<epam>

SwiftUI Architectures overview

Illia Kucheriavyi

Sept 2020

Kharkiv, Ukraine

About me

- **7+ years** of experience in software engineering
- **3+ years** as a Team Lead/Solution Architect/Coordinator

- 6+ months of SwiftUI in production
- Hands-on since iOS 6
- Negotiate about mobile specifics and architecture with customers
- Perform Engineer/Lead/Solution Architect/Coordinator roles

Illia Kucheriavyi
Lead Software Engineer

Contents

Limitations

SwiftUI v1 engineering issues

Reactive Limitations

1. A lot of new property wrappers is carried by SwiftUI
    - Existing code should be modified to support SwiftUI
    - @ObservedObject, @EnvironmentObject cannot be optional, and they are class-only protocols
    - ViewModels should inherit from ObservableObject

2. Reactive way is more complex then imperative for:
    - Animations
    - Navigation

3. Reactive programming can only partially avoid

Design & Implementation Limitations

1. Complicated way of working:
    - Frames for example there is no easy way to do scroll-to-textfield when active, functionality
    - ScrollView offset and custom gestures is very complicated

2. Missed functionality:
    - No controller infrastructure, any View can trigger navigation
      means violation of Router, Coordinator logic
    - No UIResponder
    - No full screen popover
    - No multiline textField / textView
    - No activity indicator

3. Rendering HStack and VStack for 10 000 objects performance is very poor
   first rendering time measuring in seconds

Generics & Generic Protocols Limitations

1. View is a generic protocol: Views can be only structure
   final class is compillable but crashes, class is not compillable

2. Identifiable is a generic protocol:
   - Abstraction level of your models should be generic protocols, means
     all abstraction level is generic protocols
     OR
   - base classes and inheritance instead of protocols

3. Some SDK Views are generic structs
   - Generics increasing entry threshold for new engineers
   - Need to use type-erased AnyView

Benchmark Task

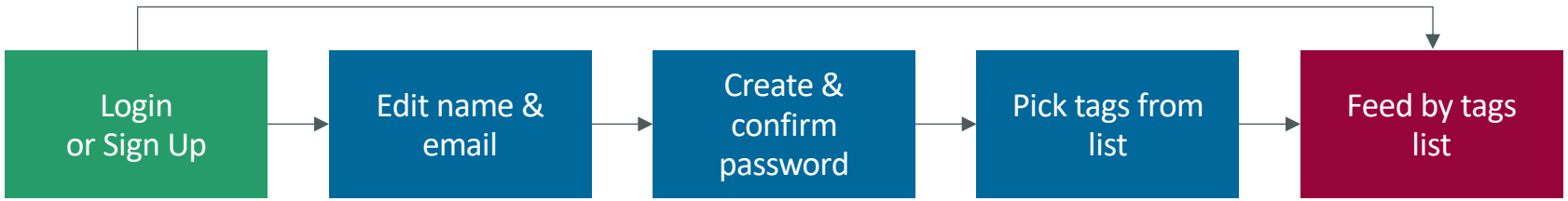
Typical business app to measure and metrics

Requirements

1. To measure difference between approaches lets create a typical app/flow we are implementing in everyday life:
    - 3+ screens to navigate
    - Network
    - List Layout
    - Component layout
    - Navigation Bar
    - Validation
    - Unit tests

2. So for example let’s create a simple ***login/sign-up flow and list viewing***

Architecture diagram (Logic Modules)



Metrics used

- Files count
- Declarations count
- Lines of code (LOCs)

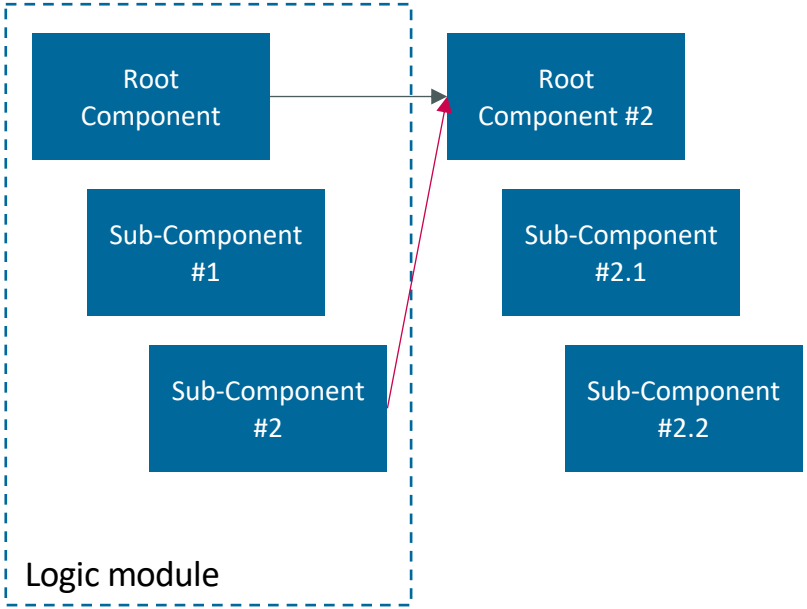
Common elements

- InputField.swift
  default designed field to handle unsecure and secure input
- ErrorText.swift
  error message with some design
- LoadingView.swift
  default view representing loading state
- Mocks.swift
  network interaction will be mocked with timer to simulate network connectivity, so need to keep responses same for all architectures

Component Architecture

Only Views & Models

Typical Component logic module



- Component is functional entity
  - has input arguments
  - internal logic
  - output is mostly UI

- Logic Module consist from 1+ component
- No restrictions for business logic separation between components
- Major part of components it's a Views
- Routing can be performed at any level
- Changes to Sub-Component arguments requires changes to whole chain

Code Highlights: all-in-one State

```
struct LoginView: View {

    @State private var email = ""
    @State private var password = ""

    @State private var isLoading = false
    @State private var error = ""

    @State private var navigatingSignUp = false
    @State private var navigatingHome = false

    ...

    var body: some View {
        ...
    }

    ...
}
```

Code Highlights: all-in-one Navigation

```
struct LoginView: View {
    ...

     var body: some View {
        VStack(spacing: spacing) {

            ErrorText(error: error)
                .padding(.vertical, spacing)

            ... (some UI)

            NavigationLink(destination: PostsView(tags: ["Login", "Tag", "1"],
                           goingLogin: $navigatingHome), isActive: $navigatingHome) {
                EmptyView()
            }
                .isDetailLink(false)

            NavigationLink(destination: EditBasicInfoView(), isActive: $navigatingSignUp) {
                EmptyView()
            }
        }
            .navigationBarTitle("Login", displayMode: .inline)
    }

    ...
}
```

Code Highlights: all-in-one UI, Action handlers, Logic

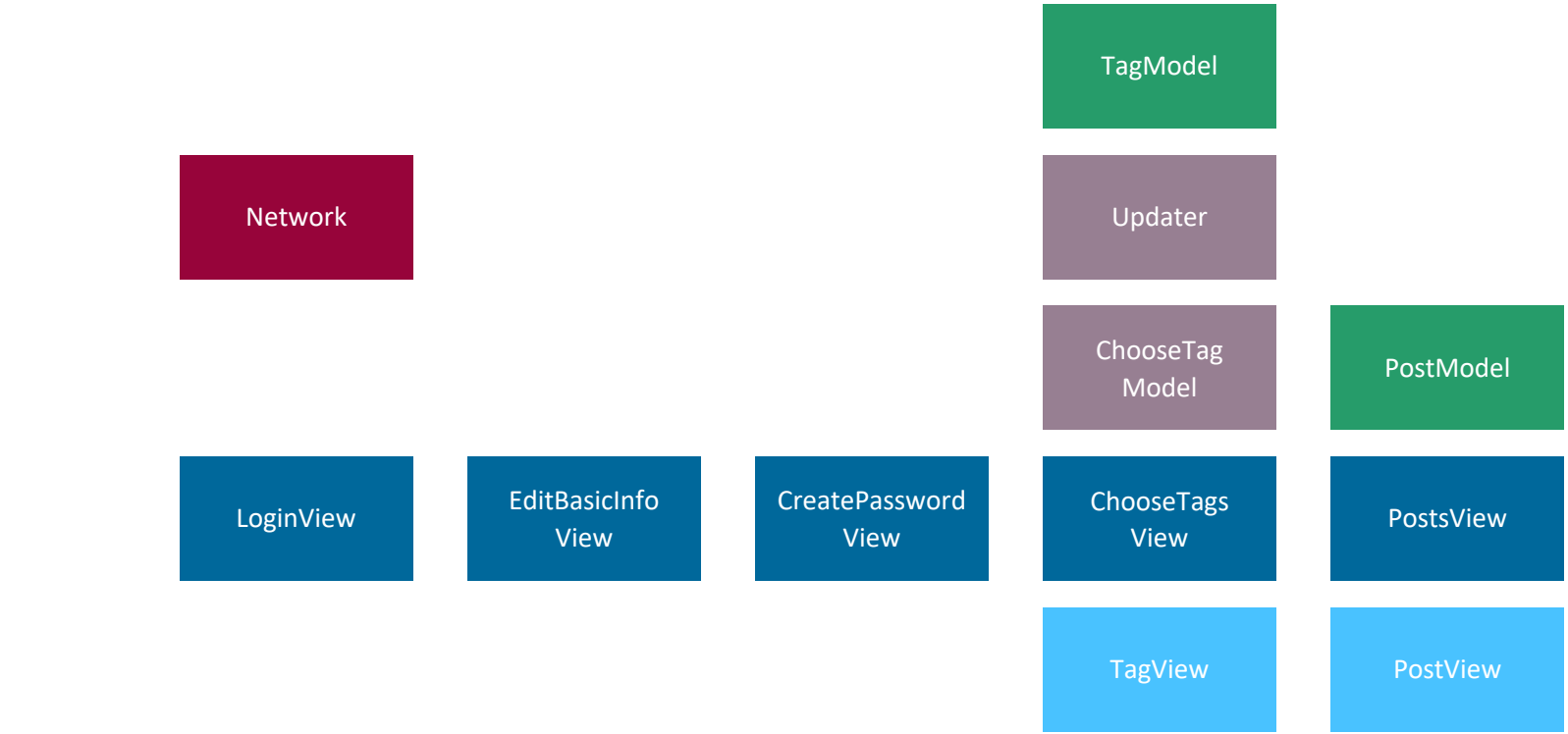
```
struct LoginView: View {
   ...

    private var defaultView: some View {
        VStack(spacing: spacing) {
            InputField(placeholder: "Email",
                       text: $email)
            InputField(placeholder: "Password",
                       text: $password,
                       isSecureField: true)

            Button(action: self.loginButtonClicked) {
                Text("Login")
            }

            Button(action: self.signUpClicked) {
                Text("Sign Up")
            }
        }
    }

    /// Actions & Logic

    ...
}
```

Code Highlights: all-in-one

```
struct LoginView: View {

    @State private var email = ""
    @State private var password = ""

    @State private var isLoading = false
    @State private var error = ""

    @State private var navigatingSignUp = false
    @State private var navigatingHome = false

    ...

    var body: some View {
        VStack(spacing: spacing) {

            ErrorText(error: error)
                .padding(.vertical, spacing)

            if isLoading {
                loadingView
            } else {
                defaultView
            }

            NavigationLink(destination: PostsView(tags: ["Login", "Tag", "1"],
                           goingLogin: $navigatingHome), isActive: $navigatingHome) {
                EmptyView()
            }
                .isDetailLink(false)

            NavigationLink(destination: EditBasicInfoView(), isActive: $navigatingSignUp) {
                EmptyView()
            }
        }
            .navigationBarTitle("Login", displayMode: .inline)
    }
```

```
    private var defaultView: some View {
        VStack(spacing: spacing) {
            InputField(placeholder: "Email",
                       text: $email)
            InputField(placeholder: "Password",
                       text: $password,
                       isSecureField: true)

            Button(action: loginButtonClicked) {
                Text("Login")
            }

            Button(action: signUpClicked) {
                Text("Sign Up")
            }
        }
    }

    /// Actions

    ...

    /// Logic

    ...
}
```

Implementation Classes diagram



Code Metrics

130 Blank

50 Comment

449 Code

629 Total

LOCs

12

Declarations

(previews not counts)

11

Files

Benefits

1. **Low entry threshold** only basic SwiftUI knowledge needed
2. Easy to read as tutorial
3. **Low amount of LOCs**
4. Low amount of classes & files
5. Declarative navigation for each module
6. Legacy Network layer can be used without any modifications
7. Preview support impact is low

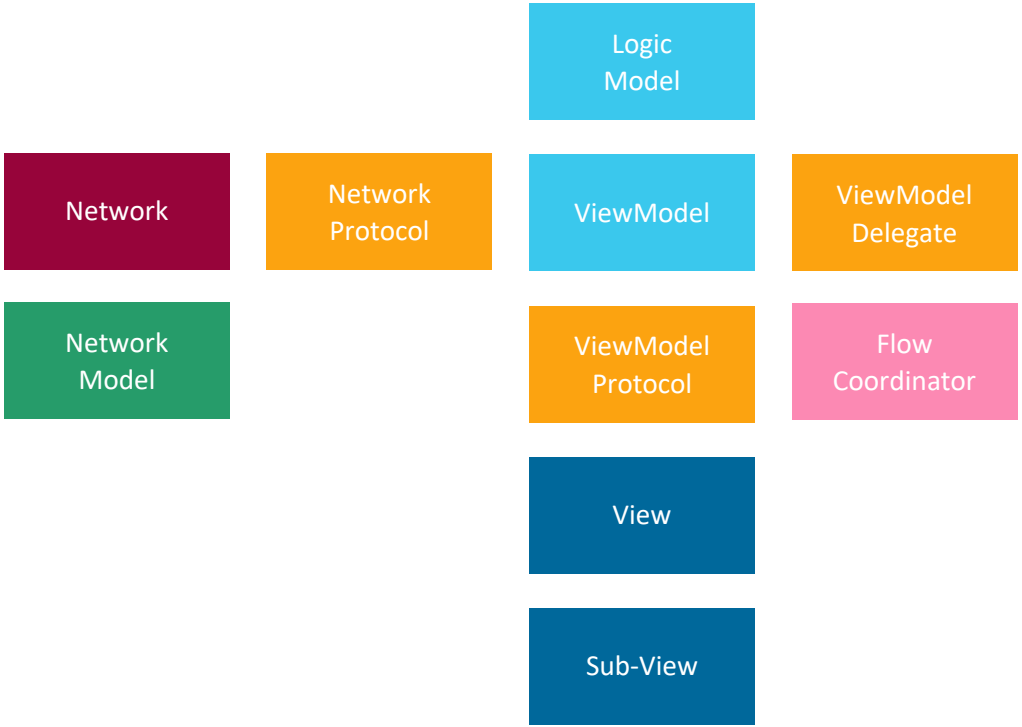
Drawbacks

1. Mixing logic for navigation, business rules and user interface, **SOLID violation**
2. Previews can affect production code, so additional preview support needed
3. In some logic cases to update View need to use hacks
4. **Unit Testing is complicated**, and its support affects production code

Protocol Architecture

MVVM-C with Protocol Binding

Typical MVVM-C module



- Logic Module consist from View, Model and ViewModel
- All dependencies are injected with protocols
- Navigation/Construction of other modules is performed on Coordinator level
- There is no connection from ViewModel to View, cause of SwiftUI limitations

Code Highlights: View part 1

```
struct LoginView<ViewModel: LoginViewModelProtocol>: View {
    @ObservedObject var model: ViewModel

    private let spacing: CGFloat = 16

    var body: some View {
        VStack(spacing: spacing) {

            ErrorText(error: model.error)
                .padding(.vertical, spacing)

            if model.isLoading {
                loadingView
            } else {
                defaultView
            }
        }
            .navigationBarTitle("Login", displayMode: .inline)
    }

    ...
}
```

- View is generic struct
- View contains only UI logic
- All actions are delegated to ViewModel

Code Highlights: View part 2

```
...

    private var defaultView: some View {
        VStack(spacing: spacing) {
            InputField(placeholder: "Email", text: $model.email)
            InputField(placeholder: "Password", text: $model.password, isSecureField: true)

            Button(action: model.loginButtonClicked) {
                Text("Login")
            }

            Button(action: model.signUpClicked) {
                Text("Sign Up")
            }
        }
    }
}
```

- View is generic struct
- View contains only UI logic
- All actions are delegated to ViewModel

Code Highlights: ViewModel part 1, Protocols Declaration

```
protocol LoginViewModelDelegate: class {
    func navigateHome()
    func navigateSignUp()
}

protocol LoginViewModelProtocol: ObservableObject {

    var delegate: LoginViewModelDelegate? { get }
    var network: NetworkProtocol { get }

    init(delegate: LoginViewModelDelegate)

    var email: String { get set }
    var password: String { get set }

    var isLoading: Bool { get set }
    var error: String { get set }

    func loginButtonClicked()
    func signUpClicked()
}
```

Code Highlights: ViewModel part 2

```
class LoginViewModel: LoginViewModelProtocol {

    @Published var email: String = ""
    @Published var password: String = ""
    ...

    @Published var network: NetworkProtocol = Network.shared

    private(set) weak var delegate: LoginViewModelDelegate?
        ...

    func loginButtonClicked()
        ...

    func signUpClicked()
        ...

    private func validate() -> String {
        if email.isEmpty {
            return "Email is empty"
        }
        if password.isEmpty {
            return "Password is empty"
        }

        return ""
    }
    ...
}
```

- All dependencies are injected through protocols
- ViewModel contains only business logic

Implementation Classes list

ChooseTagsViewModelDelegate
CreatePasswordViewModelDelegate
EditBasicInfoViewModelDelegate
LoginViewModelDelegate
PostsViewModelDelegate

ChooseTagsViewModelProtocol
CreatePasswordViewModelProtocol
EditBasicInfoViewModelProtocol
LoginViewModelProtocol
PostsViewModelProtocol

ChooseTagsViewModel
CreatePasswordViewModel
EditBasicInfoViewModel
LoginViewModel
PostsViewModel

ChooseTagsView
CreatePasswordView
EditBasicInfoView
LoginView
PostsView
PostView
TagView

NetworkProtocol
Network

ChooseTagModel
PostModel
TagModel

ChooseTagsViewModelDelegateMock
ChooseTagsViewModelMock
CreatePasswordViewModelDelegateMock
CreatePasswordViewModelMock
EditBasicInfoViewModelDelegateMock
EditBasicInfoViewModelMock
LoginViewModelDelegateMock
LoginViewModelMock
PostsViewModelDelegateMock
PostsViewModelMock
NetworkMock

ApplicationFlowCoordinator
DirectionViewModel
DirectionView
NodeView

Code Metrics

247 Blank

97 Comment

770 Code

1114 Total

LOCs

42

Declarations

(previews not counts)

23

Files

Benefits

1. **SOLIDity** (logic are separated by components):
    - Coordinator handles all the flow construction/navigation logic
    - Views is pure UI
    - ViewModels handles all business logic
    - Network and Network Models are fully separated

2. **Easy testing** on every level: Unit, Integration, UI
3. Feels like almost classic enterprise MVVM-C (easy onboarding for veterans)
4. Legacy Network layer needs only small refactoring to integrate (protocols declaration)
5. Previews cannot affect production logic

Drawbacks

1. **70% code and x3,5 declarations increase** compared with Component architecture
2. Preview support impact is high

   need to create a lot of preview mocks
3. Complicated router infrastructure

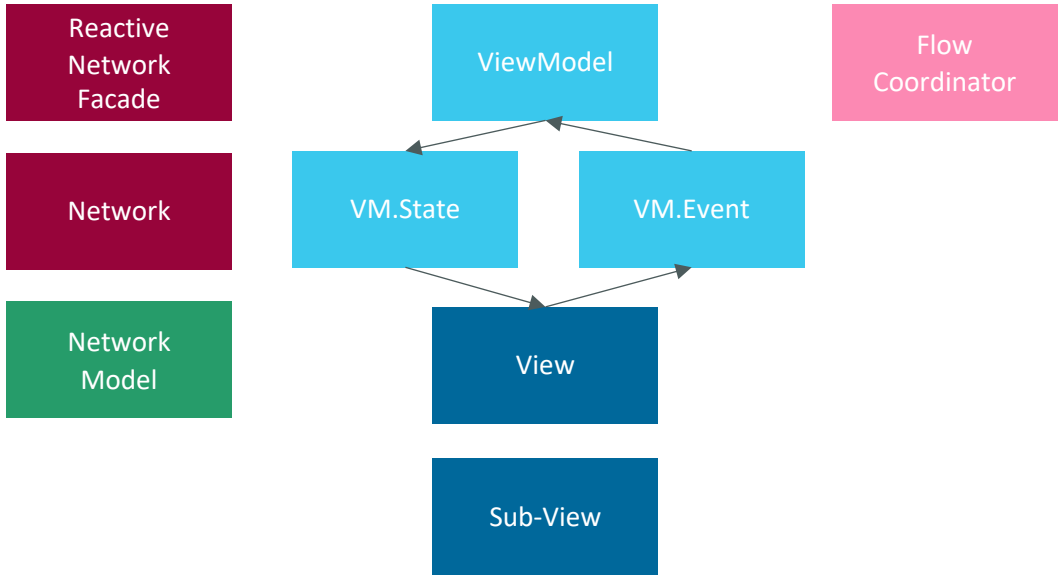
   additional classes and infrastructure should be created to support seamless routing
4. View is function of state, so ViewModel cannot have ViewInterface declared as separated protocol, only ViewState can be separated, but its still not protocol to protocol connection
5. **Higher entry threshold** to newcomers compared with Component architecture

   views transforms from struct to generic struct, protocols added, more code to read

Reactive Architecture

MVVM-C with Reactive Binding

Typical Reactive MVVM-C module



- Logic Module consist from View, Model and ViewModel
- View is a function of VM.State
- ViewModel is a function of VM.Event which is in results VM.State change
- Navigation/Construction of other modules is performed on Coordinator level

Code Highlights: View part 1

```
struct LoginView: View {
    @ObservedObject var viewModel: LoginViewModel

    @State private var email = ""
    @State private var password = ""

    var body: some View {
        VStack(spacing: .zero) {
            stateView
        }
            .onAppear(perform: { self.viewModel.send(event: .onAppear)})
            .navigationBarTitle("Login", displayMode: .inline)
    }

    private var stateView: some View {
        switch viewModel.state {
        case .idle:
            return defaultView.toAnyView()

        case .loading:
            return loadingView.toAnyView()

        case .error(let error):
            return errorView(error).toAnyView()

        case .navigating(let view):
            return activeRoute(to: view).toAnyView()
        }
    }
...
```

- View stores its internal state
- View is a big state view

Code Highlights: View part 2

```
...
    private func errorView(_ error: String) -> some View {
        VStack(spacing: spacing) {
            ErrorText(error: error)
                .padding(.vertical, spacing)

            defaultView
        }
    }

    private var defaultView: some View {
        VStack(spacing: spacing) {
            InputField(placeholder: "Email", text: $email)
            InputField(placeholder: "Password", text: $password,
                       isSecureField: true)

            Button(action: { self.viewModel.send(event: .loginClicked(self.email, self.password)) })
            {
                Text("Login")
            }

            Button(action: { self.viewModel.send(event: .signUpClicked) }) {
                Text("Sign Up")
            }
        }
    }
}
```

- View is sending events with arguments to ViewModel

Code Highlights: ViewModel part 1

```
final class LoginViewModel: ObservableObject {

    @Published private(set) var state = State.idle

    ...

    private var store = Set<AnyCancellable>()
    private let input = PassthroughSubject<Event, Never>()


    enum State {
        case idle
        case loading
        case error(String)
        case navigating(AnyView)
    }

    enum Event {
        case onAppear
        case loginClicked(String, String)
        case signUpClicked
    }

    ...
```

- ViewModel contains no View internal state
- State and Events are expressively declared
- Initial state is set

Code Highlights: ViewModel part 2

```
...

init(coordinator: ApplicationFlowCoordinator) {
        self.coordinator = coordinator

        let queue = DispatchQueue.main

        input
            .receive(on: queue)
            .filter { [weak self] _ in
                ...
            }
            .sink { [weak self] event in
                guard let strongSelf = self else { return }

                switch event {
                case .onAppear:
                    strongSelf.state = .idle

                case .signUpClicked:
                    strongSelf.navigateOrFail(route: .editBasicInfo)

                case .loginClicked(let email, let password):
                ...
                }
            }
            .store(in: &store)
    }
...
```

- Business logic is written in declarative-style
- Input is mapped to state

Implementation Classes list

PostModel
TagModel

Network
ReactiveNetwork
ReactiveNetworkMock
ReactiveNetworkFacade

PostsViewModel
LoginViewModel
EditBasicInfoViewModel
CreatePasswordViewModel
ChooseTagModel
ChooseTagsViewModel

ApplicationFlowCoordinator

ChooseTagsView
CreatePasswordView
EditBasicInfoView
LoginView
PostsView
PostView
TagView

Code Metrics

254 Blank

79 Comment

861 Code

1194 Total

LOCs

20

Declarations

(previews not counts)

18

Files

Benefits

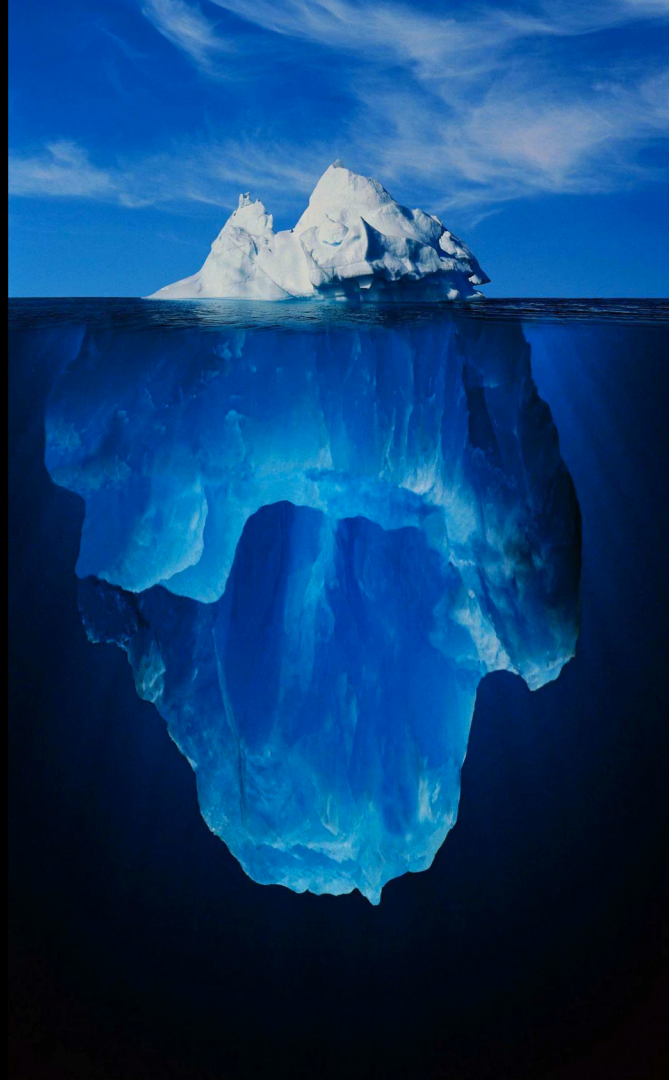
1. Logic is separated by components
2. Clear separation between possible View States and Events
3. Declarative style of business logic could be easier to read
4. **Lower declarations count** compared to Protocol approach
5. Easy testing for business logic at whole (Event to State mapping)
6. **Router infrastructure less complicated** compared to Protocol approach
7. **Previews can easily transform view with needed States to test**

Drawbacks

1. **70% code increase** compared with Component architecture
2. **Almost the same LOCs count as a Protocol approach**
3. Default pop-back behavior complicates ViewModel State management
4. Sometimes **State management could be tricky**, so requires more developer/user acceptance testing
5. **Higher entry threshold** to newcomers cause of Reactive and Functional Programming
6. Without sticking to Events and States could create a mess
7. Legacy Network layer needs an additional Facade created

Summary

Comparison and conclusion



Links

https://github.com/kojiba/SwiftUI-architectures/tree/master/Component

https://github.com/kojiba/SwiftUI-architectures/tree/master/Reactive

https://github.com/kojiba/SwiftUI-architectures/tree/master/Protocol

Architectures comparison

	Component	Protocol	Reactive
	CONSUMER ATTRIBUTES		
Entry threshold	Low	Medium	High
Code count (LOCs)	Low	High	High
Testability	Low	High	Medium
SOLIDity	Low	High	Medium
Preview support impact	Low	High	Low
Legacy Network Support	High	Medium	Medium
Side effects	Reload	No	State management errors

Q&A

Thank you