



Arquitectura en Android

Patrones de arquitectura en Android (Parte I)

Utilizar patrones de arquitectura escalables para la construcción de una aplicación Android de acuerdo a los requerimientos

- Unidad 1: Kotlin para el desarrollo de aplicaciones.
- Unidad 2: Ciclo de vida de componentes.
- Unidad 3: Arquitectura en Android.
- Unidad 4: Programación asíncrona en Android.



Te encuentras aquí



¿Qué aprenderás en esta sesión?

- *Distinguir patrones de diseño sugeridos y sus características para el desarrollo de un aplicativo Android.*

Cuando ves el código
fuente de una app, ¿has
puesto atención a la forma
en que están distribuidas
las clases y los paquetes?



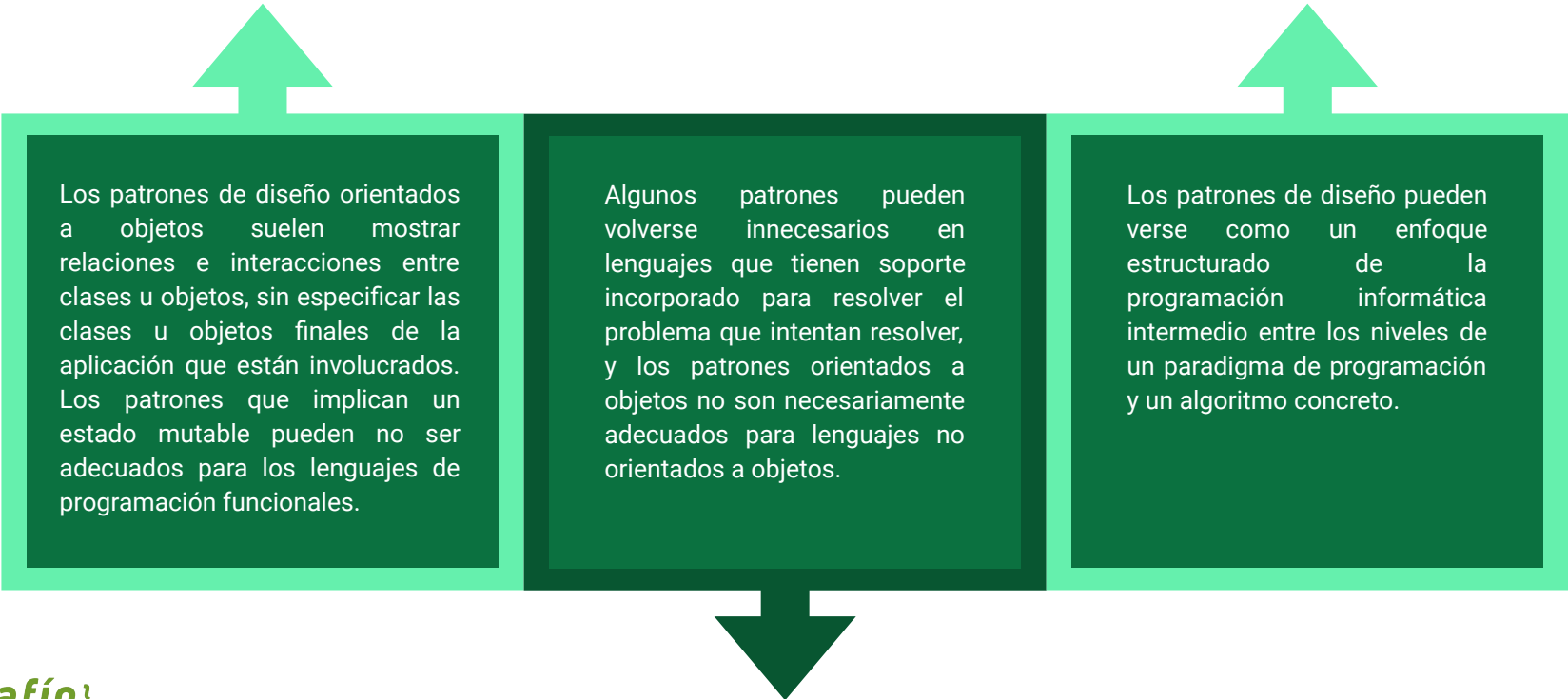
/* Qué son los patrones de diseño */

¿Qué son los patrones de diseño?

- Es una solución general y reutilizable para un problema común dentro de un contexto dado en el diseño de software.
- El patrón de diseño no es un diseño terminado que pueda transformarse directamente en código fuente o máquina.
- Más bien es una descripción o plantilla sobre cómo resolver un problema que se puede usar en muchas situaciones diferentes.
- Los patrones de diseño son mejores prácticas formalizadas que el programador puede usar para resolver problemas comunes al diseñar una aplicación o sistema.



¿Qué son los patrones de diseño?



Los patrones de diseño orientados a objetos suelen mostrar relaciones e interacciones entre clases u objetos, sin especificar las clases u objetos finales de la aplicación que están involucrados. Los patrones que implican un estado mutable pueden no ser adecuados para los lenguajes de programación funcionales.

Algunos patrones pueden volverse innecesarios en lenguajes que tienen soporte incorporado para resolver el problema que intentan resolver, y los patrones orientados a objetos no son necesariamente adecuados para lenguajes no orientados a objetos.

Los patrones de diseño pueden verse como un enfoque estructurado de la programación informática intermedio entre los niveles de un paradigma de programación y un algoritmo concreto.

/* Tipos de patrones de diseño */

MVC

El patrón MVC (Model View Controller) sugiere dividir el código en 3 componentes. Al crear la clase/archivo de la aplicación, el desarrollador debe categorizarlo en una de las siguientes tres capas:

Model

Este componente almacena los datos de la aplicación. No tiene conocimiento sobre la interfaz. El modelo es responsable de manejar la lógica del dominio (reglas comerciales del mundo real) y la comunicación con la base de datos y las capas de red.

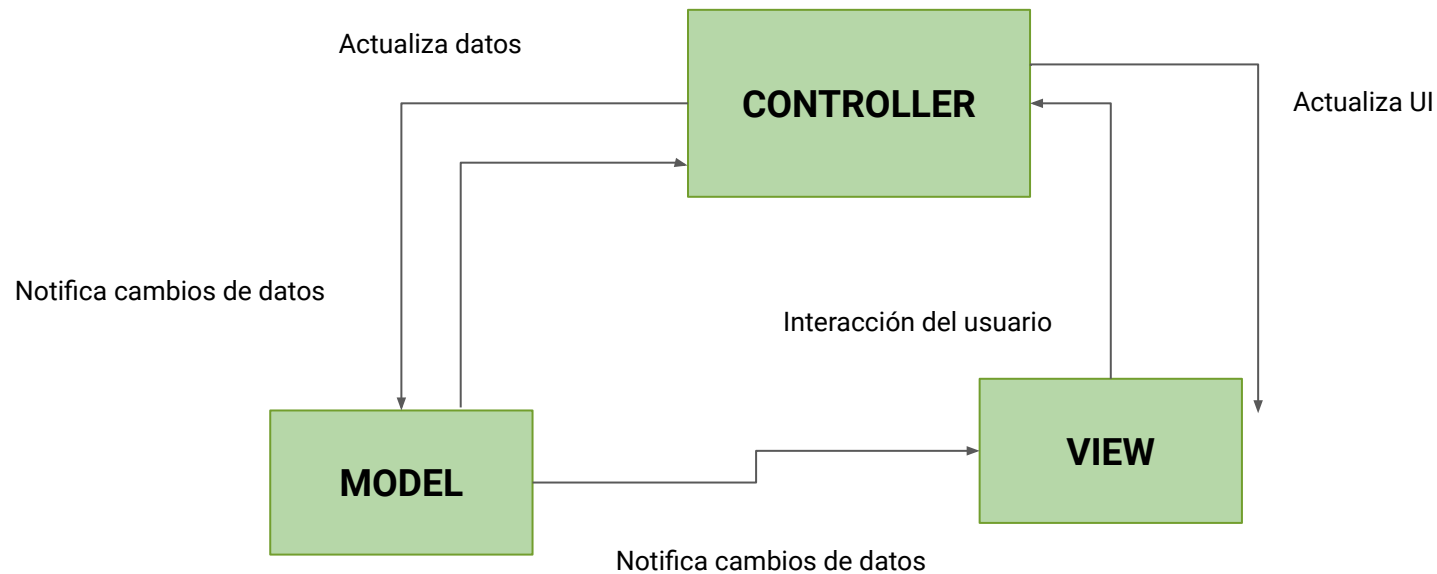
View

Es la capa de UI (interfaz de usuario) que contiene los componentes que son visibles en la pantalla. Además, proporciona la visualización de los datos almacenados en el Modelo y ofrece interacción con el usuario.

Controller

Este componente establece la relación entre View y Model. Contiene la lógica de la aplicación central y se informa del comportamiento del usuario y actualiza el Model según la necesidad.

MVC



MVP



- MVP (Model View Presenter) es una derivación del patrón MVC que se usa principalmente para construir interfaces de usuario.
- En MVP, el presentador asume la funcionalidad de “intermediario” y toda la lógica de presentación se envía al presentador.
- MVP aboga por separar la lógica comercial y de persistencia de la Actividad y el Fragmento.

MVP

View

La Vista, normalmente implementada por una Actividad, contendrá una referencia al presentador. Lo único que hará la vista es llamar a un método desde el Presentador cada vez que haya una acción en la interfaz.



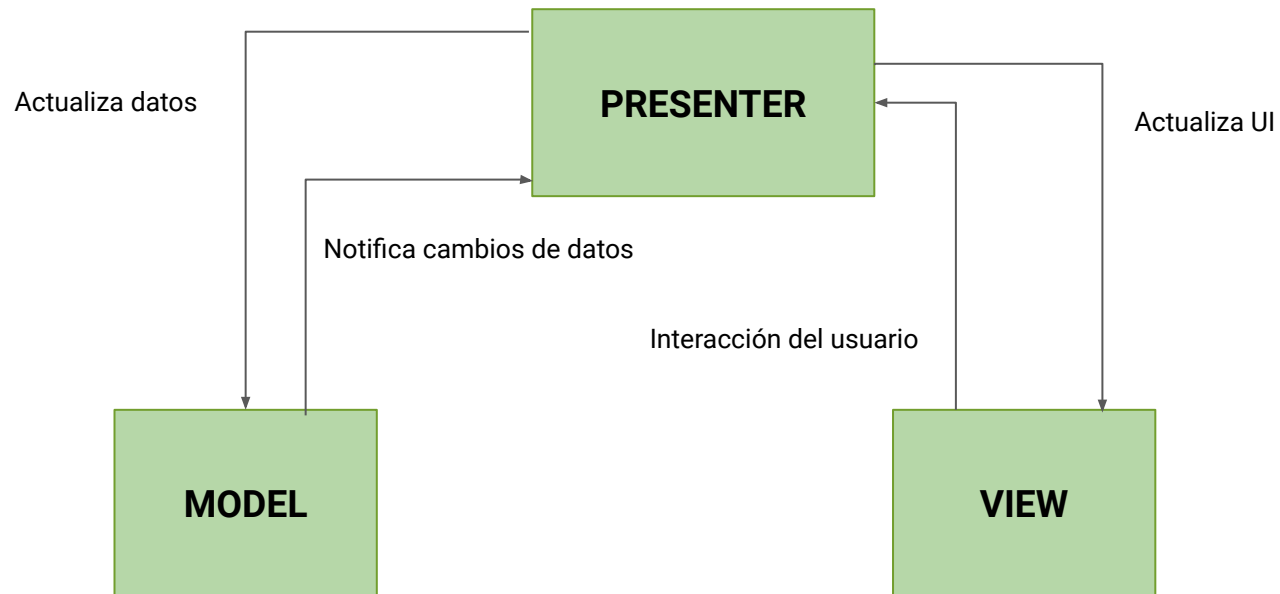
Model

En una aplicación con una buena arquitectura en capas, este modelo solo sería la puerta de entrada a la capa de dominio o lógica de negocios. Véalo como el proveedor de los datos que queremos mostrar en la vista. Las responsabilidades del modelo incluyen el uso de API, el almacenamiento en caché de datos, la gestión de bases de datos, etc.

Presenter

El presentador es responsable de actuar como intermediario entre la vista y el modelo. Recupera datos del modelo y los devuelve formateados a la vista. Pero a diferencia del MVC típico, también decide qué sucede cuando interactúa con la Vista.

MVP



MVVM

Model View ViewModel (MVVM), a primera vista, parece muy similar al MVP porque ambos hacen un gran trabajo al abstraer el estado y el comportamiento de la vista.

El modelo de presentación abstrae una vista independiente de una plataforma de interfaz de usuario específica, mientras que el patrón MVVM se creó para simplificar la programación de interfaces de usuario impulsada por eventos.



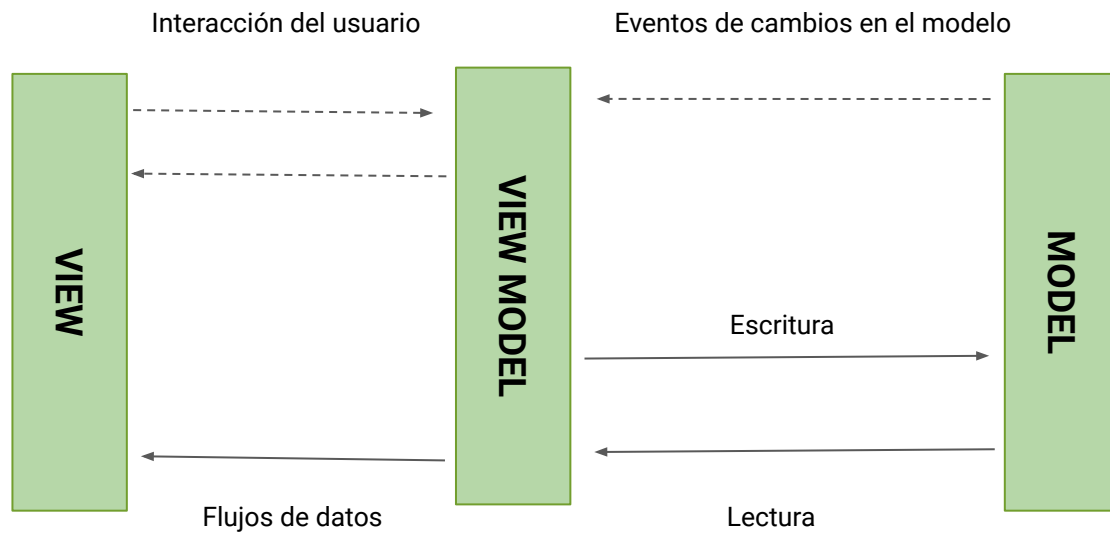
MVVM

Si el patrón MVP significaba que el presentador le decía a la vista directamente qué mostrar, en MVVM ViewModel expone flujos de eventos a los que se pueden vincular las vistas. De esta manera, el modelo de vista ya no necesita contener una referencia a la vista, como lo es el presentador.

Esto también significa que ahora se eliminan todas las interfaces que requiere el patrón MVP.



MVVM



MVI



Model-View-Intent (MVI) agiliza el proceso de creación y desarrollo de aplicaciones mediante la introducción de un enfoque reactivo. En cierto modo, este patrón es una mezcla de MVP y MVVM adaptado a la programación reactiva.

MVI elimina el uso de devolución de llamada y reduce significativamente la cantidad de métodos de entrada/salida. También es una gran solución para los estados de sincronización entre la vista y la capa de lógica de negocio.

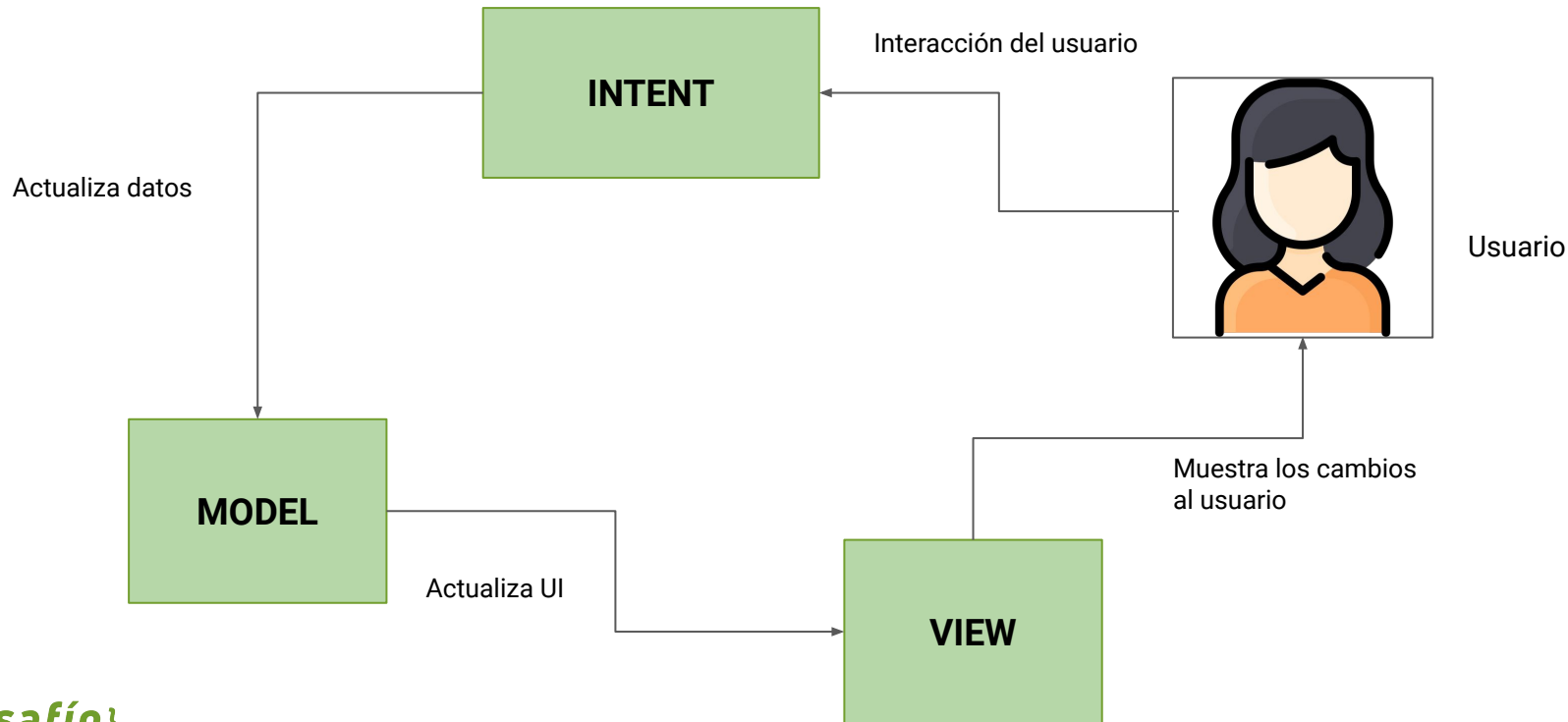
MVI



Ventajas de MVI:

- única fuente de verdad: un estado inmutable común a todas las capas, como la única fuente de verdad
- flujo de datos unidireccional y cíclico
- facilidad de captura y corrección de errores
- facilidad de prueba del código
- capacidad de probar todas las capas de la aplicación con pruebas unitarias

MVI



***/* Patrones de diseño sugeridos para
Android */***

Patrones de diseño sugeridos para Android

Todos los patrones de diseños mencionados anteriormente pueden ser ocupados para el desarrollo de aplicaciones en Android, sin embargo, los más comunes son:

- **MVP** (Model View Presenter): Ampliamente utilizado, previo a la introducción del ViewModel, es muy probable encontrar este patrón de diseño en aplicaciones antiguas.
- **MVI** (Model View Intent): Este patrón de diseño ha estado ganando popularidad con la introducción de Jetpack Compose.
- **MVVM** (Model View ViewModel): Patrón de diseño sugerido para el desarrollo de apps en Android.

¿Por qué se sugiere MVVM?

Hay muchos beneficios de MVVM. Veamos algunos:

- Esta es una arquitectura recientemente introducida y contiene componentes como LiveData o StateFlow, que es un contenedor de datos observable, los que veremos en detalles más adelante.
- LiveData y StateFlow son conscientes del ciclo de vida. Significa que el ViewModel se conserva a lo largo del ciclo de vida del Activity.
- Los objetos LiveData y StateFlow pueden observar los cambios sin crear rutas de dependencia explícitas y rígidas entre ellos.
- ViewModel retiene automáticamente sus datos de almacenamiento durante los cambios de configuración.

Resumen

- Antes de finalizar recuerda que los patrones de diseños cambian dependiendo de las necesidades del software, así que no es de sorprender que tengamos un nuevo patrón de diseño en el futuro
- Una de las mejores formas de entender MVVM, es viendo el código fuente de las apps. Puedes encontrar muchos ejemplos en Github, por ejemplo...
 - <https://github.com/akhilesh0707/Rick-and-Morty>
 - <https://github.com/MindorksOpenSource/android-mvvm-architecture>
 - <https://github.com/mitchtabian/MVVMRecipeApp>



Próxima sesión...

- *Implementar un aplicativo utilizando el patrón MVVM de acuerdo a las buenas prácticas recomendadas.*

{desafío}
latam_

*Academia de
talentos digitales*

