International Journal of Computer Applications (0975 - 8887)
Volume 187 - No.42, September 2025

Survey of Presentation-Layer Architecture Patterns for Mobile Applications

Siarhei Krupenich
Mobile Engineer, Resources Lead, Independent Researcher
Warsaw, Poland

ABSTRACT

This article presents an exploration of the Clean Architecture presentation layer patterns for mobile development, which includes native and cross-platform technologies. The study delves into implementation details through schemes and "pseudocode", outlining the strengths and weaknesses of each pattern using a pros-cons analysis. It emphasizes careful interaction with the domain layer to ensure scalability and testability. As a result, a comparative analysis matrix is provided, highlighting the compatibility of each pattern with various technologies, UI paradigms (imperative versus declarative), testability, scalability, and learning complexity. This work aims to offer a clear understanding of when and where each pattern is most appropriate for mobile development.

General Terms

Mobile Software Architecture, Presentation Layer, Architecture Patterns

Keywords

MVP, Viper, MVVM, MVI, MVU, BloC, Redux

1. INTRODUCTION

Nowadays, modern mobile software development increasingly adopts Clean Architecture [1] to ensure that code remains testable, scalable, and modular. Clean Architecture is typically structured into three layers: Data, Domain, and Presentation. The Data layer is responsible for storing and retrieving data from local or external sources and is isolated from the other layers. The Domain layer, which handles business logic, is aware of the Data layer and accesses it through defined interfaces. The Presentation layer depends on the Domain layer but remains unaware of the Data layer. It retrieves the required data using specific entry points in the Domain layer – commonly referred to as use-cases or interactors. Use-cases are generally implemented following a one-intent–one-action principle, while interactors are designed to handle specific tasks. In this work, use-cases are employed, and a sample implementation is provided. The use-case communicates with the Data layer to return domain-related data. There are various patterns available for implementing the presentation layer in mobile software development, generally classified into bidirectional [2] and unidirectional [3] data flows, both of which are examined in this article. Choosing a suitable pattern involves evaluating both the underlying technology and the user interface approach, typically categorized as imperative or declarative [4]. This study presents both UI paradigms alongside diagrams, "pseudocode" examples, and discussions of each pattern's strengths and limitations. To demonstrate practical applicability, each pattern is applied to a conceptual weather forecasting application, chosen for its accessibility and clarity. This work employs two primary comparison methods: a Pros and Cons list covering each pattern, and a Comparative Analysis presented as a feature table. The former highlights the strengths and weaknesses of each pattern, considering their specific assignments and platform peculiarities, while the latter illustrates additional aspects such as data flow type, testability, learning curve, and others. Analyzing the patterns, the following ones have been identified and examined: those with bidirectional data flow – MVP, Viper, and MVVM (actually it could be implemented with both ways) – and those with unidirectional data flow – MVI, MVU, Redux, and BLoC. The main idea of MVP and Viper is to split the data flow into three primary components: the Model, which holds the state of the view; the View, which interacts with users; and the Presenter, which facilitates the connection between the View and the Model. MVVM is based on the Model, View, and ViewModel, which is conceptually similar to the Presenter; however, the ViewModel is more autonomous and fully maintains the state. MVI, abbreviated as Model-View-Intent, is closely related to MVVM in terms of its components. At the same time, it is structured around sending Intents as Messages from the View to the ViewModel, while the View passively listens to the complete State as it is updated. MVU, which is primarily derived from MVI, uses Messages for interaction and a State that is updated through an Update component – typically implemented as a state machine. In the case of Redux, it consists of four main elements: Actions, Reducer, Store, and Middleware. Its core mechanism involves interaction through Actions – for both user intents and state updates. The Store is used to register components, and Middleware enables communication with the domain layer. BLoC is developed using small, logical Blocks of business logic, structured through States, Events (similar to Messages), and the so-called Blocs. Screens can be split across multiple Blocs or encapsulated within a single Bloc, which listens to Events and updates the State accordingly. This work undertakes a thorough, technology-agnostic examination of prominent architectural patterns employed within the presentation layer of mobile applications. The objective is to elucidate their distinct characteristics, operational principles, and suitability across diverse development paradigms-spanning imperative and declarative UI models – and platform ecosystems (e.g., Android, iOS, Flutter, Kotlin Multiplatform Mobile, JavaScript frameworks). Practical insights are furnished through a systematic comparative analysis, including detailed strengths and weaknesses assessments, to aid developers, architects, and researchers in discerning the optimal pattern for specific project constraints and platform requirements. The presented methodologies establish a structured, balanced, and replicable framework, intended to facilitate informed architectural decision-making in the evolving landscape of mobile UI development.

2. RELATED WORK

(1) D. Plakalovic, D. Simic (2010), "Applying MVC and PAC Patterns in Mobile Applications". arXiv:1001.3489 [Online] Available: `https://arxiv.org/abs/1001.3489`

(2) Mario Fuksa, Sandro Speth, Steffen Becker, 2025, "MVVM Revisited: Exploring Design Variants of the Model-View-ViewModel Pattern". arXiv: 2504.18191 [Online] Available: `https://arxiv.org/abs/2504.18191` TODO: check it out, the links not working:

(3) Fajar Pradana, Raziqa Izza Langundi, Djoko Pramono, & Nur Ida Iriani. (2025). Comparative Analysis of MVVM and MVP Patterns Performance on Android Dashboard System. Jurnal Nasional Teknik Elektro Dan Teknologi Informasi, 14(2), 87-95. https://doi.org/10.22146/jnteti.v14i2.18985 [Online] Available: `https://jurnal.ugm.ac.id/v3/JNTETI/article/view/18985`

(4) Luis Cruz, Rui Abreu (2019), "Catalog of Energy Patterns for Mobile Applications", arXiv:1901.03302 [cs.SE], (or arXiv:1901.03302v1 [cs.SE] for this version) [Online] Available: `https://arxiv.org/abs/1901.03302`

(5) Dragos Dobrean & Laura Diosan (2019), "A Comparative Study of Software Architectures in Mobile Applications", DOI:10.24193/subbi.2019.2.04 [Online] Available: `https://www.researchgate.net/publication/337837154_A_Comparative_Study_of_Software_Architectures_in_Mobile_Applications`

(6) Ayush Vijaywargi, Uchinta Kumar Boddapati (2024), "Architectural Patterns in Android Development: Comparing MVP, MVVM, and MVI.", DOI Link: `https://doi.org/10.22214/ijraset.2024.60762` [Online] Available: `https://www.researchgate.net/publication/337837154_A_Comparative_Study_of_Software_Architectures_in_Mobile_Applications`

3. METHODOLOGY

This study employs a dual-comparison methodology, combining a Pros and Cons List with a Feature-Based Tabular Matrix. The Pros and Cons List allows for a qualitative assessment of the practical benefits of each pattern, drawing from published literature, community practices, and real-world applicability. In parallel, the Tabular matrix provides a structured, side-by-side view across key evaluation dimensions such as testability, scalability, platform integration, and complexity of adoption. As an exploratory study, the analysis is based on qualitative synthesis rather than experimental measurement, ensuring both a narrative and systematic perspective for evaluating the diverse patterns of the presentation layer in Android, iOS, and cross-platform ecosystems.

3.1 Pros and Cons List

A straightforward method where the advantages and disadvantages of each alternative are listed, facilitating a balanced view of each option [5]. Effective for initial evaluations and discussions, especially when introducing new concepts or patterns. The Pros and Cons List is an intuitive and widely used evaluation technique that helps decision makers consider both the benefits and drawbacks of each alternative in a balanced manner. It is particularly effective during the early stages of analysis, offering a low barrier approach to structuring thought and facilitating discussions around complex options or unfamiliar design patterns [6]. This method does not rely on numerical weighting or complex models, but instead encourages qualitative insight and reflection, making it suitable for settings where time or data are limited [7]. When applied to mobile presentation layer patterns, the pros and cons approach enables developers and researchers to highlight key trade-offs, for example, MVVM support for testability versus its increased architectural complexity, in a format that is accessible to both technical and non-technical audiences. Moreover, it serves as a foundation for deeper analysis, acting as a precursor to more formal methods such as the Analytic Hierarchy Process or empirical performance benchmarking [8]. This method follows each pattern exploration and is used to outline and assess the individual strengths and weaknesses of the approach qualitatively.

3.2 Feature-Based Comparison (Tabular Matrix)

This method involves identifying key characteristics or criteria relevant to the subject (for example, presentation layer patterns) and comparing each alternative (for example, MVP, Viper, MVVM, MVI, MVU, Bloc) against these features in a structured table [9]. Ideal for providing a clear side-by-side comparison of different patterns based on specific attributes such as testability, scalability, or ease of implementation. The feature-based comparison approach uses a structured table to compare alternatives – such as architectural patterns – across predefined evaluation criteria. It begins by identifying key features (e.g., testability, scalability, implementation complexity) relevant to the domain and then scores each pattern (e.g., MVP, VIPER, MVVM, MVI, MVU, BLoC) against these criteria in a matrix format. The strength of this method lies in its clarity and accessibility, enabling the reader to quickly discern relative strengths and trade-offs [10]. Such a comparison is particularly effective for mobile presentation layer patterns: the structured layout highlights differences in architectural characteristics like boilerplate code or life-cycle handling-in a visually intuitive way. Although inherently qualitative, the feature matrix can be extended to include numerical scores, making it suitable for mixed-method analysis. This method appears in the conclusion section, where all patterns are evaluated side by side across consistent criteria, allowing us to highlight distinguishing features in a comparative format.

4. PATTERNS ANALYSIS

This analysis examines each pattern through detailed explanations, diagrams, and ready-to-use pseudocode. The code provided addresses a specific task that involves a weather forecasting application, making the discussion more relevant to real-world scenarios. The application itself is simple, with only three states: current weather, loading, and error. Additionally, the pseudocode introduces a new element, the `FetchWeatherUseCase` interface, which references an instance beyond the scope of this article. This use case serves to return a model representing the current location and acts as an entry point to the domain layer. Consequently, each pattern is evaluated with a list of pros and cons, and its suitability is assessed with declarative or imperative UI approaches.

4.1 MVP or Viper

While both MVP (Model-View-Presenter) and Viper patterns are built upon a similar conceptual foundation of separating concerns, their typical adoption varies significantly by platform, with MVP being prevalent in Android development and Viper being more commonly encountered in iOS. Aside from this platform focus, the components within each pattern are largely similar. For simplicity, this article will refer to MVP throughout, implicitly including Viper as well. The core concept of MVP involves the Presenter updating the model, which in turn prompts the Presenter to update the View. The View is aware of the Presenter and triggers actions that the Presenter handles. Subsequently, the Presenter updates the model and refreshes the view accordingly. Figure 1 illustrates the behavior:

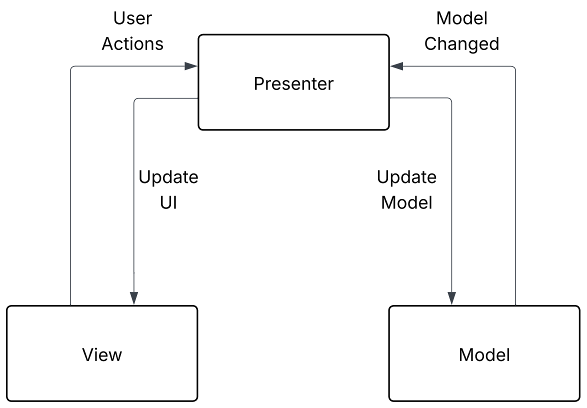

Fig. 1. MVP schema.

4.1.1 Code snippet. Drawing from the schema in Figure 1, the pattern can be rigorously defined as follows.

```
// Model
CLASS WeatherModel
    VARIABLE uiWeather: UiWeather
    CONSTRUCTOR(uiWeather: UiWeather)
        SET this.uiWeather = uiWeather

// View Contract
INTERFACE WeatherViewContract
    FUNCTION showLoading()
    FUNCTION showWeather(weather: WeatherModel)
    FUNCTION showError(error: Exception)

// Presenter
CLASS WeatherPresenter
    PRIVATE VARIABLE view: WeatherViewContract
    PRIVATE VARIABLE fetchWeather:
        FetchWeatherUseCase
    PRIVATE VARIABLE mapper: DomainToUiMapper
    CONSTRUCTOR(view: WeatherViewContract)
        SET this.view = view
    FUNCTION loadWeather()
        view.showLoading()
        TRY
            weather = fetchWeather.call()
            uiWeather = mapper.map(weather)
            view.showWeather(
                NEW WeatherModel(uiWeather)
            )
        CATCH error AS Exception
            view.showError(error)


// View
CLASS WeatherView IMPLEMENTS WeatherViewContract
    PRIVATE VARIABLE presenter: WeatherPresenter
    FUNCTION initialize()
        presenter = NEW WeatherPresenter(this)
        presenter.loadWeather()
    FUNCTION showLoading()
        showLoadingUI()
    FUNCTION showWeather(weather: WeatherModel)
        showWeatherUI(weather)
    FUNCTION showError(error: Exception)
        showErrorUI()
```

The view implements the `WeatherViewContract`, which is recognized by the `WeatherPresenter` responsible for managing all states. Since the presenter operates on the model to update the UI, this pattern is particularly well suited for imperative approaches, such as Android Views with XML or iOS UIKit.

4.1.2 Pros and Cons. Table 1 demonstrates the strengths and weaknesses of the MVP and Viper patterns.

Table 1. Pros and Cons of the MVP and Viper patterns

Strength	Weaknesses
- Separation of Concerns	- Boilerplate Code
- Improved Testability	- Tight Coupling between View and Presenter
- Platform-Specific Support	- Overkill for Small Projects
- Deterministic Flow	- VIPER's learning curve
- VIPER adds scalability	- Cyclic Dependencies Risk

4.2 MVVM

MVVM (Model-View-ViewModel) consists of three main components: View, ViewModel, and Model, where the View is responsible for appearing UI by listening to actions from the ViewModel. The sending of messages from the View to the ViewModel is performed via the ViewModel input methods. At the same time, the ViewModel handles the Model to update it and responds to the View that represents the UI using the updated Model. As an entry to the domain module, interactors or use-cases may be used in the ViewModel; thus, Figure 2 demonstrates the logic:

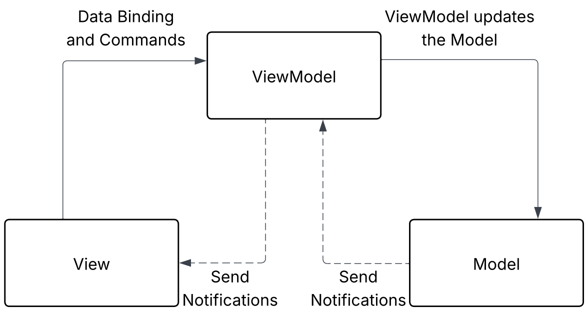

Fig. 2. MVVM Schema.

4.2.1 Code snippet. Drawing from the schema in Figure 2, the pattern can be rigorously defined as follows.

```
// ViewModel
CLASS WeatherViewModel
    PRIVATE VARIABLE weather: UiWeather = null
    PRIVATE VARIABLE error: Exception = null
    PRIVATE VARIABLE isLoading: Boolean = false
    PRIVATE VARIABLE fetchWeather:
        FetchWeatherUseCase
    PRIVATE VARIABLE mapper: DomainToUiMapper

    CONSTRUCTOR(
        fetchWeather: FetchWeatherUseCase,
        mapper: DomainToUiMapper
    )
        SET this.fetchWeather = fetchWeather
        SET this.mapper = mapper
    FUNCTION getWeather() RETURNS UiWeather
        RETURN weather
    FUNCTION getError() RETURNS Exception
        RETURN error
    FUNCTION getIsLoading() RETURNS Boolean
        RETURN isLoading
    FUNCTION fetchWeather()
        SET isLoading = true
        TRY
            domainWeather = fetchWeather.call()
            weather = mapper.map(domainWeather)
            error = null
        CATCH exception AS Exception
            error = exception
        SET isLoading = false

// View
CLASS WeatherView
    VARIABLE viewModel: WeatherViewModel
    FUNCTION render()
        RETURN renderWeatherState(viewModel)
    FUNCTION renderWeatherState(
        viewModel: WeatherViewModel
    )
        IF viewModel.getIsLoading() THEN
            RETURN renderLoadingIndicator()
        ELSE IF viewModel.getError() NOT null
            THEN RETURN renderError(
                viewModel.getError()
            )
        ELSE IF viewModel.getWeather() NOT null
            THEN RETURN renderWeather(
                viewModel.getWeather(),
                onRefresh = viewModel
                    .fetchWeather()
            )
```

The View knows about the ViewModel, which has an entry-point to a domain layer (e.g. use-cases) and updates the Model that is then used by the View. Since the View is updated through specific properties of the ViewModel rather than a unified state, this pattern is commonly adopted for imperative UI approaches; in the meantime, it may be used for the Declarative UI approach as well by splitting the ViewModel by Input/Output interfaces, incorporating a state into the ViewModel and listening to it to represent the View. The pattern is often considered the gold standard in Android development and is also used in other platforms such as Xamarin, though less frequently in Flutter, Redux, and iOS.

4.2.2 Pros and Cons. Table 2 demonstrates the strengths and weaknesses of the MVVM pattern.

Table 2. Pros and Cons of the MVVM pattern

Strength	Weaknesses
- Separation of Concerns	- Steeper Learning Curve
- Improved Testability	- Overhead for Simple Apps
- Two-Way Data Binding	- Debugging Challenges
- Scalability	- Tooling and Framework Dependence
- Platform Support (Android, iOS, Flutter, MAUI)	- Potential ViewModel Bloat

4.3 Redux

Redux is a presentation layer pattern with unidirectional data flow (UDF-based pattern) and is also a part of the TEA (The Elm Architecture). Basically, it includes the following core components: View, Actions, Reducer, and Store. Optionally, Middleware can be included for interacting with the Domain layer. All communication is conducted through Actions, both for dispatching and for consuming the ready-to-use State, which is also an additional component to render the View. Based on the state, the view is being redrawn, while the reducer, in fact, is a state machine and is responsible for handling all actions and updating the state. An additional component is the Store, which is responsible for registering all the components – Reducer, initial State, and Middleware (if it is involved). Figure 3 illustrates the data flow:

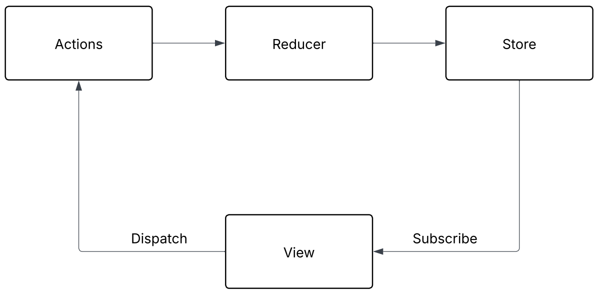

Fig. 3. Redux Schema.

4.3.1 Code snippet. As depicted in the schema in Figure 3, the formal structure of the pattern is as follows.

```
// Actions
CLASS FetchWeatherAction
CLASS ReloadWeatherAction
CLASS LoadingWeatherAction
CLASS WeatherLoadedAction
    VARIABLE weather: UiWeather
    CONSTRUCTOR(weather: UiWeather)
CLASS WeatherErrorAction
    VARIABLE error: Exception
    CONSTRUCTOR(error: Exception)

// State to appear UI
CLASS WeatherState
    VARIABLE weather: UiWeather = null
    VARIABLE isLoading: Boolean = false
    VARIABLE error: Exception = null
    FUNCTION initLoading()
        RETURN NEW WeatherState(
            isLoading = true
        )
    FUNCTION copyWith(
        OPTIONAL weather: UiWeather,
        OPTIONAL isLoading: Boolean,
        OPTIONAL error: Exception
    ) RETURNS WeatherState
        RETURN NEW WeatherState(
            weather = IF weather IS SET
                THEN weather
                ELSE this.weather,
            isLoading = IF isLoading IS SET
                THEN isLoading
                ELSE this.isLoading,
            error = IF error IS SET
                THEN error
                ELSE this.error
        )

// Reducer which updates the state
FUNCTION weatherReducer(
    state: WeatherState,
    action: Action
) RETURNS WeatherState
    IF action IS LoadingWeatherAction THEN
        RETURN state.copyWith(
            isLoading = true,
            weather = null,
            error = null
        )
    ELSE IF action IS WeatherLoadedAction THEN
        RETURN state.copyWith(
            isLoading = false,
            weather = action.weather
        )
    ELSE IF action IS WeatherErrorAction THEN
        RETURN state.copyWith(
            isLoading = false,
            error = action.error
        )
    ELSE RETURN state

// Middleware { an optional component which is
// responsible here to call
// the fetchWeatherUseCase
FUNCTION weatherMiddleware(
    store: Store<WeatherState>,
    action: Action, next: Function
)
    IF action IS FetchWeatherAction
        OR action IS ReloadWeatherAction THEN
        IF action IS ReloadWeatherAction THEN
            store.dispatch(LoadingWeatherAction)
        TRY
            domainWeather = fetchWeatherUseCase
                .call()
            uiWeather = mapper.map(domainWeather)
            store.dispatch(
                WeatherLoadedAction(uiWeather)
            )
        CATCH e AS Exception
            store.dispatch(WeatherErrorAction(e))
    // Always call the next middleware/reducer,
    // depending on a particular implementation
    next(action)

// Store which registers and connects
// all the related components
VARIABLE store = Store(
    reducer = weatherReducer,
    initialState = WeatherState.initLoading(),
    middleware = [weatherMiddleware]
)

// ViewModel which is not a classical ViewModel
// but only keeps the State using the Store
CLASS WeatherViewModel
    VARIABLE isLoading: Boolean
    VARIABLE error: Exception
    VARIABLE weather: UiWeather
    VARIABLE fetchWeather: Function
    STATIC FUNCTION fromStore(
        store: Store<WeatherState>
    ) RETURNS WeatherViewModel
        RETURN NEW WeatherViewModel(
            isLoading = store.state.isLoading,
            error = store.state.error,
            weather = store.state.weather,
            fetchWeather = FUNCTION() {
                store.dispatch(
                    ReloadWeatherAction
                )
            }
        )

// View
CLASS WeatherView
    FUNCTION initialize()
        // No-op here
    FUNCTION render()
        RETURN renderWithStoreConnector(
            onInit = FUNCTION(store) {
                store.dispatch(FetchWeatherAction)
            },
            converter = FUNCTION(store) {
                RETURN WeatherViewModel
                    .fromStore(store)
            },
            builder = FUNCTION(viewModel) {
                RETURN renderWeather(viewModel)
            }
        )

    FUNCTION renderWeather(
        viewModel: WeatherViewModel
    )
        IF viewModel.isLoading THEN
            RETURN renderLoading()
        ELSE IF viewModel.error IS NOT null THEN
            RETURN renderError(viewModel.error)
        ELSE IF viewModel.weather IS NOT null THEN
            RETURN renderWeatherData(
                viewModel.weather,
                onRefresh = viewModel.fetchWeather
            )
```

4.3.2 Pros and Cons. Table 3 demonstrates the strengths and weaknesses of the Redux pattern.

Table 3. Pros and Cons of the Redux pattern

Strength	Weaknesses
- Predictable State Management - Single Source of Truth - Time Travel & Debugging Tools - Decoupled Architecture - Scalable for Large Applications - Cross-platform Pattern	- Boilerplate Code - Steep Learning Curve - Verbosity for Simple Use Cases - Indirect State Changes - Maintenance Burden

The Redux pattern is being actively used with JS-based mobile development frameworks; at the same time, it also tends to be applied in iOS (SwiftUI) and rarely Flutter.

4.4 MVI

The MVI pattern (Model-View-Intent) is a part of The Elm Architecture (TEA) and consists of three components and in general refers to MVVM. The main difference is related to the communication between the View and its ViewModel, which is related to the State-based principle involving a single parent object representing UI states. On the other hand, View sends messages, referred to as dispatching intents, in the context of MVI. Like states, Intents are also operated as one parent object containing child objects, each representing a specific Intent. Both dispatching Intents and consuming results to handle the State are usually handled using a switch-based structure or its analogue. The following picture fully represents the pattern:

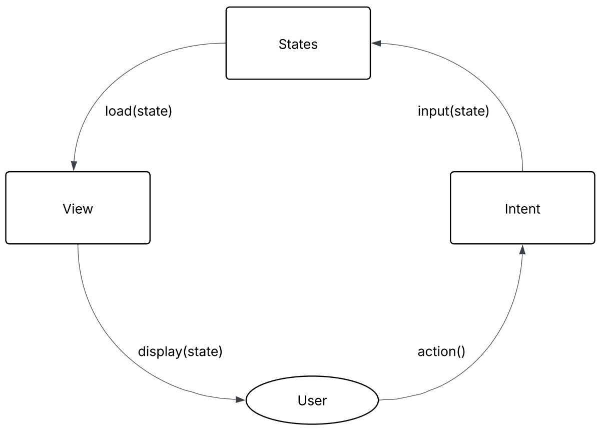

Fig. 4. MVI Schema.

4.4.1 Code snippet. The schema in Figure 4 informs the following formal representation of the pattern:

```
ABSTRACT CLASS WeatherIntent
CLASS LoadWeather EXTENDS WeatherIntent
CLASS RefreshWeather EXTENDS WeatherIntent

// State
CLASS WeatherState
    VARIABLE isLoading: Boolean = false
    VARIABLE uiWeather: UiWeather = null
    VARIABLE error: Exception = null
    CONSTRUCTOR(
        OPTIONAL isLoading: Boolean,
        OPTIONAL uiWeather: UiWeather,
        OPTIONAL error: Exception
    )
    FUNCTION copyWith(
        OPTIONAL isLoading: Boolean,
        OPTIONAL uiWeather: UiWeather,
        OPTIONAL error: Exception
    ) RETURNS WeatherState
        RETURN NEW WeatherState(
            isLoading = IF isLoading IS SET
                THEN isLoading
                ELSE this.isLoading,
            uiWeather = IF uiWeather IS SET
                THEN uiWeather
                ELSE this.uiWeather,
            error = error // overrides always
        )
    STATIC FUNCTION initial() RETURNS WeatherState
        RETURN NEW WeatherState()

// ViewModel
CLASS WeatherViewModel EXTENDS Observable
```

```
    PRIVATE VARIABLE state: WeatherState
        = WeatherState.initial()
    PRIVATE VARIABLE fetchWeather:
        FetchWeatherUseCase
    PRIVATE VARIABLE mapper: DomainToUiMapper
    CONSTRUCTOR(
        fetchWeather: FetchWeatherUseCase,
        mapper: DomainToUiMapper
    )
    FUNCTION getState() RETURNS WeatherState
        RETURN state
    FUNCTION dispatch(intent: WeatherIntent)
        SWITCH intent
            CASE LoadWeather:
            CASE RefreshWeather:
                AWAIT loadWeather()
            END
        END SWITCH
    PRIVATE FUNCTION loadWeather()
        updateState(state.copyWith(
            isLoading = true,
            error = null)
        )
        TRY
            domainWeather = fetchWeather.call()
            uiWeather = mapper.map(domainWeather)
            updateState(
                state.copyWith(
                    isLoading = false,
                    uiWeather = uiWeather
                )
            )
        CATCH e AS Exception
            updateState(
                state.copyWith(
                    isLoading = false,
                    error = e
                )
            )
    PRIVATE FUNCTION updateState(
        newState: WeatherState
    ) state = newState

// View
CLASS WeatherView
    FUNCTION build()
        RETURN WeatherViewInternal()
CLASS WeatherViewInternal
    FUNCTION build()
        state = observe(WeatherViewModel)
            .getState()
        RETURN renderWeatherUI(state)
    FUNCTION renderWeatherUI(state: WeatherState)
        IF state.isLoading THEN
            RETURN renderLoading()
        ELSE IF state.error IS NOT null THEN
            RETURN renderError(state.error)
        ELSE IF state.uiWeather IS NOT null THEN
            RETURN renderWeather(
                state.uiWeather,
                onRefresh = FUNCTION() {
                    viewModel.dispatch(
                        RefreshWeather
                    )
                }
            )
```

Figure 4 illustrates that the flow between states of intent is cycled and follows only one direction, in contrast to MVVM or MVP (Viper). A user sends an intent which is being handled by the ViewModel, and consequently, the state is updated and represented by the View (UI). Therefore, the pattern refers to a state-based UI flow and follows a one-direction flow (or unidirectional data flow – UDF); it is mainly suited for the declarative UI approach. Currently, MVI is considered a gold standard for the declarative approach in Android (e.g., Jetpack Compose), and it can also be used in some cross-platform technologies such as Flutter or Kotlin Multiplatform (KMP).

4.4.2 Pros and Cons. Table 4 demonstrates the strengths and weaknesses of the MVI pattern.

Table 4. Pros and Cons of the MVI pattern

Strength	Weaknesses
- Unidirectional Data Flow	- Boilerplate Code
- Single Source of Truth	- Verbose State Management
- Highly Testable	- Steep Learning Curve
- Concurrency Safety	- Performance Concerns in Large UIs
- Reactive by Design	- Tooling and Ecosystem Variability

4.5 MVU

MVU (Model-View-Update) belongs to the TEA group with unidirectional data flow and consists of a State which is being handled by the UI (or View); the View, which renders the State; and Update, which is, in fact, a method responsible for changing the State to appear. The current state is used to appear in the UI (View), or the UI dispatches an action, which is being handled in the Update and returns a new state. Communication between View and Update is conducted by messages and actions that are being handled by Update. Figure 5 illustrates the behavior:

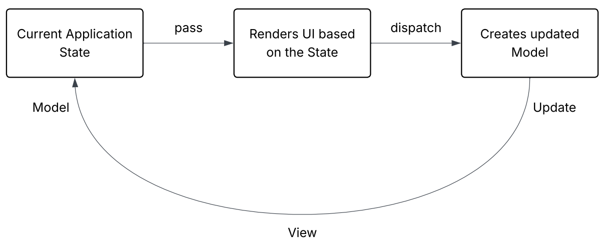

Fig. 5. MVU Schema.

4.5.1 Code snippet. Based on the architectural schema in Figure 5, the formalization of the pattern is presented below:

```
// Messages for communicating
ABSTRACT CLASS WeatherMsg
CLASS FetchWeather EXTENDS WeatherMsg
CLASS WeatherLoaded EXTENDS WeatherMsg
    VARIABLE data: UiWeather
    CONSTRUCTOR(data)
CLASS WeatherFailed EXTENDS WeatherMsg
    VARIABLE error: Exception
    CONSTRUCTOR(error)

// Model to appear data
CLASS WeatherModel
    VARIABLE model: UiWeather = null
    VARIABLE isLoading: Boolean = false
    VARIABLE error: Exception = null
    CONSTRUCTOR(
        OPTIONAL model: UiWeather,
        OPTIONAL isLoading: Boolean,
        OPTIONAL error: Exception
    )
    FUNCTION copyWith(
        OPTIONAL isLoading: Boolean,
        OPTIONAL data: UiWeather,
        OPTIONAL error: Exception
    ) RETURNS WeatherModel
        RETURN NEW WeatherModel(
            model = IF data IS SET
                THEN data
                ELSE this.model,
            isLoading = IF isLoading IS SET
                THEN isLoading
                ELSE this.isLoading,
            error = IF error IS SET
                THEN error
                ELSE this.error
        )

// Update to handle the messages and change
CLASS WeatherUpdate
    PRIVATE VARIABLE model: WeatherModel
    PRIVATE VARIABLE fetchWeather:
        FetchWeatherUseCase
    PRIVATE VARIABLE mapper:
        DomainToUiMapper
    PRIVATE VARIABLE setModel:
        FUNCTION(WeatherModel)
    CONSTRUCTOR(
        fetchWeather, mapper, model, setModel
    )
    ASYNC FUNCTION update(msg: WeatherMsg)
        SWITCH msg
            CASE FetchWeather:
                setModel(
                    model.copyWith(
                        isLoading = true,
                        error = null
                    )
                )


                TRY
                    domainData = fetchWeather
                        .call()
                    uiData = mapper.map(
                        domainData
                    )
                    setModel(
                        model.copyWith(
                            isLoading = false,
                            data = uiData
                        )
                    )
                CATCH e AS Exception
                    setModel(
                        model.copyWith(
                            isLoading = false,
                            error = e
                        )
                    )
                END TRY
            CASE WeatherLoaded(data):
                setModel(
                    model.copyWith(
                        data = data,
                        isLoading = false
                    )
                )
            CASE WeatherFailed(error):
                setModel(
                    model.copyWith(
                        error = error,
                        isLoading = false
                    )
                )
        END SWITCH

// Notifier which dispatches messages
// to the Update
CLASS WeatherNotifier EXTENDS Observable
    PRIVATE VARIABLE model: WeatherModel =
        NEW WeatherModel()
    PRIVATE VARIABLE updater:
        WeatherUpdate
    FUNCTION getModel()
        RETURN model
    CONSTRUCTOR(
        fetchWeather: FetchWeatherUseCase,
        mapper: DomainToUiMapper
    )
        updater = NEW WeatherUpdate(
            fetchWeather,
            mapper,
            model,
            setModel
        )
        updater.update(FetchWeather)
    FUNCTION setModel(
        newModel: WeatherModel
    )
        model = newModel
        notifyObservers()
    FUNCTION dispatch(msg: WeatherMsg)
        RETURN updater.update(msg)

// View represents the State
CLASS WeatherView
    FUNCTION build()
        RETURN WeatherInternal()
CLASS WeatherInternal
    FUNCTION build()
        notifier = observe(WeatherNotifier)
        RETURN renderWeather(
            notifier.getModel()
        )
    FUNCTION renderWeather(model: WeatherModel)
        IF model.isLoading THEN
            RETURN showLoading()
        ELSE IF model.error NOT null
            THEN RETURN showError(model.error)
        ELSE IF model.model NOT null
            THEN RETURN showWeather(
                model.model,
                onRefresh = FUNCTION() {
                    notifier.dispatch(
                        FetchWeather
                    )
                }
            )
```

The MVU pattern has unidirectional data flow (UDF) and belongs to The Elm Architecture (TEA); therefore, it is recommended to be used with a declarative UI approach (e.g., Flutter, Jetpack Compose (Android native), SwiftUI (iOS) and React).

4.5.2 Pros and Cons. Table 5 demonstrates the strengths and weaknesses of the MVU pattern:

Table 5. Pros and Cons of the MVU pattern

Strength	Weaknesses
- Predictable State Management - Testability - Unidirectional Data Flow - Consistency Across Platforms - Encourages Separation of Concerns - Inspiration for Modern Architectures	- Boilerplate Code - Steep Learning Curve - Scalability Challenges - Tooling and Ecosystem - Performance Overhead in Some Cases

4.6 BLoC

BLoC (Business Logic Component), a part of The Elm Architecture (TEA) with unidirectional data flow, consists of the following main components: Event, which is used for messaging; State, which is handled by the UI (or View, as a separate component); and Bloc, the component responsible for processing messages and updating the state. Typically, business logic is split across logical units represented as individual Blocs. Figure 6 illustrates the logic of the BLoC pattern.

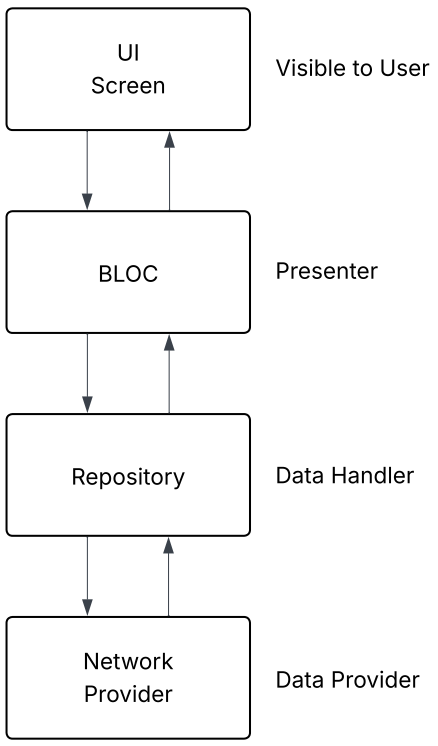

Fig. 6. BLoC Schema.

4.6.1 Code snippet. The following formal representation of the pattern is derived from the schema presented in Figure 6.

```
// Events
ABSTRACT CLASS WeatherEvent
CLASS GetWeatherEvent EXTENDS WeatherEvent

// State
ABSTRACT CLASS WeatherState
CLASS WeatherInitial EXTENDS WeatherState
CLASS WeatherLoading EXTENDS WeatherState
CLASS WeatherLoaded EXTENDS WeatherState
    VARIABLE weather: UiWeather
    CONSTRUCTOR(weather)
CLASS WeatherError EXTENDS WeatherState
    VARIABLE message: String
    VARIABLE throwable: Exception
    CONSTRUCTOR(message, throwable)

// Bloc
CLASS WeatherBloc EXTENDS
    Bloc<WeatherEvent, WeatherState>

    VARIABLE fetchWeather: FetchWeatherUseCase
    VARIABLE mapper: DomainToUiMapper
    CONSTRUCTOR(fetchWeather, mapper)
        SET this.fetchWeather = fetchWeather
        SET this.mapper = mapper
        INITIALIZE WITH WeatherInitial
        ON GetWeatherEvent DO (event, emit) ->
            emit(WeatherLoading)
            TRY
                weatherDomain = fetchWeather()
                uiWeather = mapper.map(
                    weatherDomain
                )
                emit(WeatherLoaded(uiWeather))
            CATCH error AS Exception
                emit(
                    WeatherError(
                        error.toString(),
                        error
                    )
                )

// View
CLASS WeatherView
    FUNCTION build()
        RETURN BlocBuilder
            <WeatherBloc, WeatherState>
        (
            builder = (context, state) ->
                buildFunction(context, state)
        )

    VARIABLE buildFunction: FUNCTION(
        context: BuildContext,
        state: WeatherState
    )
        IF state IS WeatherLoading
            THEN RETURN showLoading()
        ELSE IF state IS WeatherLoaded THEN
            RETURN showWeather(
                state.weather,
                onRefresh = FUNCTION() {
                    context.dispatch(
                        GetWeatherEvent
                    )
                })
        ELSE IF state IS WeatherError THEN
            RETURN showError(state.message)
        ELSE
            RETURN showInitial()
```

4.6.2 Pros and Cons. Table 6 demonstrates the strengths and weaknesses of the BLoC pattern.

Table 6. Pros and Cons of the BLoC pattern

Strength	Weaknesses
- Separation of Concerns	- Boilerplate Code
- Highly Testable	- Steep Learning Curve for Beginners
- Reusability	- Verbosity Hurts Productivity
- Scalable Architecture	- Debugging Can Be Tricky
- Stream-based Reactive Programming	- Tight Coupling with flutter_bloc Library
- Community and Tooling Support	

5. COMPARATIVE ANALYSIS

As this work is exploratory in nature, comparative analysis emphasizes conceptual and qualitative evaluation rather than experimental or performance-based measurement.
Table 7 represents all the strengths, weaknesses, technology support, testability, and complexity of learning.
It consists of MVP, Viper, MVVM, MVI, MVU, Redux, and BLoC presentation layer patterns, where each pattern is characterized by its technology, UI approach, whether UDF is supported, Testability, Scalability, and Learning curve. Both MVP and Viper, which are actually representations of patterns with the same components, are actively used in the development of both mobile platforms (MVP - Android, Viper - iOS); are appropriate for the Imperative UI approach, having bidirectional data flow, and allow us to test their components separately. Since Viper may have more components (e.g. navigation-related components), it is a bit more scalable than MVP and, consequently, is a bit more complicated for learning purposes.
MVVM, which is a gold standard not only for Android development but also is actively used in KMP, MAUI, and Xamarin, and can be appropriate for both Declarative and Imperative UI approaches depending on a particular implementation and therefore, MVVM is not a UDF-based pattern in classical understanding; at the same time, MVVM can be modified to follow the UDF principles. MVVM is highly split by some components that are clearly responsible for their own purposes; the pattern supports tests and is highly scalable. Although there are some nuances to investigate, it is not too hard to learn.
MVI is appropriate to use in Android native Software development based on Declarative UI approach (e.g. Jetpack Compose), KMP; additionally, it can be incorporated into other technologies (e.g. Flutter which is not a standard in general). MVI is not preferred for the Imperative approach, since it belongs to UDF patterns. As it has additional components for interacting between ViewModel and View, it has High Scalability and is strongly supported by testing; curve learning of MVI is high due to additional components and not obvious data flow.
MVU is a good solution to incorporate into mobile applications using Flutter via Elm-inspired libraries, F# (Fabulous) and Elm, which is not a mobile software development framework, but which is important to highlight. It mainly refers to the Declarative style approach due to its unified data flow and is not recommended for the Imperative UI approach. The pattern allows us to strictly split a project by a variety of components, making it highly scalable and supported for tests. Since it has no obvious messaging structure between update components and view, it has medium-high complexity of learning.
The redux pattern allows us to create good mobile solutions using React Native, Flutter, JavaScript-based tools, and iOS native applications, which is strongly designed for the Declarative UI approach. Using the pattern, a solution is being split by a variety of components which enhance Testability and make it highly supported due to its high Scalability. At the same time, its learning curve remains high due to the highlighted variety of the components.
BLoC is used mainly in Flutter-based solutions and involves splitting and representing the entire business logic as smaller blocks. The blocks serve for the Declarative UI approach since the pattern has the Unidirectional data flow which makes the pattern highly supported for testing and the scalability is High. Due to its UDF specific aspects and components, its learning curve is medium to high.

Table 7. Comparative analysis

Pattern	Best Fit Technologies	Declarative UI	Imperative UI	UDF	Testability	Scalability	Complexity	Notes
MVP (Model View Presenter)	Android (Java/Kotlin), iOS (with adaptations)	Not Preferred	Supported	Not Preferred	Supported	Medium	Medium	Classical pattern; explicit View-Presenter boundary
VIPER	iOS (Swift), Clean iOS architectures	Not Preferred	Supported	Not Preferred	Supported	High	High	Heavy structure, often considered over-engineered
MVVM (Model View ViewModel)	Android (Jetpack Compose, XML), Xamarin, MAUI, KMP	Supported	Supported	Partially supported	Supported	Medium–High	Medium	Common in Android, flexible with Compose
MVI (Model View Intent)	Android (Jetpack Compose), Kotlin Multiplatform	Supported	Not Preferred	Supported	Strong support	High	High	Fully reactive, inspired by UDF principles
MVU (Model View Update)	Elm, Flutter (via Elm-inspired libs), F# (Fabulous)	Supported	Not Preferred	Supported	Supported	High	Medium–High	Direct implementation of Elm architecture
Redux	React Native, Flutter (via Redux), JavaScript-based tools	Supported	Not Preferred	Strong support	Strong support	High	High	Centralized state container, verbose
BLoC (Business Logic Component)	Flutter	Supported	Not Preferred	Supported	Strong support	High	Medium–High	Flutter-native pattern; UDF via Streams or Cubit

6. CONCLUSION

In this work, the core concepts of the presentation layer in mobile software development are explored, focusing on architectural patterns commonly adopted across platforms. These patterns are broadly categorized by the direction of data flow - either bidirectional or unidirectional - and their alignment with the imperative or declarative UI paradigms.
In order to thoroughly compare the patterns, a Pros and Cons analysis was applied and a comparative tabular study was conducted, highlighting key aspects such as testability, scalability, platform closeness, and compatibility with modern development approaches. This work conducted a study of the patterns including MVP, Viper, MVVM, MVI, MVU, and BLoC, each suitable for specific tasks, ecosystems, and developer communities.
Diving into the concepts, strengths, and weaknesses of patterns and recognizing their trade-offs can help make the correct decisions to select a particular pattern for its environment carefully, which must lead to keeping the code base cleaner, more maintainable, and testable. The findings of this study may also serve as a foundation for further empirical evaluation or automation of architectural decisions in mobile frameworks.
As a future direction, further research will be aimed at benchmarking the most widely used patterns for Android development, particularly comparing MVI and MVVM in combination with both Views and Jetpack Compose for UI implementation. Such empirical evaluations will provide performance-based insights, complementing the conceptual analysis presented here and helping practitioners align architectural choices with efficiency and platform evolution.

7. REFERENCES

[1] Martin, Robert C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.

[2] Khedker, U. P., & Dhamdhere, D. M. (1993). Complexity of bidirectional data flow analysis. In Proceedings of the Annual ACM Symposium on Principles of Programming Languages (pp. 397–408). ACM.

[3] Android Developers. "Architecting your Compose UI: Unidirectional data flow." Android Developers Documentation. "A unidirectional data flow (UDF) is a design pattern where state flows down and events flow up. By following UDF, you

decouple composables that display state in the UI from the parts of your app that store and change state.” [Online] Available: https://developer.android.com/develop/ui/compose/architecture [Accessed: 7-July-2023]

[4] Shiprocket Engineering. (2024). “Imperative vs Declarative: a practical example.” Shiprocket Tech Blog. [Online] Available: https://acme.shiprocket.com/shiprockets-declarative-ui/ [Accessed: 7-July-2023]

[5] Cullen, C. (2023, May 4). “Using a Pros/Cons List to Help Navigate Technical Discussions.” ChipCullen.com. [Online] Available: https://chipcullen.com/using-a-pros-cons-list-in-technical-discussions/ [Accessed: 7-July-2023]

[6] Browne, M. N., & Keeley, S. M. (2011). Asking the Right Questions: A Guide to Critical Thinking (10th ed.). Pearson Education.

[7] Harris, R. (2012). Introduction to Decision Making. VirtualSalt. [Online] Available: http://www.virtualsalt.com/crebook5.htm [Accessed: 7-July-2023]

[8] Paul, R., & Elder, L. (2014). Critical Thinking: Tools for Taking Charge of Your Professional and Personal Life (2nd ed.). FT Press.

[9] Kouli, M., & Rasoolzadegan, A. (2022, July 21). “A Feature-Based Method for Detecting Design Patterns in Source Code.” Symmetry (MDPI), 14(7), 1491. [Online] Available: https://www.mdpi.com/2073-8994/14/7/1491 [Accessed: 7-July-2023]

[10] Nazar, N., Aleti, A., & Zheng, Y. (2020). “Feature-Based Software Design Pattern Detection.” arXiv preprint. [Online] Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/82f39c7409155b74d15d73b048f06771-Paper-Datasets_and_Benchmarks.pdf [Accessed: 7-July-2023]