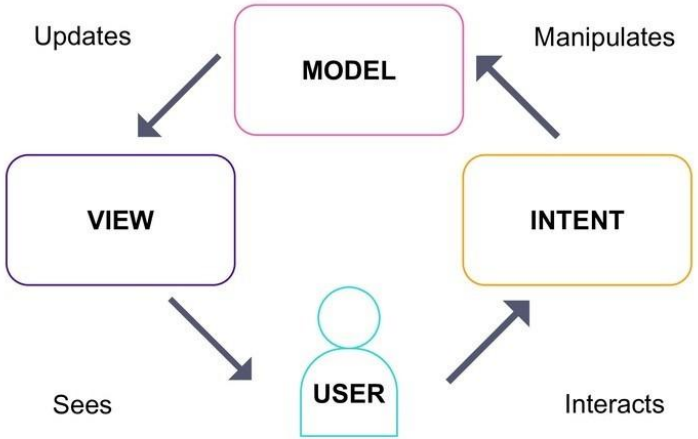
MVI ANDROID

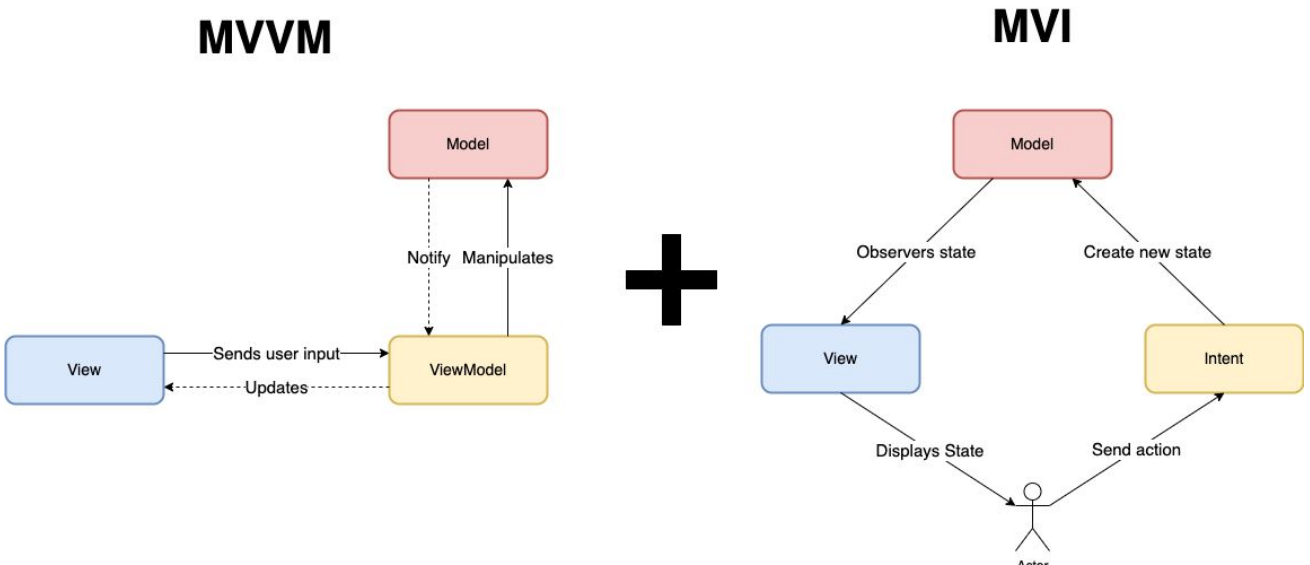
WHAT IS MVI?

The meaning of MVI is Model-View-Intent where the Model consists of business logic that provides data to the View (UI) based on the requested Intent by the user. & Intents are the actions that will trigger based on user action.



Is MVI better than MVVM?

Each has its own strengths and weaknesses. MVVM shines with data binding and view state management, while MVI stands out with its unidirectional data flow and robust state management.



UI States in MVI:

- Unlike MVVM, MVI employs screen States. Instead of having separate states for each changing value, there’s a single State representing the entire screen. This state is usually a data class that combines all the values subject to change on that screen.
- When updating the screen State, the common practice is to use the `copy` function, modifying the fields that have changed. This enhances code readability, as only one State is needed for the whole screen.

States of MVI

1. Model: The Model represents the application state and data. It encompasses all the business logic and manages the state changes. The state is immutable and can only be modified through intents.

2. View: The View layer is responsible for rendering the UI and displaying the current state to the user. It is a passive component that listens for user interactions and translates them into intents.

3. Intent: The Intent component captures the user's actions or events that trigger state changes. It is responsible for processing user inputs, such as button clicks or text entry, and generating corresponding intents to modify the state.

Flow of MVI

The MVI architecture follows a unidirectional data flow, which ensures predictability and consistency in the application's behavior. Let's walk through the flow of data in MVI:

1. The View layer receives user interactions and converts them into intents, which are dispatched to the Model layer.

2. The Model processes the intent and generates a new state based on the current state and the intent received.

3. The new state is propagated back to the View layer, which updates the UI accordingly to reflect the changes.

4. The updated UI presents the new state to the user, and the cycle continues as the user interacts further.

Advantages

- Readability: MVI can enhance code readability, especially when dealing with screen States. It provides a clear overview of potential changes on the screen.
- Integration with Compose: MVI integrates seamlessly with Jetpack Compose, simplifying the propagation of events from child composable.

Separation of Concerns: MVI promotes a clear separation between business logic (Model) and UI rendering (View). This separation enhances code organization and maintainability, allowing developers to work on individual components without affecting others.

Testability: MVI facilitates unit testing by making the Model layer completely decoupled from the Android framework. Since the state changes are driven by intents, it becomes easier to write test cases that cover different application scenarios and validate the expected outcomes.

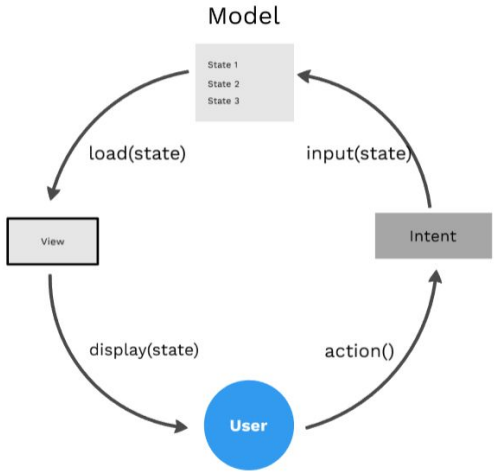
Immutable State: The state in MVI is immutable, meaning it cannot be modified directly. This approach guarantees that the state remains consistent and reduces the chances of introducing bugs due to unexpected state mutations.

Reactive Programming: MVI leverages reactive programming principles, often with libraries like RxJava or Kotlin Flow, to handle asynchronous operations and stream data between components. This enables developers to handle complex UI scenarios with ease and ensures a responsive user experience.

Error Handling: With MVI, error handling becomes more explicit. Any errors encountered during state modifications can be propagated to the View layer, allowing developers to handle them appropriately and display relevant error messages to the user.

The Model-View-Intent (MVI) architecture offers a powerful and effective approach to building Android applications. Its unidirectional data flow, separation of concerns, and emphasis on testability make it a preferred choice for developers aiming to create robust and maintainable apps.

By adopting MVI, developers can achieve better code organization, improved UI responsiveness, and enhanced test coverage, ultimately leading to a superior user experience. As the Android app development landscape continues to evolve, mastering the MVI architecture becomes a valuable skill for any Android developer.



MVI cyclic flow representation

Disadvantages

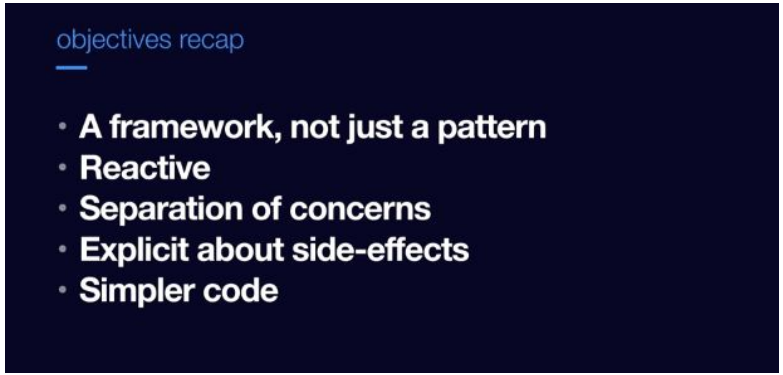
- Race Conditions: MVI is more susceptible to race conditions compared to MVVM. The `copy` function, used for updating the State, lacks thread safety. It's crucial to take precautions to prevent race conditions, particularly in a Compose environment.
- Process Death Handling: Managing process death in MVI can be challenging. Restoring the entire screen State might not always be ideal, leading to added complexities and workarounds.
- UI Re-rendering: Any alteration in the screen State triggers a complete UI re-render. This may be inefficient, especially in scenarios like typing in a text field in Jetpack Compose, where every character update causes a full UI re-render.

What is Mobius?

Mobius is a functional reactive framework for managing state evolution and side-effects, with add-ons for connecting to Android UIs and RxJava Observables. It emphasizes separation of concerns, testability, and isolating stateful parts of the code.

It uses Spotify's Mobius library. It as a framework to manage State updates, along with side-effects (asynchronous, non-deterministic behaviors) that may result in State updates.

To use Mobius to power your front-end, you will likely want to extend MobiusViewModel. You will notice that to do so, you have to define certain type parameters and class properties.



comparing to mvi

- Events instead of intent
- Less Rx intense
- MVI: intent -> action / state change
- Mobius: event -> state change -> effect

What is an Action?

An Action is the primary input type into the Mobius. Actions can represent events that are generated from outside of our Mobius, such as:

- User touch events (tapping a button, dragging across a screen)
- A network response
- A database/dao update

If we were building a simple app where you can increment and decrement an integer, we might model our Actions like this:

```
sealed class Action {
    object Increment : Action()
    object Decrement : Action()
}
```

How do Actions get sent to the Mobius?

Via the `actionRelay`. `MobiusViewModel.action(action: Action)` is provided as a convenience method.

What do Actions do after they are sent?

Actions are consumed by the Mobius and used to update the **State** and generate **Effect**s.

What is State

The State is an immutable data structure that holds the data needed to render the UI that the Mobius supports. It is literally the "state" of the screen or component being displayed/powered.

If we were building a simple app where you can increment and decrement an integer, the State would look like this:

```
data class State(
    val currentInt: Integer = 0
)
```

How does the front end consume the State?

Mobius provides an `Observable` stream that allows the front end to consume new instances of the State as they are created.

Why is the State immutable?

The point of a framework like Mobius is to ensure that the front-end (Activity, Fragment, or ViewModel) is updated in a predictable, managed way. By making the State immutable, we ensure that updates are only generated in a particular way.

So, how do State updates get created?

The Updater!

What is an Updater

The Updater is a pure function, which takes the current State and a received Action in order to return an instance of `Next`. We will focus on `Next` containing the updated `State`. The type signature looks like this:

```
typealias Updater<State, Action, Effect> = (currentState: State, action: Action) -> Next.State
```

If we were building a simple app where you can increment and decrement an integer, as we have been defining, the Updater might look like this:

```
object CounterUpdater : Updater<State, Action, Effect> {
    override fun invoke(currentState: State, action: Action) {
        val newState = when (action) {
            Action.Increment -> currentState.copy(currentInt = currentState.currentInt + 1)
            Action.Decrement -> currentState.copy(currentInt = currentState.currentInt - 1)
        }
        return Next.State(newState)
    }
}
```

Why is it a pure function?

A pure function is one that for a given input always has the same output.

It is important for our Updater to adhere to this principle because it ensures that State updates are predictable and easily testable.

```
class CounterUpdaterTest {

    private val initialState = State(currentInt =  5)

    @Test
    fun testIncrement() {
        val next = CounterUpdater.invoke(initialState, Action.Increment)
        assertEquals(6, next.state.currentInt)
    }

    @Test
    fun testDecrement() {
        val next = CounterUpdater.invoke(initialState, Action.Decrement)
        assertEquals(4, next.state.currentInt)
    }
}
```

What if I need to update my State based upon asynchronous, impure behavior?

You need to send `Effects` to the `Processor`!

What are Effects?

Effects are the input in to creating "side-effects." Typically, you can think of side-effects as asynchronous or non-deterministic behaviors that may or may not be used to update the State. Examples of common side-effects are:

- Kicking off a network request
- Fetching data from a database
- Initializing the observation of a database's changes
- Storing data to a database/shared preferences
- Sending an analytics event

For our Counter example, let's imagine that one of our product requirements is to send an Analytics event each time the user increments or decrements the counter. We could model the Effects like this:

```
sealed class Effect {
    sealed class Analytics : Effect() {
        object CounterIncrement : Analytics()
        object CounterDecrement : Analytics()
    }
}
```

What is a Processor?

The `Processor` is where we take Effects, execute side-effects, and are able to use their responses to optionally return more `Actions` to the `Updater`. The type signature looks like this:

```
typealias Processor<Effect, Action> = (effects: Observable<Effect>) -> Observable<out Action>
```

So, the `Processor` takes a stream of `Effects` and returns a stream of `Actions`.

In the case of our analytics example, the `Processor` could look something like this:

```
class CounterProcessor constructor(private val analytics: CounterAnalyticsService) : Processor<Effect, Action> {

    private val handleAnalyticsEffects: (Observable<Effect>): Observable<out Action> { effects ->
        effects.ofType<Effect.Analytics>()
            .doOnNext { effect ->
                when (effect) {
                    Effect.Analytics.CounterIncrement -> analytics.counterIncremented()
                    Effect.Analytics.CounterDecrement -> analytics.counterDecremented()
                }
            }
            .flatMap { Observable.empty<Action>() } // Analytics, as a side-effect, doesn't need to update the State
or anything, so we return nothing
    }

    override fun invoke(effectsOnBgThread: Observable<Effect>): Observable<out Action> {
        return effectsOnBgThread.publish { effects -> effects.compose(handleAnalyticsEffects) }
    }
}
```

Conclusion

MVI is basically your favorite architecture (MVVM or MVP) with state management where there is only one state for all the app layers, a single-source of truth. Although there is no only one best way to structure your code, it depends on the project, the team, or the client requirement, as long as it follows the principle of clean code with clear separation of concern and scalable codebase, but let’s say that the MVI is a great architecture pattern that let you stick to those principles.

References

- https://www.youtube.com/watch?v=ilsQZFhjaMo&t=2156s
- https://github.com/spotify/mobius
- https://www.linkedin.com/pulse/mastering-mvi-architecture-revolutionizing-android-app-dhule/
- https://shirsh94.medium.com/decoding-android-architectures-mvvm-mvc-mvp-mvi-6ee0ee313565
- https://medium.com/swlh/mvi-architecture-with-android-fcde123e3c4a

QUESTIONS