THE USA JOURNALS
THE AMERICAN JOURNAL OF ENGINEERING AND TECHNOLOGY (ISSN – 2689-0984)
VOLUME 06 ISSUE07

PUBLISHED DATE: - 22-07-2024
DOI: - https://doi.org/10.37547/tajet/Volume06Issue07-06 **PAGE NO.: - 50-56**

RESEARCH ARTICLE **Open Access**

APPLYING MVI ARCHITECTURE TO ENHANCE TESTABILITY AND MAINTAINABILITY IN ANDROID APPLICATIONS

Chike Mgbemena
Mobile Software Engineer, Lagos, Nigeria

Abstract
In the world of Android application development, building scalable and easily supported systems is a crucial task. The Model-View-Intent (MVI) architectural pattern has proven to be an effective approach to achieve these goals due to the strict separation of responsibilities between components, unidirectional data flow and an emphasis on immutability. This article discusses the basic principles of the MVI architecture, its advantages and disadvantages, as well as its impact on the testability and maintainability of applications. Using MVI allows you to improve the predictability of application behavior, simplify debugging and state management, and also contributes to modularity and ease of code maintenance. In conclusion, examples of successful application of MVI in practice are discussed, which confirms its usefulness for the development of complex and highly loaded applications.

Keywords MVC architecture, increased testability, increased maintainability, android applications, MVIDEOPLAYER.

INTRODUCTION

In the field of Android application development, creating scalable and easily maintainable applications is a top priority. To achieve this goal, software engineers rely on meticulously designed architectural patterns that ensure code clarity and separation of concerns. One such pattern gaining popularity is the Model-View-Intent (MVI) architecture. MVI architecture strictly delineates application responsibilities by dividing it into three key components: model, view, and intent.

This topic remains relevant against the backdrop of increasing demands for software quality. MVI architecture, with its advantages in state management and predictable application behavior, provides developers with the necessary tools to meet these demands, as evidenced by successful industry implementations. The aim of this article is to explore the application of MVI architecture to enhance the testability and maintainability of Android applications.

1. General Characteristics of MVI Architecture

The Model-View-Intent (MVI) architecture represents an advanced approach to developing user interfaces based on the principles of unidirectional data flow and immutable state. This architecture is particularly effective in the context of Android application development, where state management and handling side effects traditionally pose significant challenges.

At the core of MVI lies the idea of representing the user interface as a function of the state: UI = f(state). This means that any change in the user interface results from a change in the application's state. This approach ensures the predictability of UI behavior and simplifies debugging.

A key feature of MVI is the cyclical nature of the interaction between components (Table 1).

Table 1. Elements of MVI Architecture

Element	Description
Model	In architectures such as MVP or MVVM, the model typically represents the data and domain layer, serving as a bridge between the application and a remote data source. In MVI, the model also represents data but in the form of an immutable state. This means the state is updated only in one place – in the business logic, ensuring its immutability in other parts of the application. Thus, the business logic becomes the single source of truth. An example of state representation through a sealed class containing data allows managing states (loading, success, errors) centrally. The render method in the view tracks state changes and updates the user interface accordingly.
View	Reactive programming is used to observe the state returned by the ViewModel. The render() method, implemented in the child classes of BaseActivity or BaseFragment, is used to update the user interface when the state changes. This is especially important when using a shared ViewModel for different fragments, which helps avoid redundant "if-else" statements for state logic management.
Intent	The view observes the state and visualizes its changes. However, the process starts with the view initiating a state change: - A view is created (layout file and fragment/activity). - All view operations are defined: user actions (e.g., click) or actions initiated by the application itself. For instance, the user enters text in the search bar and clicks the search button. This action sends the SearchCharacter intent with the entered text to the ViewModel. The intent is interpreted by the ViewModel into a corresponding action. This is necessary as the same action can be initiated by different intents. Furthermore, different views can create the same intents using the same ViewModel. The action is then processed by the ViewModel and passed to the reducer to determine the new state, which is sent to the view. The reducer, implemented as an extension function, maps the result to the corresponding state. The view, observing the state, updates accordingly.

This cyclic model can be visualized as follows:

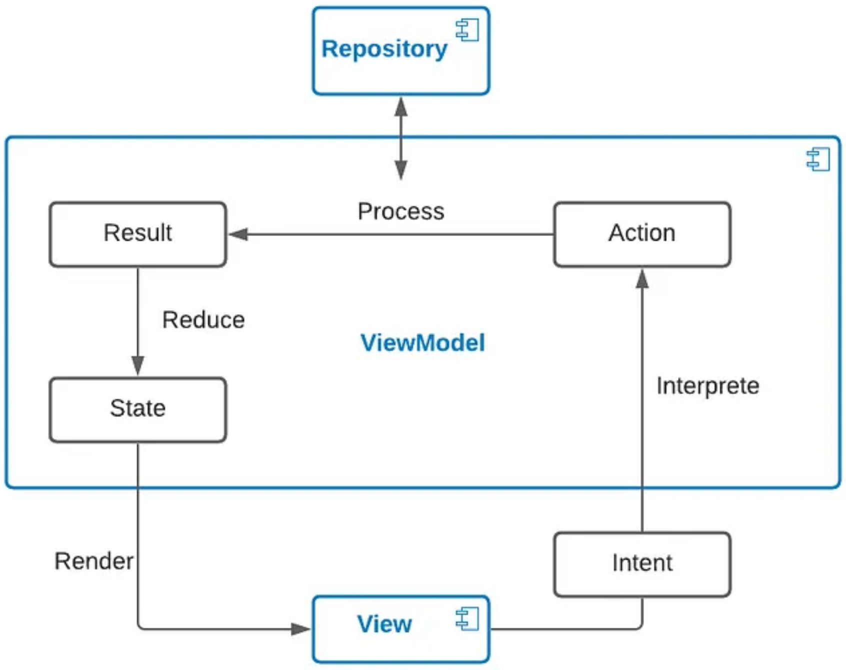

Fig.1.MVI data flow [1]

2. Key Components of MVI Architecture

The MVI architecture consists of the following key components:

a) **Model**: Represents the state of the application. In the context of MVI, the model is immutable, ensuring a unidirectional flow of data and simplifying change tracking.

b) **View**: Responsible for displaying the application's state to the user. In Android, this can be implemented using Activity, Fragment, or Composable functions when using Jetpack Compose.

c) **Intent**: Represents the user's intention to change the application's state. Intents in MVI should not be confused with Android Intents; here, they are a semantic expression of the user's action.

d) **Action**: Derived from Intent, representing a specific action that needs to be performed to change the state.

e) **Result**: The outcome of processing an Action, containing information to update the state.

f) **Reducer**: A pure function that takes the current state and the Result, returning a new state.

g) **Store**: The central component that manages the data flow and the state of the application.

The lifecycle of data in the MVI architecture can be represented by the following diagram (Figure 2).

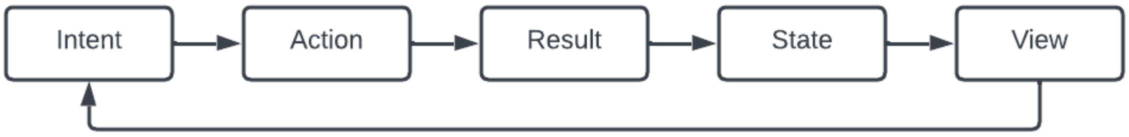

Figure 2 – Data Lifecycle in MVI Architecture

1. A user action generates an Intent.
2. The Intent is transformed into an Action.
3. The Action is processed and produces a Result.
4. The Result is used by the Reducer to create a new State.
5. The new State is displayed in the View.
6. The View responds to user actions, generating new Intents.

This cycle ensures a clear separation of concerns and simplifies tracking changes in the application's state.

In the context of Android development, MVI is particularly effective when combined with reactive tools such as RxJava or Kotlin Flow. These tools allow for efficient handling of asynchronous operations and data flow management, which is critical for creating responsive and reliable mobile applications .

It is important to note that while MVI provides a powerful toolkit for managing state and data flow, its effective application requires careful design and a deep understanding of the principles of functional and reactive programming. When implemented correctly, MVI can significantly enhance the testability and maintainability of Android applications, especially in the context of complex user interfaces and distributed systems.

3. Impact of MVI on Testability of Android Applications

The Model-View-Intent (MVI) architecture has a substantial impact on the testability of Android applications, offering several advantages that contribute to improved software quality and reliability. Let's consider the key aspects of this impact.

3.1 Isolation of Components

MVI architecture ensures a clear separation of responsibilities among components, greatly simplifying their isolation for testing. Each MVI component (Model, View, Intent, Action, Result, Reducer) has a well-defined role and interaction interface, allowing them to be tested independently.

Example of component isolation for testing:

```
class ReducerTest {
  @Test
  fun `when TaskAdded result is processed, new task should be added to state`() {
    val initialState = TaskState(tasks = emptyList())
    val newTask = Task(id = "1", title = "New Task")
    val result = TaskResult.TaskAdded(newTask)

    val newState = taskReducer(initialState, result)

    assert(newState.tasks.contains(newTask))
  }
}
```

In this example, we can easily test the logic of the Reducer without interacting with other system components.

3.2 Predictability of States

One of the key advantages of MVI is the predictability of application states. With unidirectional data flow and immutable states, every change in the application becomes deterministic and easily reproducible.

Data flow diagram in MVI (Figure 3)

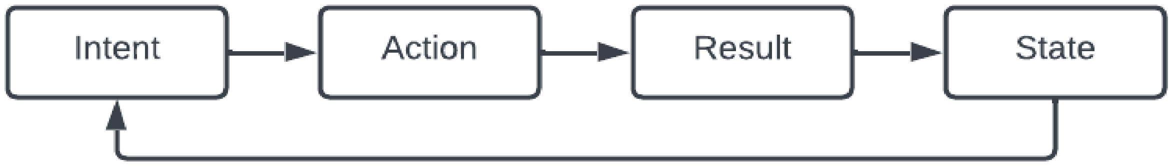

Figure 3 – Data Flow Diagram in MVI

This predictability allows for more reliable tests, as we can precisely determine the expected state after processing a specific Intent or Action.

3.3 Simplification of Unit Testing

MVI architecture promotes the creation of more modular and, therefore, more testable components. Key aspects include:

a) Testing Intents and Actions: Intents and Actions in MVI are simple data objects, making them easy to test. We can verify that Intents are correctly transformed into corresponding Actions.

b) Testing Reducer: The Reducer in MVI is a pure function that takes the current state and a result, returning a new state. This is an ideal scenario for unit testing, as we can easily check that the Reducer produces the expected state for given inputs.

c) Testing Side Effects: In MVI, side effects are typically handled separately from the main data flow, allowing them to be isolated and tested independently.

Example test for a side effect:

```
@Test
fun `when error occurs, error effect should be emitted`() = runTest {
   val actionProcessor = TestActionProcessor()
   val action = TaskAction.LoadTasks

   val results = actionProcessor.processAction(action).toList()

   assert(results.any { it is TaskResult.Error })
}
```

3.4 Enhancement of Integration Testing

MVI also improves integration testing. With clear separation between components and predictable data flow, we can easily create tests that verify the interaction between different parts of the system.

Example of an integration test:

```
@Test
fun `when add task intent is dispatched, new task should appear in view`() {
   val viewModel = TaskViewModel(testRepository)
   val testObserver = viewModel.state.test()
```

```
  viewModel.dispatch(TaskIntent.AddTask("New Task"))

  testObserver.assertValueAt(1) { state ->
      state.tasks.any { it.title == "New Task" }
  }
}
```

3.5 Improvement of UI Testing Efficiency

MVI architecture also positively impacts UI testing. Since the application state is centralized and predictable, we can easily create test scenarios for various UI states.

Example of a UI test using Jetpack Compose:

```
@Test
fun taskListDisplaysCorrectly() {
  val tasks = listOf(Task("1", "Task 1"), Task("2", "Task 2"))
  val state = TaskState(tasks = tasks)

  composeTestRule.setContent {
      TaskList(state = state, onTaskClick = {})
  }

  composeTestRule.onNodeWithText("Task 1").assertIsDisplayed()
  composeTestRule.onNodeWithText("Task 2").assertIsDisplayed()
}
```

In conclusion, the impact of MVI on the testability of Android applications is multifaceted and significant. The architecture promotes the creation of more modular, predictable, and therefore more testable components. This, in turn, leads to higher code quality, fewer bugs, and improved overall application reliability. However, it is important to consider that the effective use of MVI requires careful planning and a deep understanding of the architecture's principles, which can pose certain challenges for teams lacking experience with this approach.

CONCLUSION

The Model-View-Intent (MVI) architecture is a powerful tool for developing scalable and easily maintainable Android applications. Its use allows for a high level of testability and maintainability due to strict separation of concerns and unidirectional data flow. Despite some challenges related to the learning curve and increased boilerplate code, the advantages of MVI far outweigh its drawbacks. In practice, MVI has proven effective, as demonstrated in the development of the Yelp application, where performance was significantly improved, and the number of frozen frames was reduced. Therefore, MVI is an important architectural approach capable of significantly enhancing the quality and stability of modern mobile applications.

REFERENCES

1. The MVI architecture for Android. [Electronic resource] Access mode: https://medium.com/swlh/mvi-architecture-with-android-fcde123e3c4a (accessed 06/20/2024).
2. MVI architecture for Android. [Electronic resource] Access mode: https://www.scaler.com/topics/android/mvi-architecture-android / (accessed 06/20/2024).
3. MVI-graphic editors for Android applications: applications, programs and practical recommendations. [Electronic resource] Access mode: https://www.codetd.com/en/article/15285478 (accessed 06/20/2024).
4. Reactive Applications with Model-View-Intent - Part 2: View and Intent. [Electronic resource] Access mode: http://hannesdorfmann.com/android/mosby3-mvi-2 (accessed 06/20/2024).
5. Yelp has implemented the MVI graphics editor to improve the performance and testability of its Android application. [Electronic resource] Access mode: https://mobilemonitoringsolutions.com/yelp-adopted-the-mvi-architecture-to-improve-performance-and-testability-of-their-android-app / (accessed 06/20/2024).