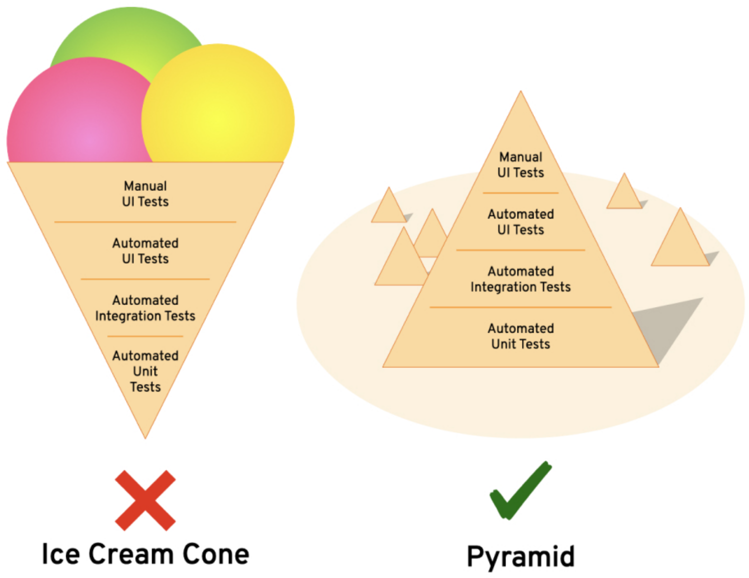
Testing MVI View Models on Android

What is MVI?

MVI (“Model, View, Intent”)

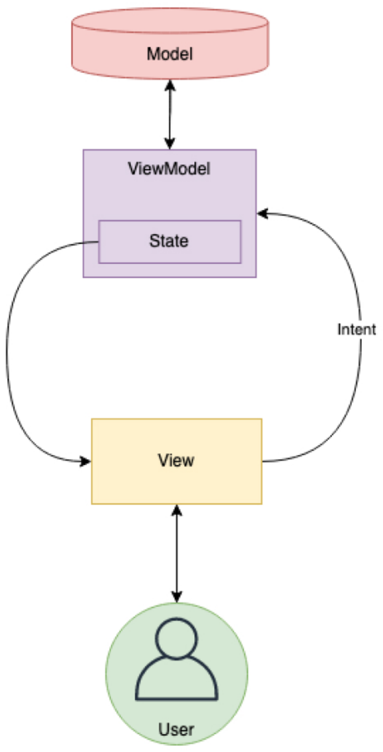
A safer and faster way for testing View Models on Android is to make it a part of the automated integration tests.



How MVI works on Android

MVI (“Model, View, Intent”)

The main concept of MVI is that the View sends events as Intents to the ViewModel. The ViewModel handles these events and communicates with the Model. As a result, the ViewModel updates the View with new states and is then displayed to the user.



In order to test our View Models, we first need to know how our ViewModel looks like. Let’s create one using Coroutines and Flow!

Creating View Models

Step ①: Receiving & Handling Intents

For the base of our MVI-ViewModel we'll first extend the Android ViewModel. First, we add the ability to receive and handle Intents. With this little modification, we get a ViewModel that can react to events from a Fragment, but it can't communicate back to it:

```
abstract class IntentViewModel<Intent> : ViewModel() {

    private val intents = Channel<Intent>()

    fun send(intent: Intent) = viewModelScope.launch { intents.send(intent) }

    protected abstract suspend fun handleIntent(intent: Intent)

    init {
        viewModelScope.launch {
            intents.consumeEach { intent ->
                handleIntent(intent)
            }
        }
    }
}
```

Step ②: Sending States

Next, we'll extend the IntentViewModel with the ability to send states back to the Fragment as a reaction to received intents:

```
abstract class StatefulIntentViewModel<Intent, State>(
    val initialState: State
) : IntentViewModel<Intent>() {

    private var state = initialState

    private val statesBroadcast = BroadcastChannel<State>(1)

    private val stateMutex = Mutex()

    val states = statesBroadcast.asFlow()

    protected suspend fun setState(reducer: State.() -> State) =
        stateMutex.withLock {
            state = state.reducer()
            statesBroadcast.send(state)
        }

    protected suspend fun withState(action: (State).() -> Unit) =
        setState {
            this.apply(action)
        }
}
```

The Ultimate Test (Class)

Next, we'll extend the IntentViewModel with the ability to send states back to the Fragment as a reaction to received intents:

```
abstract class StatefulIntentViewModelTest<
    Intent,
    State,
    ViewModel : StatefulIntentViewModel<Intent, State>
> {
    val testDispatcher = TestCoroutineDispatcher()

    @Before
    fun setup() {
        Dispatchers.setMain(testDispatcher)
    }

    protected fun test(...) { ... }

    @After
    fun cleanUp {
        Dispatchers.resetMain()
        testDispatcher.cleanupTestCoroutines()
    }
}
```

The Ultimate Test (Class)

Next, we'll extend the IntentViewModel with the ability to send states back to the Fragment as a reaction to received intents:

```
abstract class StatefulIntentViewModel<Intent, State>(
    val initialState: State
) : IntentViewModel<Intent>() {

    private var state = initialState

    private val statesBroadcast = BroadcastChannel<State>(1)

    private val stateMutex = Mutex()

    val states = statesBroadcast.asFlow()

    protected suspend fun setState(reducer: State.() -> State) =
        stateMutex.withLock {
            state = state.reducer()
            statesBroadcast.send(state)
        }

    protected suspend fun withState(action: (State).() -> Unit) =
        setState {
            this.apply(action)
        }
}
```