Building Modern Apps for Android

Compose, Kotlin, Coroutines, Jetpack and the best tools for native development.

Spanish version

Yair Carreno

Building Modern Apps for Android

Compose, Kotlin, Coroutines, Jetpack, and the best tools for native development.

Yair Carreno

Este libro está a la venta en http://leanpub.com/building-modern-apps-for-android-spanish

Esta versión se publicó en 2022-07-04

Índice general

Capítulo 1: Principios de diseño

"State" es el corazón en las vistas declarativas

El primer paradigma que dedemos tener claro cuando se diseña vistas declarativas a través de frameworks como *Compose* o *SwiftUI* es el "State".

Un componente UI es la combinación entre su representación gráfica (*View*) y su estado (*State*).

Toda aquella propiedad o dato que pueda cambiar en el componente UI puede ser representado como un estado. Por ejemplo, en un componente UI de tipo `TextFiel`, el texto ingresado por el usuario es una variable que puede cambiar, por lo tanto, `value` es una variable que podría ser representada como un estado (*name*), así como se muestra en el siguiente *code snippet 1.1*.

Code snippet 1.1

```
TextField(
    label = { Text("User name") },
    value = name,
    onValueChange = onNameChange
)
```

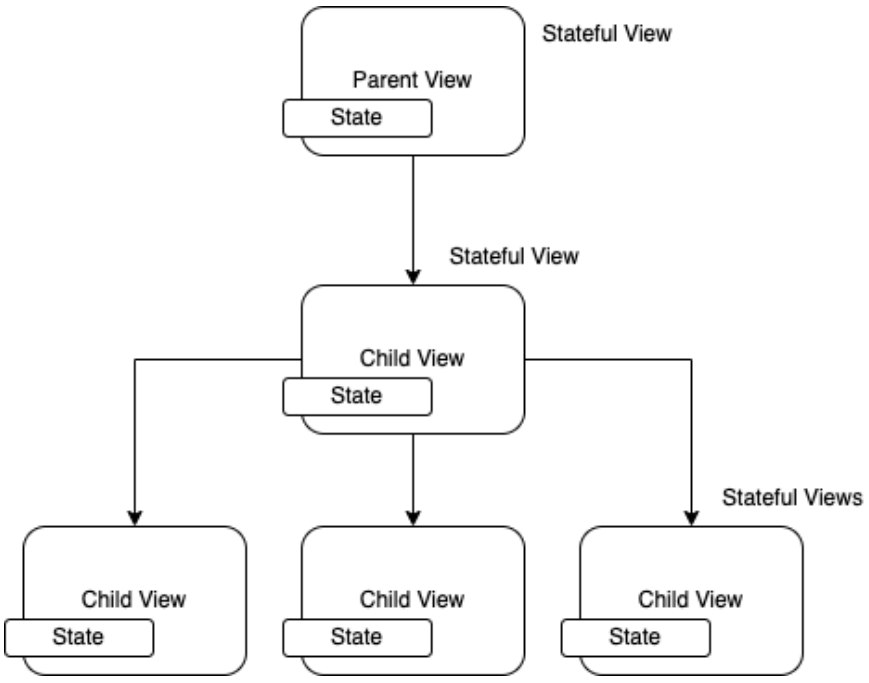

Figure 1.1 Declarative Views's Hierarchy

La pantalla (*Screen*) de una aplicación móvil, puede estar conformada por una jerarquía de vistas, así como se muestra en la figura 1.1.

Cada vista, a su vez, puede contener múltiples variables de estado. Por ejemplo, todas las vistas de la figura 1.1 contienen un estado.

A las vistas que contienen o dependen de un estado se les denomina *Stateful Views* y a aquellas vistas que carecen o no tienen dependencia de algún estado se les conoce como *Stateless Views*.

Tanto *Google* como *Apple*, recomiendan como una buena práctica diseñar, en la medida de lo posible, vistas de tipo *stateless* por las siguientes ventajas que produce usar este tipo de vistas:

- Son vistas que pueden ser reutilizadas.
- Permiten delegar el manejo de estado a otros componentes.
- Son funcionales y evitan *side-effects*.

De acuerdo a dichas recomendaciones, es importante que el diseño sea orientado al uso de vistas sin estados y convertir aquellas vistas de tipo *stateful* a vistas de tipo *stateless*.

¿Esto cómo se logra? En la siguiente sección lo averiguaremos.

Aplicando “State hoisting” para delegar estados

State hoisting es una técnica para convertir vistas de tipo *stateful* (con estado) a vistas de tipo *stateless* (sin estado). Esto se logra a través de la inversión de control, así como se muestra en el siguiente *code snippet 1.2*:

Code snippet 1.2

```
// This is a Stateful View
@Composable
fun OrderScreen() {

    var name by remember { mutableStateOf("") }
    var phone by remember { mutableStateOf("") }

    ContactInformation(
        name = name,
        onNameChange = { name = it },
        phone = phone,
        onPhoneChange = { phone = it })
}

// This is a Stateless View
@Composable
fun ContactInformation(
    name: String,
    onNameChange: (String) -> Unit,
    phone: String,
```

```
    onPhoneChange: (String) -> Unit
) {

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(8.dp),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        TextField(
            label = {
                Text("User name")
            },
            value = name,
            onValueChange = onNameChange
        )
        Spacer(Modifier.padding(5.dp))
        TextField(
            label = {
                Text("Phone number")
            },
            value = phone,
            onValueChange = onPhoneChange
        )
        Spacer(Modifier.padding(5.dp))
        Button(
            onClick = {
                println("Order generated for $name and phone $phone")
            },
        ) {
            Text("Pay order")
        }
    }
}
```

En el *code snippet 1.2*, el control de los estados *name* y *phone* se le delega a la vista *OrderScreen*, de tal forma que la vista *ContactInformation* se despreocupa del estado de sus datos y podría ser reutilizada por otras vistas.

OrderScreen se convierte en *Stateful* y *ContactInformation* en *Stateless*.

Code snippet 1.3

```
@Composable
fun OrderScreen() {

    // States name and phone
    var name by remember { mutableStateOf("") }
    var phone by remember { mutableStateOf("") }

    ContactInformation(
        name = name,
        onNameChange = { name = it },
        phone = phone,
        onPhoneChange = { phone = it })
}

@Composable
fun ContactInformation(
    name: String,
    onNameChange: (String) -> Unit,
    phone: String,
    onPhoneChange: (String) -> Unit,
    payOrder: () -> Unit
) {
    // Code omitted for simplicity
}
```

En el ejemplo *Code snippet 1.3*, la inversión de control se logra implementar a través de *Higher order functions* permitiendo pasar como argumentos a la vista *ContactInformation* las definiciones de los estados y operaciones.

Definiendo el "Source of truth", ¿Quién es responsable de entregar los estados?

Primero aclaremos que es el término **Source of truth**.

Source of truth, hace referencia a la fuente fiable que provee los datos que requiere una vista para ser presentados en pantalla y con los cuales estará interactuando el usuario.

En nuestro análisis, **datos** está estrechamente relacionado a **estados**. Las vistas usan los estados como mecanismo para recibir la información (datos) que necesitan para hacer su trabajo.

En la figura 1.1 veíamos como los estados se encuentran en cada una de sus respectivas vistas. Eso significa que en dicho diagrama cada una de las vistas es un *source of truth*.

Incluso la variable `name` del componente UI `TextFiel` del que hablábamos antes (*code snippet 1.1*), podría ser un estado y, por tanto, también es un **source of truth**.

¿Es bueno tener tantos "Source of truth" en una jerarquía de vistas?

La respuesta es **No**.

Lo recomendable es que se limiten los *source of truth* a un único componente (o al mínimo posible), así se puede tener mayor control sobre flujo y evitar inconsistencias de estados.

Tener una única *source of truth* claramente definida también ayuda a la implementación correcta del patrón de diseño *Unidirectional Data Flow*[1], el cual es el patrón que promueven las vistas declarativas como *Compose* o *SwiftUI*.

En la sección Entendiendo el flujo de los datos se hablará un poco más de este patrón.

¿Y como reduzco el número de "source of truth" en mi diseño?

Reduciendo el número de vistas *Stateful* a través de la técnica *State hoisting* explicada anteriormente y centralizando el estado en una vista. Generalmente, la delegada es la vista con mayor nivel jerárquico, es decir, una vista padre.

Por ejemplo, en la figura 1.2 se muestra que solo existe una única *Source of truth* y es la vista padre.

Las vistas hijas (*Child Views*) por un lado, únicamente se encargan de propagar los **eventos** recibidos por la interacción con el usuario y, por otro lado, reciben los **estados** que harán que se renderice la vista (*Recomposition*[2]) para reflejar los cambios de la UI.

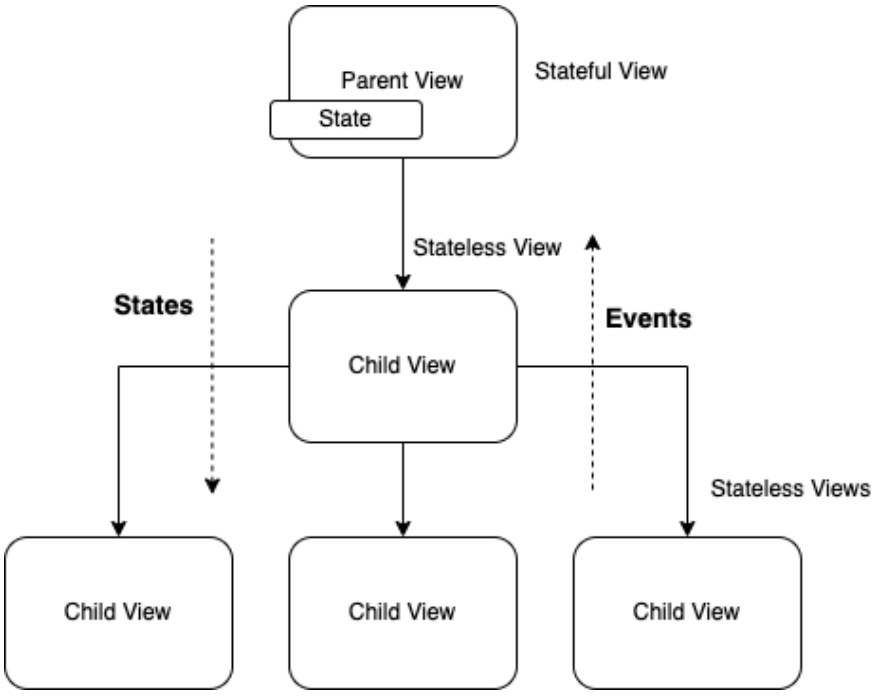

Figure 1.2 Delegating state handling to a View

[1] Manage state with Unidirectional Data Flow
[2] Recomposition

¿Existe otra opción que no sea delegarle toda la responsabilidad de manejo de estados a solo una vista?

La respuesta es **Si**.

Una mejor opción es delegarle dicha responsabilidad a un *State Holder* o un *ViewModel* que cumpla ese rol[3]. Veamos mas detalle en la siguiente sección.

ViewModel como Source of truth

Para evitar que la vista sea saturada de responsabilidades, se recurre a otro componente para que se encargue de administrar el estado. El componente adecuado para este propósito es el conocido *ViewModel*.

Como se muestra en la figura 1.3, trasladar los estados de la *Vista* al *ViewModel* genera la separación de responsabilidades, permitiendo que la lógica de presentación y sus efectos sobre el estado puedan ser centralizadas.

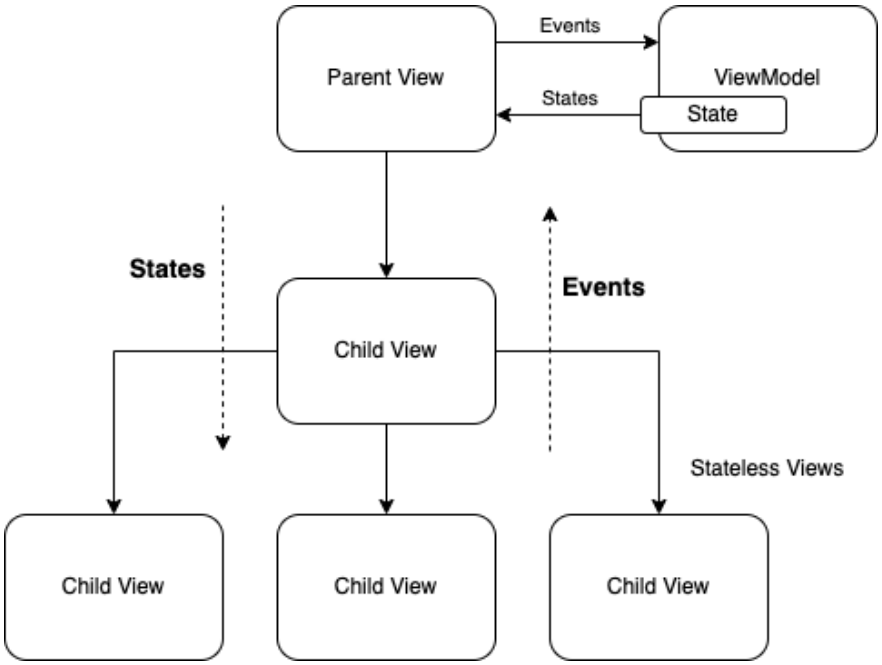

Figure 1.3 Delegating state handling to a ViewModel

Aun cuando este componente (*ViewModel*) es opcional en las implementaciones, recomiendo arduamente que sea usado, ya que provee de muchas ventajas a la implementación, como por ejemplo el efectivo manejo del ciclo de vida entre los datos y las vistas. Para mayor información sobre este componente de arquitectura recomiendo revisar la documentación oficial de Google sobre *ViewModels*[4].

La comunicación entre *View* y *ViewModel* consta de tan solo dos tipos de mensajes, los eventos(*Events*) y los estados (*States*):

[3] Managing state in Compose
[4] ViewModel Overview

- **Los eventos** son las acciones notificadas al *ViewModel* por cualquier *Vista* o *Sub-vista* como consecuencia de una acción o interacción del usuario con los componentes UI.
- **Los estados** son la representación de la información (*data*) que entrega el *ViewModel* a las *Vistas* para su respectiva interpretación gráfica.

La función principal del *ViewModel* es recibir los eventos enviados desde las vistas, interpretarlos, aplicar lógica de negocio y transformarlos en estados para ser entregados nuevamente a las vistas.

La función de la *Vista* es recibir los estados enviados por el *ViewModel* y traducirlos a su representación gráfica a través de la recomposición o renderizado de las vistas.

Hasta aquí, teniendo un poco más clara la responsabilidad de cada componente y los mensajes entre ellos, analicemos a continuación lo que ocurre con el flujo de la información.

Entendiendo el flujo de los datos, “Unidirectional Data Flow”

Si realizamos una simplificación del diagrama de la figura 1.3, el resultado va a hacer el siguiente diagrama de la figura 1.4

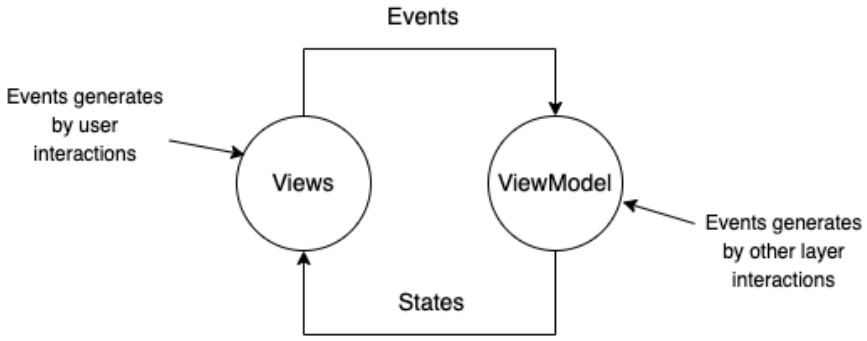

Figure 1.4 Unidirectional Data Flow

Claramente, es un **ciclo de mensajes** entre la *Vista* y el *ViewModel*. El flujo de la información solo sigue una única dirección, de allí el nombre del patrón *Unidirectional Data Flow*.

Los factores externos que pueden inyectar eventos al ciclo son interacciones del usuario, como por ejemplo un *scroll* en una lista, un *click* en un botón y las interacciones con otras capas de la aplicación como por ejemplo una respuesta del *Repository*, o la repuesta de un temporizador en *background*, o quizás la llegada de un *push Notification*.

El ciclo no puede ser interrumpido, ya que cualquier interrupción o demora inducida se reflejará en una pobre experiencia de usuario. El usuario percibirá la aplicación lenta, bloqueada y de mala calidad.

Por ello, el diseño debe tener en mente las siguientes reglas hasta donde sea posible:

- El *Composable* que define la vista debe ser idempotente y funcional.

- En la vista, no puede haber tareas que retarden el ciclo. Toda tarea que requiera un proceso extenso debe ser delegado al *ViewModel* para que a través de *Reactive programming* y usando *Flow Coroutines* realice de forma asincrónica dichas tareas.

Ahora que se tiene mayor idea del flujo de los datos y los mensajes intercambiados entre "View" y "ViewModel", es lógico preguntar:

¿Cómo se implementa el canal de comunicación entre View y ViewModel? Lo veremos a continuación.

Conectando los componentes "View" y "ViewModel"

Analizando la figura 1.4, se identifica claramente los dos tipos de canales de comunicación que se requieren implementar.

El primer canal es el de los eventos (*Eventos*) que va en sentido *View —> ViewModel*. Para esta implementación, solo se requiere que el *ViewModel* exponga las operaciones públicas que puedan ser llamadas por *View*, así como se muestra en el siguiente *code snippet 1.4*.

Code snippet 1.4

```
//UI's Events
fun onNameChange(): (String) -> Unit = {
    name = it
}

fun onPhoneChange(): (String) -> Unit = {
    phone = it
}
```

El segundo canal es el de los estados (*States*) que va en sentido *ViewModel –> View*.

¿Cómo se entera la UI que el estado ha cambiado?

Observando los estados. Para observar los estados primero, el *ViewModel* debe exponerlos hacia la UI a través del componente *mutableStateOf* así:

Code snippet 1.5

```
// UI's states
var name by mutableStateOf("")
    private set
var phone by mutableStateOf("")
    private set
```

El componente *mutableStateOf* no solo permitirá exponer el estado a la vista, sino que además permitirá que la vista puede susbcribirse para que le sea notificado cualquier cambio en dicho estado.

Veamos la implementación completa tanto del *ViewModel* como de la vista (*Composable*):

Code snippet 1.6: ViewModel

```
class OrderViewModel : ViewModel() {

    // UI's states
    var name by mutableStateOf("")
        private set
    var phone by mutableStateOf("")
        private set

    //UI's Events
    fun onNameChange(): (String) -> Unit = {
        name = it
    }

    fun onPhoneChange(): (String) -> Unit = {
        phone = it
    }

    fun payOrder(): () -> Unit = {
        println("Order generated for $name and phone $phone")
    }
}
```

Code snippet 1.7: View (Composables)

```
@Composable
fun OrderScreen(viewModel: OrderViewModel = viewModel()) {

    ContactInformation(
        name = viewModel.name,
        onNameChange = viewModel.onNameChange(),
        phone = viewModel.phone,
        onPhoneChange = viewModel.onPhoneChange(),
        payOrder = viewModel.payOrder()
    )
}

@Composable
fun ContactInformation(
    name: String,
    onNameChange: (String) -> Unit,
    phone: String,
    onPhoneChange: (String) -> Unit,
    payOrder: () -> Unit
) {

```

```
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(8.dp),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        TextField(
            label = {
                Text("User name")
            },
            value = name,
            onValueChange = onNameChange
        )
        Spacer(Modifier.padding(5.dp))
        TextField(
            label = {
                Text("Phone number")
            },
            value = phone,
            onValueChange = onPhoneChange
        )
        Spacer(Modifier.padding(5.dp))
        Button(
            onClick = payOrder,
        ) {
            Text("Pay order")
        }
    }
}
```

Hasta el momento, hemos visto que los estados, tales como *name* y *phone*, son representación de una variable de tipo *String*, es decir, el estado está representando una **variable primitiva**. Sin embargo, la representación del estado puede ser extendiendo a componentes (*components*) y pantallas (*screens*).

En la siguiente sección, veremos otras opciones para la representación de los estados.

Estructuras que pueden ser representadas como estados

En *Compose* y en general en las vistas declarativas, los estados podrían representar diferentes tipos de estructuras UI, así como se muestra en la siguiente figura 1.5.

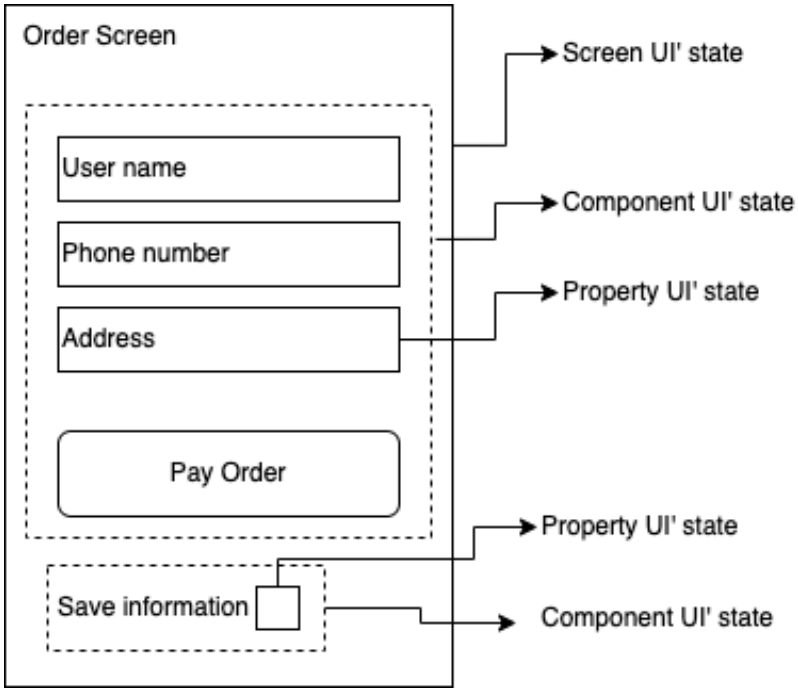

Figure 1.5 Structures represented by states

- **Property UI's state**: Son variables primitivas representadas como estados. En la figura 1.5, los campos de entrada de texto como *name*, *phone* o *address* son de este tipo.
- **Component UI's state**: Representa los estados asociados a un componente que agrupa elementos UI relacionados. Por ejemplo, en la pantalla *OrderScreen* podría existir un componente llamado *ContactInformationForm* que agrupa los datos requeridos como información de contacto. Este componente podría tener estados de *Name Value Changed*, *Phone Value Changed*, *Success validated*.
- **Screen UI's state**: Representa los estados asociados a una pantalla (*Screen*) y que pueden ser tratados como estados absolutos e independientes, por ejemplo una pantalla llamada *OrderScreen* podría tener los estados: *Loading*, *Loaded successfully* o *Load failed*.

Ahora veamos que opciones de implementación existen en *Android* y *Kotlin* para definir estos tipos de estados.

Property UI's state

Son estados declarados a partir de una variable de tipo primitivo, tales como *String*, *Boolean*, *List*, *Int*, entre otros.

Si es declarado en ViewModel (*ViewModel* as *Source of truth*), su definición podría ser así:

Code snippet 1.8

```
var name by mutableStateOf("")
    private set

var phone by mutableStateOf("")
    private set

var address by mutableStateOf("")
    private set

var payEnable by mutableStateOf(false)
    private set
```

Si es declarado en View (*View* as *Source of truth*), su definición en el *Composable* podría ser así:

Code snippet 1.9

```
var name by remember { mutableStateOf("") }
var phone by remember { mutableStateOf("") }
var address by remember { mutableStateOf("") }
var payEnable by remember { mutableStateOf(false) }
```

remember es un composable que permite mantener temporalmente el estado de la variable durante la recomposición. Al ser un *Composable*, esta propiedad puede ser definida únicamente en vistas declarativas, es decir, en funciones *Composables*.

Siempre recordar que para usar la delegación a través de "by", es necesario importar:

Code snippet 1.10

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
```

En los ejemplos anteriores, solo hemos hablado de representar propiedades o variables a través de estados usando el componente *mutableStateOf*. Sin embargo, también es posible que **flujos de datos** puedan ser representados como estados y ser observados por los *Composables*. Estas opciones adicionales están relacionadas con *Flow*, *LiveData* o *RxJava*. En el Capítulo 7: Implementando "Features" veremos varios ejemplos usando *StateFlow*.

Component UI's state

Cuando se tiene un conjunto de elementos UI relacionados entre sí, sus estados podrían ser agrupados y consolidados en una única estructura o componente UI con un único estado.

En la figura 1.5 por ejemplo, los elementos *User name*, *Phone number*, *Address* e incluso el botón *Pay Order*, podrían ser agrupados en un solo componente UI y representarle los estados en un único estado llamado por ejemplo *FormUiState*.

Code snippet 1.11

```
data class FormUiState(
    val nameValueChanged: String = "",
    val phoneValueChanged: String = ""
    val addressValueChanged: String = ""
)

val FormUiState.successValidated: Boolean get() = nameValueChanged.length > 1
                                                  && phoneValueChanged.length > 3
```

En este caso, modelar múltiples estados en una clase consolidada de estados funciona muy bien, ya que las variables están relacionadas e incluso definen el valor de otras variables, por ejemplo, esto es lo que ocurre con la variable *successValidated*, la cual depende de las variables *nameValueChanged* y *phoneValueChanged*.

Consolidar los estados agrega ventajas a la implementación, centraliza el control y organiza el código. Será la técnica a la cual se recurirá con mayor frecuencia en la implementación.

Screen UI's state

Si lo que se requiere es modelar estados que puedan ser totalmente independientes, pero que forman parte de la misma familia, se podría usar la siguiente definición:

Code snippet 1.12

```
sealed class OrderScreenUiState {
    data class Success(val order: Order): OrderScreenUiState()
    data class Failed(val message: String): OrderScreenUiState()
    object Loading: OrderScreenUiState()
}
```

Este tipo de implementación es útil cuando se trabaja con estados absolutos y excluyentes, es decir, o se tiene un estado o se tiene otro estado, pero no ambos a la vez.

Generalmente, pantallas sencillas de este tipo como por ejemplo *OnboardignScreen*, *ResultScreen* pueden ser modelas con estos estados.

Cuando la pantalla es más compleja y contiene muchos elementos UI que operan de forma independiente y múltiples relaciones entre ellos, recomiendo al lector inclinarse por la definición de estados con las técnicas *Property UI' state* y *Component UI' state*.

Modelando y agrupando eventos

Volviendo al ejemplo de la pantalla *OrderScreen*, ahora analizaremos el modelado de los *Events* y como agruparlos de forma similar como se hizo con los *States*.

Considere una pantalla como la que muestra en la siguiente figura 1.6:

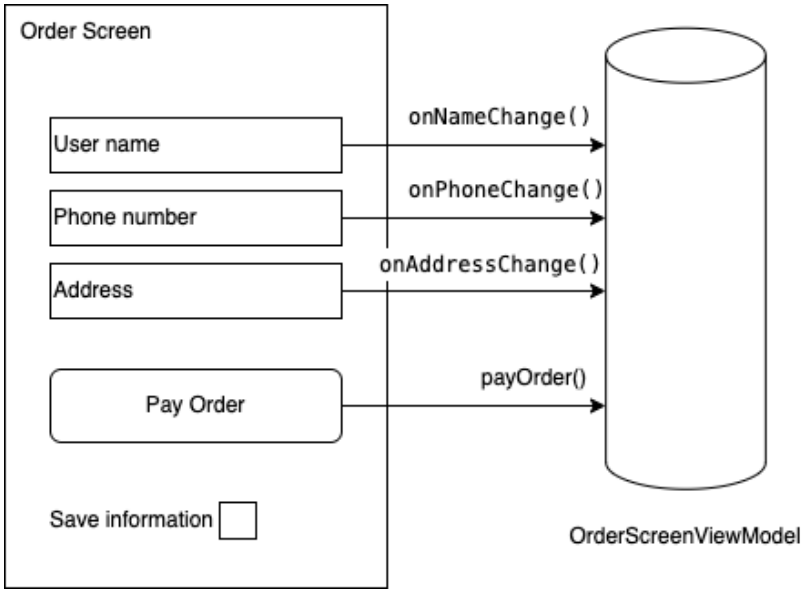

Figure 1.6 Multiple events

El *ViewModel* expone a la vista cuatro operaciones (eventos) cada una de las cuales es utilizado por un elemento UI de *View*.

Analizando, los cuatro eventos están relacionados con un formulario para ingreso de información de contacto del usuario, luego tiene sentido pensar en agruparlos en un único tipo de evento, así como se muestra en la siguiente figura 1.7:

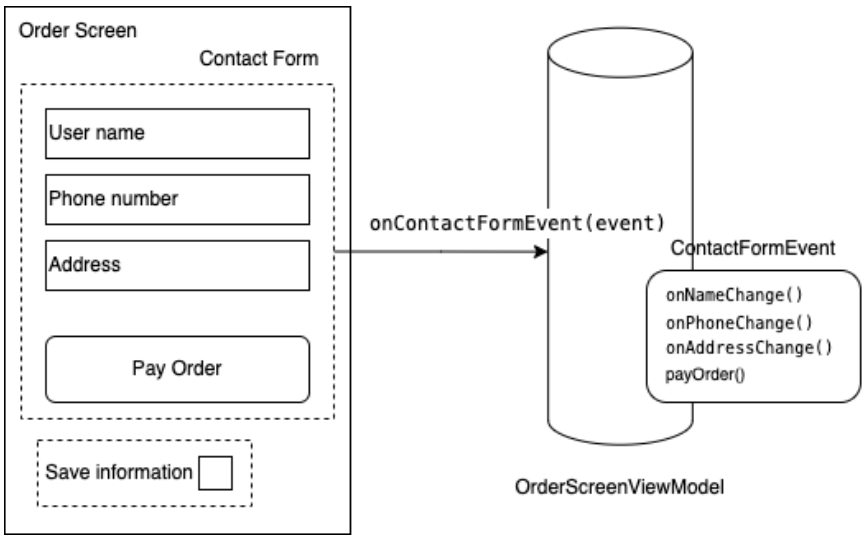

Figure 1.7 Grouping events

La implementación para representar los diferentes tipos de eventos podría ser así:

Code snippet 1.13

```
sealed class ContactFormEvent {
    data class OnNameChange(val name: String): FormUiEvent()
    data class OnPhoneChange(val phone: String): FormUiEvent()
    data class OnAddressChange(val address: String): FormUiEvent()
    object PayOrder: FormUiEvent()
}
```

Para finalizar, no hay que ser tan estrictos a la hora de simplificar los estados o los eventos. Es necesario analizar las ventajas y desventajas de cada aprovechamiento y tomar las decisiones que correspondan.

Para aquellos componentes UI que estén relacionados hace mucho sentido tenerlos agrupados y algunos otros elementos transversales será más saludable dejarlos independientes.

Resumen

En este primer capítulo hemos hecho un repaso por los principales conceptos usados en el desarrollo moderno de aplicaciones *Android*.

Conceptos tales como *States and Events*, *State hoisting*, *Source of truth*, *Unidirectional Data Flow* son indispensables entenderlos antes de comenzar con una implementación a través de *Jetpack Compose*, *ViewModels* y otras herramientas de arquitectura disponibles para *Android*. Esta ha sido la razón por la cuál hemos iniciado con dichos conceptos en este primer capítulo.

En los capítulos posteriores, entraremos a las definiciones de arquitectura y diseño en una aplicación móvil, para lo cual usaremos como referencia, los conceptos expuestos en este capítulo.

Posteriormente será implementada una aplicación móvil llamada “Order Now” usando como concepto el *e-commerce*. Esta aplicación tendrá las principales partes de un e-commerce tales como carrito de compras, lista de productos, proceso de *checkout*, entre otros. Todo esto para introducir al lector a una experiencia de diseño y desarrollo lo más cercano a una aplicación real y productiva.

Pero antes, aplicaremos los conceptos aprendidos en este capítulo en la implementación de un sencillo formulario.

Ese será el tema del siguiente capítulo descrito a continuación.

Capítulo 3: OrderNow, A Real Application

Sobre la aplicación

OrderNow es un ejemplo de *Minimum Viable Product (MVP)* de una aplicación móvil de tipo *e-commerce* que diseñaremos e implementaremos a lo largo de este libro. Usaremos dicha aplicación a manera de ejemplo para aplicar los conceptos aprendidos en cada capítulo del libro.

Implementar una solución *e-commerce*, nos acercará a los retos que demanda una aplicación real y productiva.

Las siguientes son las principales funcionalidades del *e-commerce* que desarrollaremos en *OrderNow*:

- Presentar un listado de categorías.
- Presentar un listado de productos por categorías.
- Presentar el detalle de un producto específico.
- Gestionar productos (agregar o eliminar) en un carrito de compras.
- Ver el listado de los productos seleccionados para compra.
- Realizar el diligenciamiento de información y datos para realizar la compra (checkout).
- Simular el proceso de pago.

Pantallas

Las pantallas relacionadas a diferentes funcionalidades serían:

- Home
- Listado de productos
- Detalle del producto
- Carrito de compras (Cart)
- Proceso de compras (Checkout)
- Simulación del pago

Home, Product List and Product detail

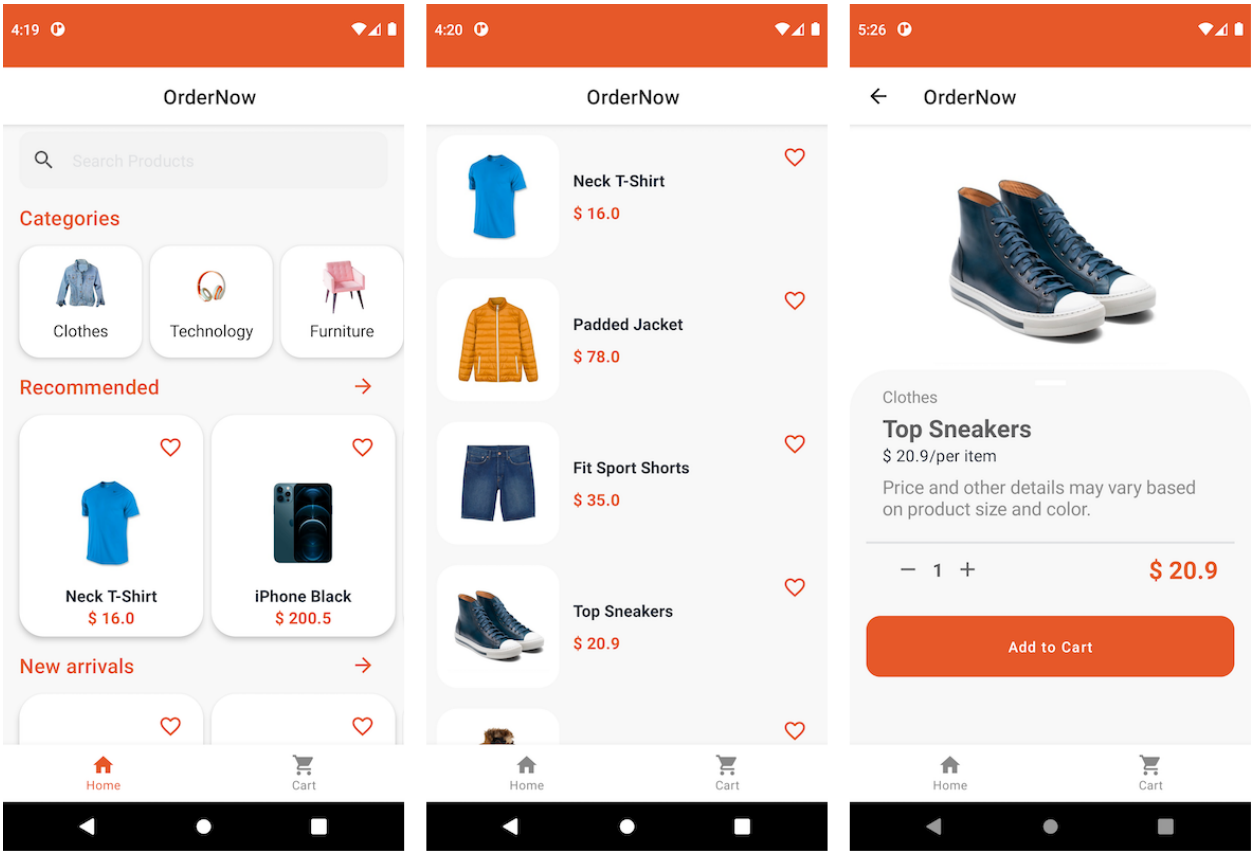

Figure 3.1 Screenshots: Home, Product List and Product detail

Cart and Checkout

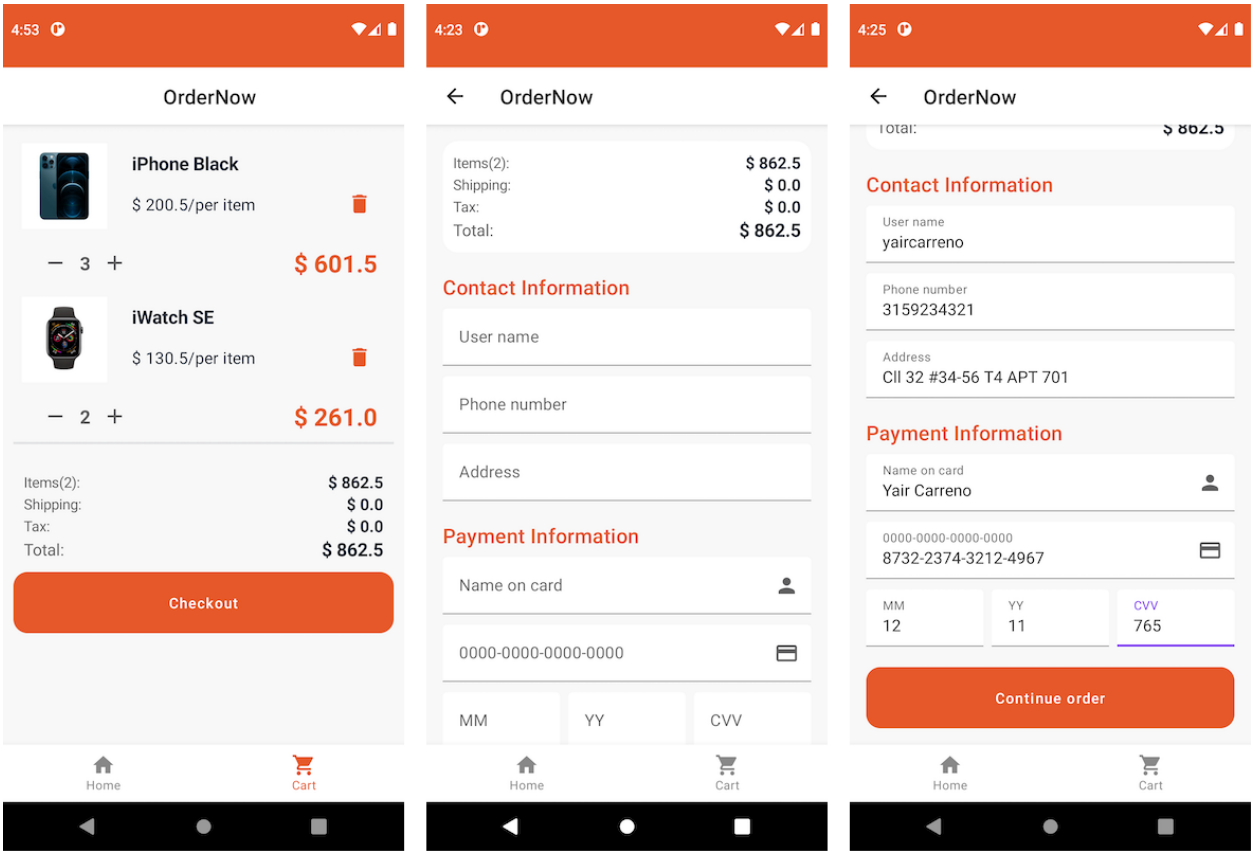

Figure 3.2 Screenshots: Cart and Checkout

Place Order

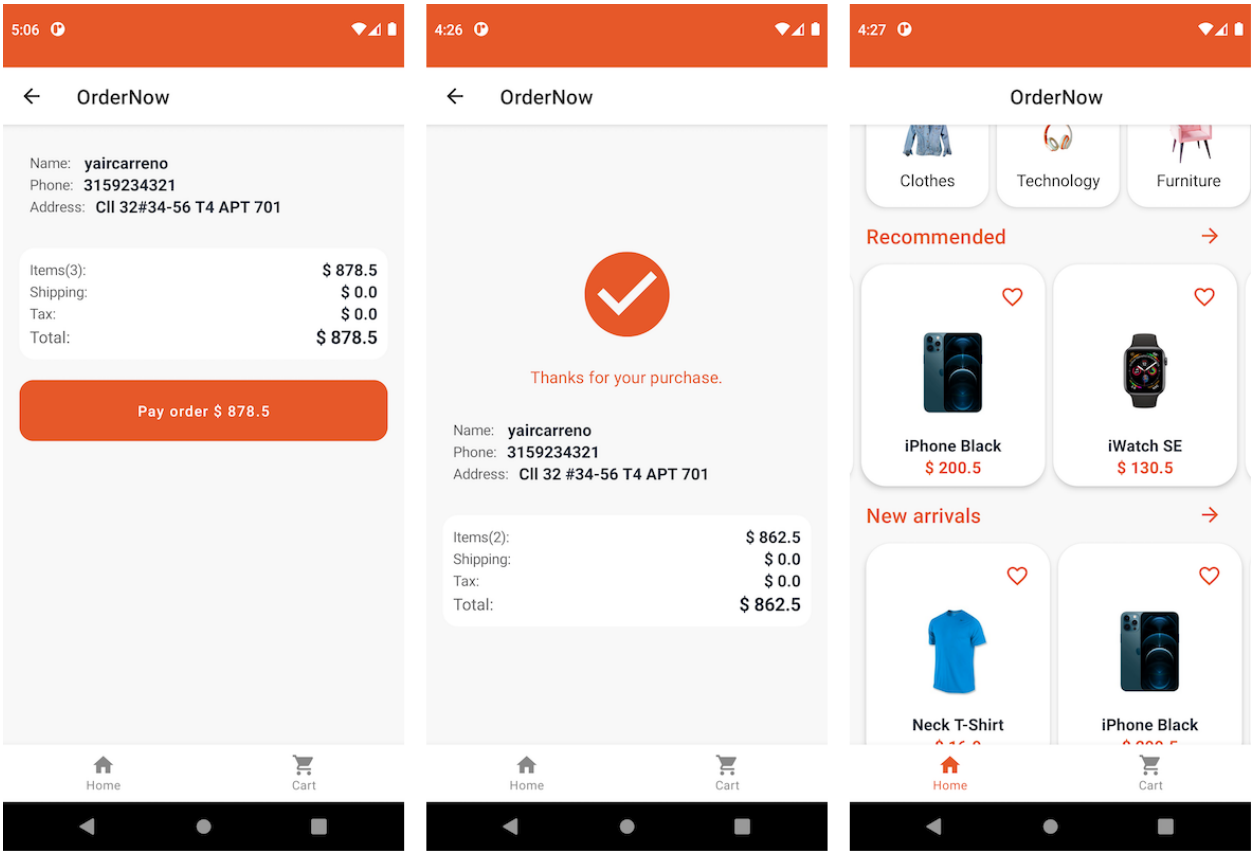

Figure 3.3 Screenshots: Place Order

Ficha técnica

Este es un resumen de las características técnicas de *OrderNow* para que el lector tenga presente las herramientas y guía de diseño que serán empleadas en la implementación.

Recordemos que esta es una propuesta de implementación. El lector estará en su libre decisión de reemplazar o incluir alguna otra herramienta con la que tenga experiencia o se sienta cómodo para trabajar.

Guía de Diseño y Arquitectura

En los capítulos llamados Principios de diseño y Arquitectura de Aplicación, se documentan las guías de diseño y arquitectura, es decir, el *Minimum Viable Architecture (MVA)*[5], que será empleada para el desarrollo de *OrderNow*.

[5] A Minimum Viable Product Needs a Minimum Viable Architecture

Arquitectura de componentes

- **Compose**[6]: Será el framework usado para implementación de vistas declarativas en nuestra capa de presentación.
- **ViewModel**[7]: Componente de Arquitectura en la capa de presentación que usaremos para encapsular lógica de negocio.
- **Hilt**[8]: Será el gestor e inyector de dependencias usado en nuestra aplicación.
- **Flow**[9]: Usaremos *Flow Coroutines* para la programación reactiva en nuestra aplicación. Flow permitirá que los mensajes, ya sean sincrónicos o asincrónicos entre los componentes del *App* sean realizados de la forma más óptima posible.
- **Navigation**[10]: Este componente de Arquitectura será usado para implementar la navegación a través de las diferentes pantallas de nuestra aplicación.

Dependencias

- Coil[11]: Librería que usaremos para cargar imágenes remotas o locales en nuestra APP, a través de *Kotlin* y con soporte para *Jetpack Compose*.

Por fuera del alcance

Algunos temas son excluidos del contenido del libro, no por ser menos importantes si no para lograr acotar el alcance y cumplir metas concretas.

Intentar cubrir todos los tópicos relacionados en una aplicación Android podría extender demasiado el contenido y desviarnos de los conceptos principales que en un inicio se deben tener claros.

No se incluye y quedan por fuera del alcance del ejemplo de *MVP* las siguientes capacidades:

- Guía de Diseño UI/UX.
- Componentes de Autenticación y Autorización de la aplicación.
- Testing.
- Otras dimensiones de pantallas diferentes a *smartphones*.
- Accesibilidad.

Resumen

En este corto capítulo, se describe de forma general, la aplicación *OrderNow* y las funcionalidades que serán implementadas a manera de ejemplo en este libro.

También se presenta un resumen de las tecnologías y componentes empleados en la implementación para que el lector no solo disponga del código fuente y trate de adivinar como fue construido, si no por el contrario conozca perfectamente cada decisión tomada tanto a nivel de diseño como a nivel de implementación.

[6] Jetpack Compose
[7] ViewModel Overview
[8] Hilt
[9] Kotlin flows on Android
[10] Navigating
[11] Coil and Jetpack Compose