuration.

⌄ Style protocol declaration

```
import Foundation
import SwiftUI

protocol SelectStyle {
    typealias Configuration = SelectStyleConfiguration
    associatedtype Body: View

    func makeBody(configuration: Configuration) -> Body
}

// MARK: - Built-in Text Input Styles

extension SelectStyle where Self == BorderedSelectStyle {
    static var bordered: BorderedSelectStyle { BorderedSelectStyle() }
}
```

The more interesting part here is the actual implementation of a specific Style that puts all the pieces together. A Style itself is not a `View`, that is implemented privately in the same file. Another thing to notice is that the `chevron` wasn't included in the Configuration, instead it's hard coded into the Style. That chevron is just a visual clue suggesting how the users can interact with the component. It's independent from the content and fully owned by this specific Style. There could be another style that has an arrow instead or doesn't display anything at all.

BorderedSelectStyle implementation

```
struct BorderedSelectStyle: SelectStyle {
    func makeBody(configuration: Configuration) -> some View {
        BorderedSelect(configuration)
    }
}

private struct BorderedSelect: View {
    @Environment(\.theme) private var theme
    @Environment(\.sizeCategory) private var sizeCategory

    private struct Constants { ... }
    private var constants: Constants { ... }
    private var stationary: Bool { configuration.tapHandler == nil }
    private let configuration: SelectStyleConfiguration

    init(_ configuration: SelectStyleConfiguration) {
        self.configuration = configuration
    }

    var body: some View {
        VStack(alignment: .leading, spacing: constants.
verticalSpacing) {
            label
            SwiftUI.Button {
                configuration.tapHandler?()
            } label: {
```

```
                content
            }
            information
        }
        .disabled(stationary)
        .accessibilityElement(children: .ignore)
        .accessibility(addTraits: stationary ? .isStaticText : .
isButton)
    }

    // MARK: - Subviews
    private var label: some View { ... }
    private var content: some View { ... }
    private var value: some View {
        configuration.value
            .textStyle(configuration.value.kind == .value ? .value : .
body1)
            .frame(maxWidth: .infinity, alignment: .leading)
    }
    private var chevron: some View {
        Icons.Utility.Small.chevronDown
            .foregroundColor(theme.color.content.accent.normal.color)
            .frame(
                width: constants.chevronSize.width,
                height: constants.chevronSize.height
            )
            .accessibility(hidden: true)
    }
    private var thumbnail: some View { ... }
    private var information: some View { ... }

    // MARK: - Styling
    private var labelStyle: TextStyle {
        .body2.with {
            guard !configuration.focused else {
                return $0.semanticColor = \.accent
            }
            switch configuration.emphasis {
            case .neutral: $0.semanticColor = \.secondary
            case .warning: $0.semanticColor = \.warning
            case .negative: $0.semanticColor = \.negative
            case .positive: $0.semanticColor = \.positive
            }
        }
    }
    private var borderColor: KeyPath<SemanticColors.Interactive,
SwiftUI.Color> { ... }
    private var informationStyle: TextStyle { ... }
}
```

It's also important to notice that all the content related accessibility modifiers are driven by the Configuration, while all the layout related accessibility modifiers (e.g. how to handle child views) are fully controlled by the Style.

Maintainability

The code above might seem a bit overly complicated for such a simple component, but as you probably aware, bordered visual is a fresh new look of the Select component. It was introduced as part of a project with a much wider scope that aims to improve all of our Inputs. To keep the overall UX of our product consistent, we can't roll out the new visuals one by one. We need to introduce all the new input styles at once. That means the new layouts have to live next to their ancestors until each of them are done. With this layered architecture it's really easy for multiple styles to coexist without depending on each other. Yet the implementations stay clean with no complicated presentation logic inside of them.

The screenshot above is the old visual design for the Select component. In order to have both Styles in our codebase we just simply need to add another implementation into a new file. Once we finished with all the inputs, we can change the default value of the SelectStyle in the `Environment` and we're good to go. After we rolled the new version out and we're confident that it hasn't got any major issues, we can just simply remove the whole file that contains the implementation of the old layout.

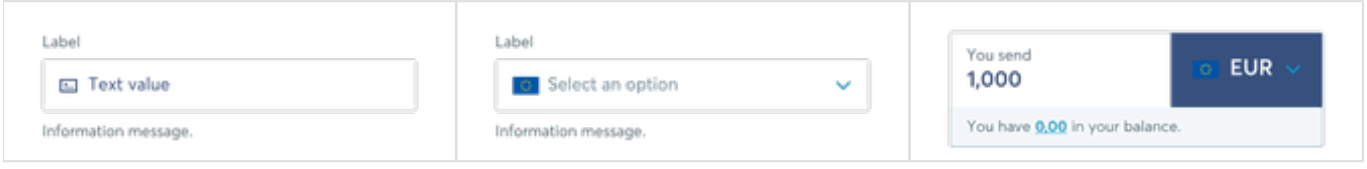
UnderlinedSelectStyle

```
struct UnderlinedSelectStyle: SelectStyle {
    func makeBody(configuration: Configuration) -> some View {
        UnderlinedSelect(configuration)
    }
}

private struct UnderlinedSelect: View { ... }

private struct SelectStyleEnvironmentKey: EnvironmentKey {
    /// To switch, we just simply need to replace the
    /// UnderlinedSelectStyle to the BorderedSelectStyle
    /// as the default value
    static let defaultValue = Style(UnderlinedSelectStyle())
}
```

Another massive maintenance improvement of this approach is that it lets us handle these changes more granularly. Until now, when we changed the visuals of a component, we had to change the whole project at once including all apps and frameworks. This was due to the combination of building un-versioned embedded frameworks and the single layered architecture of our components. Updating the whole project easily resulted in hundreds or thousands of file and line changes in a single pull request. That's not just a burden to overview but has a bigger risk of missing something that is broken.

Scalability

When we take a look at the Money Input component, we definitely find some similarities with Text Input and Select. Even though the visuals are slightly different, the functionality is basically the combination of the two. In the following section, we'll discuss how we can leverage the new architecture to reuse code without causing any trouble for future changes.



API

The API layer has no surprises, as always, it just collects the data and declares the internal states. It has no knowledge at all of what other components will be used under the hood. This prevents dependencies to be built into this layers.

MoneyInput API layer

```
public struct MoneyInput: View {

    @Environment(\.moneyInputStyle)
    private var style

    @State
    private var focused = false

    private var value: Binding<MoneyInputValue>

    private let placeholder: String?

    private let label: String

    private let emphasis: InputEmphasis

    private let currency: Currency

    private let presentCurrencies: Binding<Bool>?

    private let information: MarkupTextModel?

    private let formatter: MoneyInputFormatter

    private let textFieldAccessibilityLabel: String

    private let currencySelectAccessibilityLabel: String

    public init(
        value: Binding<MoneyInputValue>,
        placeholder: String? = nil,
        label: String = "",
        information: MarkupTextModel? = nil,
        currency: Currency,
        presentCurrencies: Binding<Bool>?,
        locale: Locale = MoneyFormatter.presentationLocale,
        emphasis: InputEmphasis = .neutral,
        textFieldAccessibilityLabel: String,
        currencySelectAccessibilityLabel: String
    ) { ... }

    public var body: some View { ... }
}
```

Style Configuration

The Configuration layer is the one containing all the little tricks that makes our components scalable.

Let's start with the obvious. Why isn't the Money Input's Style Configuration is composed from Text Input's and Select's Configuration? Firstly, the Configuration needs to be a dedicated for a specific component, to avoid dependencies. Secondly, it should contain renderable views and Configurations are not views. Lastly, the `label` and `information` would be ambiguous, since both Text Input's and Select's Configuration contains them. Hence the Style implementation wouldn't be sure which one to rely on.

Instead of sharing the configurations, we can either extract and share views that initially were implemented as a nested type of a Configuration, or we can reuse whole components as well.

```
public struct MoneyInputStyleConfiguration {
    /// Common views extracted from other's Configuration
    /// to be able to share tem internally inside Neptune
    public let label: InputLabel
    public let information: InputInformation
    public let textField: InputField

    /// The publicly available Select component as a whole
    public let currencySelect: Select

    /// Configuration's own nested type
    public let placeholder: Placeholder
    ...
}
```

In the example above, `InputLabel`, `InputInformation` and `InputField` are standalone internal view types providing functionality only without any styling.

The `InputLabel` wraps the logic to append "(Optional)" to its content when necessary. We don't implement this as a nested type in each of the Configurations individually, to avoid duplications. Instead we declare it in its own file and use across all the Inputs. The same stands for the `Input Information` where the embedded functionality is basically just to hide an optional information text inside of a conditional view.

Shared view implementations

```
import Foundation
import SwiftUI

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)
        }
    }
}
```

```
import Foundation
import SwiftUI

public struct InputInformation: View {
    @Environment(\.preferredMaxLayoutWidth)
    private var preferredMaxLayoutWidth

    @Environment(\.textStyle)
    private var textStyle

    private let information: MarkupTextModel?

    init(_ information: MarkupTextModel?) {
        self.information = information
    }

    public var body: some View {
        if let information = information, !information.text.isEmpty {
            MarkupLabel(viewModel: .markup(information), style:
textStyle)
                .preferredMaxLayoutWidth(preferredMaxLayoutWidth)
                .accessibilityElement(children: .ignore)
                .accessibility(addTraits: .isStaticText)
        }
    }
}
```

Although we could have, we didn't used the `TextInput` public component directly. If we did use that, we should have implemented a custom Style for the whole `TextInput` just to get rid of the label and information. Instead, we extracted its Configuration's nested `InputField` view type. It already implements all the functionality we need without all the unnecessary bits and styling that the whole `TextInput` contains.

This could lead to a question, why did we use the `Select` component instead one of its underlying nested view type? Well, we need most of its content, and it doesn't have a distinct part which we could easily extract. Also, the visuals are quite different when it's used publicly as a standalone component vs when it's used inside of `MoneyInput`. Thanks for the layered architecture, we can easily add a new internal Style, `CurrencySelectStyle` that we're going to use for `MoneyInput`.

CurrencySelectStyle implementation

```
struct CurrencySelectStyle: SelectStyle {
    struct FlagSize {
        let `default`: CGSize
        let accessible: CGSize

        static let regular = FlagSize(
            default: CGSize(width: 24, height: 16),
            accessible: CGSize(width: 36, height: 24)
        )
        static let compact = FlagSize(
            default: CGSize(width: 16, height: 10),
```

```
            accessible: CGSize(width: 24, height: 16)
        )
    }

    private let flagSize: FlagSize
    private let textStyle: TextStyle

    func makeBody(configuration: Configuration) -> some View {
        CurrencySelect(configuration: configuration, flagSize:
flagSize)
            .environment(\.textStyle, textStyle)
    }
}

private struct CurrencySelect: View {

    ... constants, environment, states, properties, etc ...

    private let configuration: SelectStyleConfiguration
    private let flagSize: FlagSize

    fileprivate init(configuration: SelectStyleConfiguration,
flagSize: FlagSize) {
        self.configuration = configuration
        self.flagSize = flagSize
    }

    var body: some View {
        SwiftUI.Button {
            configuration.tapHandler?.trigger()
        } label: {
            content
        }
        .backgroundColor(stationary ? \.screen : \.elevated)
        .disabled(stationary)
        .accessibilityElement(children: .ignore)
        .accessibility(addTraits: stationary ? .isStaticText : .
isButton)
    }

    // MARK: - Subviews
    ...
}

extension SelectStyle where Self == CurrencySelectStyle {
    static var regularCurrencySelect: CurrencySelectStyle {
CurrencySelectStyle(flagSize: .regular, textStyle: .title3) }
    static var compactCurrencySelect: CurrencySelectStyle {
CurrencySelectStyle(flagSize: .compact, textStyle: .title4) }
}
```

All the above already shows how flexible and scalable this architecture is. It allows us to share functionality while implementing different user interfaces. But the real power lies in its recursive nature. The views that are used by the Configurations are primitive single layered, single purpose unstyled views. But even without affecting their APIs, we can easily apply the same architecture on them. That way we could provide different built in styles for them and share them as well not just the content and functionality. Just imagine a style for the `InputLabel` where we don't communicate the optionality of the input via text, but instead we add an icon. We could implement `SpelledOutLabelStyle` and `IconLabelStyle` so the Inputs don't need to implement them in their Styles, they could just simply apply appropriate one. But again, those styles would be applied inside of the components' Style implementation, not in the Configuration.

Usability

Last but not least, we also improved convenience and usability for our developers. Money Input has two different styles, the `Regular` that we use in the main Wise app, and a `Compact` used in the Currency Converter app.

With our previous implementation, we set the style of a component via the `init` method. That means, when we need to use a Style other than the default, we had to specify it for each instance individually. In the CurrencyConverter app for example, we use the `Compact` Style of the Money Input. By using the `Environment`, we can inject the Style from the root of the calculator's screen once, and both of the inputs will be styled accordingly. This doesn't seem like a huge win first, but we could potentially inject the Currency Converter specific Styles for all the components in the root of the whole app. Overriding the default Styles in the root would affect every screen without having to deal with in place.

Styling multiple components

```
Vstack {
    /// Source input
    MoneyInput(...)

    /// TargetInput
    MoneyInput(...)
}
.moneyInputStyle(.compact)
```

And as it was mentioned earlier, this is how the standard SwiftUI components can be styled, se even new joiners will be familiar with it without having to learn any new concept.

Related Pull Requests

TBD