UNIVERSIDAD DON BOSCO

FACULTAD DE INGENIERÍA

ESCUELA DE COMPUTACIÓN

Desarrollo de Software para Móviles

Ing. Alexander Alberto Siguenza Campos

Tema:

Trabajo Investigación- Patron MVVM

Alumno	Código	Grupo
Marcelo Humberto Leiva Salazar	LS192212	04L
Christian Giovanni Tobar Cerón	TC192221	03L

29 de abril de 2023

Contenido

Introducción 3
Qué es el patrón MVVM? 4
¿Cuáles son sus componentes principales y cómo se relacionan entre sí? 5
Vista 5
ViewModel 5
Modelo 6
¿Cómo se aplica el patrón MVVM en Android con Kotlin? 7
¿Cuáles son las ventajas y desventajas de utilizar el patrón MVVM en el desarrollo de aplicaciones móviles? 8
Ventajas 8
Desventajas 8
Anexos 9
Bibliografía 11

Introducción

El patrón Modelo-Vista-ViewModel (MVVM) es uno de los patrones de diseño de software más populares y ampliamente utilizado en el desarrollo de aplicaciones para Android Kotlin. Este patrón de arquitectura de software separa la lógica de la interfaz de usuario de la lógica del negocio de la aplicación, lo que hace que el código sea más modular, escalable y fácil de mantener.

En el patrón MVVM, la Vista se encarga de mostrar la información al usuario, el Modelo maneja la lógica del negocio y el ViewModel actúa como un puente entre la Vista y el Modelo. El ViewModel también es responsable de mantener el estado de la vista y proporcionar una API limpia para que la vista interactúe con el modelo.

El uso de MVVM en aplicaciones Android Kotlin tiene varios beneficios, como una separación clara de responsabilidades, que permite a los desarrolladores trabajar en paralelo y colaborar fácilmente en grandes equipos de desarrollo. Además, el patrón MVVM facilita las pruebas unitarias, ya que la lógica del negocio se separa de la interfaz de usuario, lo que permite la realización de pruebas de manera más efectiva.

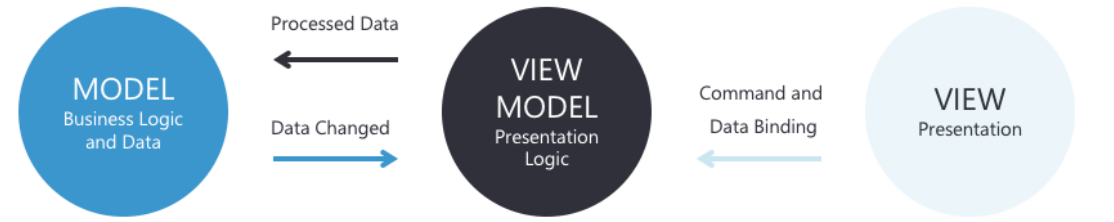
Qué es el patrón MVVM?

La clave para comprender la arquitectura MVVM es comprender cómo los tres componentes clave de MVVM interactúan entre sí. Dado que la vista sólo se comunica con el ViewModel y el ViewModel sólo se comunica con el modelo.

Toda la interacción del usuario ocurre dentro de la Vista, que se encarga de detectar la entrada del usuario (clics del mouse, entrada del teclado) y reenviarla al ViewModel a través del enlace de datos. El enlace de datos se puede implementar con devoluciones de llamada o propiedades y constituye el vínculo concreto entre View y ViewModel.

ViewModel implementa las propiedades y los comandos a los que se puede enlazar la vista. Estas propiedades y comandos definen la funcionalidad que la Vista puede ofrecer al usuario, aunque la forma de mostrarla depende totalmente de la Vista. El ViewModel también se encarga de proporcionar a la vista los datos de las clases del modelo, que la vista puede consumir. Para ello, ViewModel puede exponer clases Model directamente a View, en cuyo caso la clase Model necesitaría admitir el enlace de datos y cambiar los eventos de notificación.

Los modelos son clases que modelan el dominio de la aplicación. Los modelos encapsulan los datos y la lógica empresarial de la aplicación. Pueden considerarse objetos de negocio que no tienen absolutamente ninguna relación con el aspecto visual de la aplicación.



MVVM **MODEL VIEW VIEWMODEL**

¿Cuáles son sus componentes principales y cómo se relacionan entre sí?

Vista

La vista es responsable de definir la estructura, el diseño y la apariencia de lo que el usuario ve en la pantalla. Idealmente, cada vista se define en XAML, con un código subyacente limitado que no contiene lógica empresarial. Sin embargo, en algunos casos, el código subyacente puede contener lógica de interfaz de usuario que implementa un comportamiento visual que es difícil de expresar en XAML, como animaciones.

También es posible representar vistas mediante una plantilla de datos que define los elementos de la interfaz de usuario que se utilizarán para mostrar visualmente un objeto en pantalla. En este caso, la plantilla de datos funciona como una vista sin código subyacente, y está pensada para enlazarse con un modelo de vista particular. De esta manera, se pueden crear vistas personalizadas para diferentes tipos de objetos sin necesidad de escribir código específico para cada una.

ViewModel

El modelo de vista se encarga de implementar propiedades y comandos a los cuales se pueden vincular los datos de la vista, y notifica a la vista de cualquier cambio de estado a través de eventos de notificación de cambios. Las propiedades y comandos que se ofrecen en el modelo de vista definen la funcionalidad que brinda la interfaz de usuario, pero es la vista la que determina cómo se va a mostrar esa funcionalidad.

El modelo de vista también es responsable de coordinar las interacciones de la vista con cualquier clase de modelo que se requiera. Normalmente existe una relación de uno a varios entre el modelo de vista y las clases de modelo. El modelo de vista puede optar por exponer las clases de modelo directamente a la vista para que los controles de la vista puedan enlazar los datos directamente a ellas. En este caso, las clases de modelo deberán diseñarse para admitir el enlace de datos y los eventos de notificación de cambios.

El modelo de vista también es responsable de coordinar las interacciones de la vista con cualquier clase de modelo que se requiera. Normalmente existe una relación de uno a varios entre el modelo de vista y las clases de modelo. El modelo de vista puede optar por exponer las clases de modelo directamente a la vista para que los controles de la vista puedan enlazar los datos directamente a ellas. En este caso, las clases de modelo deberán diseñarse para admitir el enlace de datos y los eventos de notificación de cambios.

Modelo

Las clases de modelo son clases no visuales que encapsulan los datos de la aplicación. Por lo tanto, se puede pensar que el modelo representa el modelo de dominio de la aplicación, que generalmente incluye un modelo de datos junto con lógica empresarial y de validación. Entre los ejemplos de objetos de modelo se incluyen objetos de transferencia de datos (DTO), objetos CLR antiguos (POCO) y objetos proxy y de entidad generados.

Las clases de modelo se utilizan normalmente junto con servicios o repositorios que encapsulan el acceso a los datos y el almacenamiento en caché.

En resumen:

MODELO

El modelo representa el modelo de dominio de la aplicación, que puede incluir un modelo de datos, así como lógica empresarial y de validación. Se comunica con el ViewModel y carece de conocimiento de la vista.

VISTA

La vista representa la interfaz de usuario de la aplicación y contiene una lógica limitada y puramente de presentación que implementa el comportamiento visual. The View es completamente agnóstico a la lógica empresarial. En otras palabras, la Vista es una clase que nunca contiene datos, ni los manipula directamente. Se comunica con ViewModel a través del enlace de datos y no conoce el modelo.

VIEWMODEL

ViewModel es el vínculo entre la vista y el modelo. Implementa y expone propiedades y comandos públicos que View utiliza mediante el enlace de datos. Si se produce algún cambio de estado, ViewModel notifica a la vista mediante eventos de notificación.

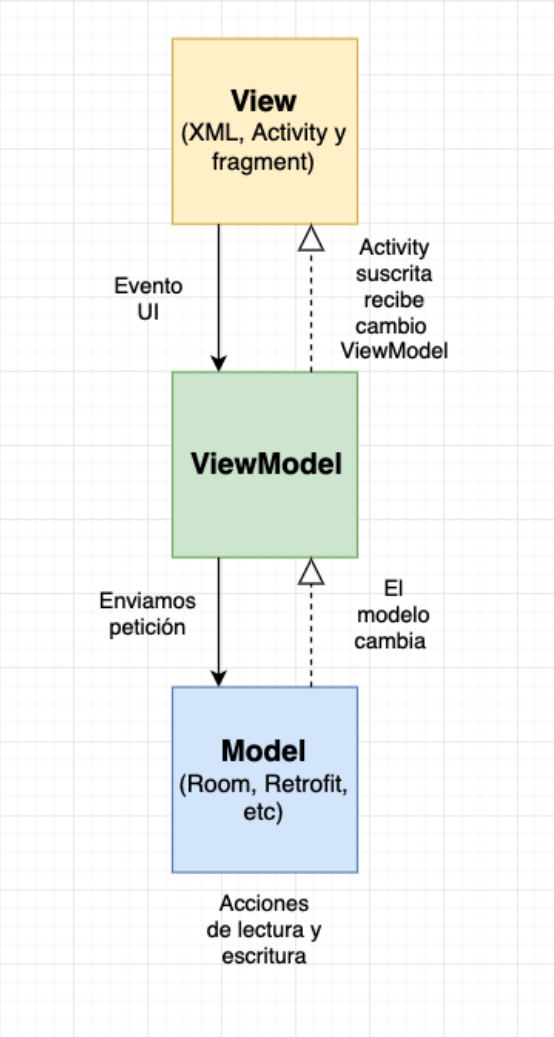
¿Cómo se aplica el patrón MVVM en Android con Kotlin?

Imagina una aplicación de citas famosas que contenga un modelo de datos con texto y autor para cada cita. Nuestra tarea consiste en mostrar una cita en la pantalla y permitir que el usuario cambie a la siguiente cita al hacer clic en la pantalla.

En la arquitectura MVVM, esto es fácil de lograr. Primero, la actividad se suscribe al ViewModel mediante LiveData, que utiliza el patrón observador para permitir que la actividad sea notificada cuando se produce un cambio. Esto es importante porque la actividad solo se actualizará cuando el ViewModel lo notifique.

Además, la actividad debe controlar el evento de clic en la pantalla para informar al ViewModel. La función de la actividad se limita a mostrar la información proporcionada por el ViewModel y notificar los eventos de la interfaz de usuario.

Nuestro viewModel acaba de recibir que ha habido un evento en la UI por lo que tendrá que modificar el modelo de la cita. Para ello llamará al modelo que este irá a Retrofit, a Room o a cualquier tipo de acceso que nos devuelva datos (en este caso una nueva cita) y se lo devolverá al viewModel que a su vez notificará a la activity el cambio del contenido para que lo refresque.



1. Crear un nuevo proyecto en Android Studio según las necesidades de tu app, por ejemplo, el lenguaje de programación y el SDK que vas a usar.
2. Crear la clase Modelo con el fin de que el programa sepa la estructura que va a definir la presentación de los datos de tu app.
3. Trabajar con el archivo activity_main.xml para establecer las diferentes entradas y componentes de la app.
4. Crear la clase ViewModel para indicar los métodos que se deben llamar para el buen funcionamiento de la app.
5. Definir las funcionalidades de View en el fichero de MainActivity.
6. Ejecutar la app.

¿Cuáles son las ventajas y desventajas de utilizar el patrón MVVM en el desarrollo de aplicaciones móviles?

Ventajas

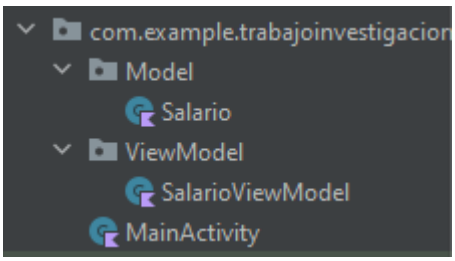
- **Más fácil de desarrollar:** Separar la vista de la lógica hace posible que diferentes equipos trabajen en diferentes componentes simultáneamente. Un equipo de diseñadores puede centrarse en la interfaz de usuario mientras los desarrolladores trabajan en la implementación de la lógica (ViewModel y Model).
- **Más fácil de probar:** Una de las cosas más difíciles de probar en una aplicación es la interfaz de usuario (UI). Dado que ViewModel y Model son completamente independientes de la vista, los desarrolladores pueden escribir pruebas para ambos sin tener que usar la vista.
- **Más fácil de mantener:** La separación entre los diferentes componentes de la aplicación hace que el código sea más simple y limpio. Como resultado, el código de la aplicación es mucho más fácil de entender y, por lo tanto, de mantener. Es más fácil entender dónde debemos implementar nuevas características y cómo se conectan con el patrón existente. Con una MVVM, también es más fácil implementar más patrones arquitectónicos (inversión de dependencias, servicios y más).

Desventajas

- **Complejidad:** MVVM es excesivo cuando se trata de crear interfaces de usuario simples. Cuando se trabaja en proyectos más grandes, diseñar el ViewModel para obtener la cantidad correcta de generalidad puede ser bastante difícil.
- **Difícil de depurar:** debido a que el enlace de datos es declarativo, puede ser más difícil de depurar que el código imperativo tradicional.
- **Aumento de la cantidad de código:** Debido a la necesidad de crear tres componentes distintos, MVVM puede aumentar la cantidad de código que necesita una aplicación. Esto puede aumentar la complejidad y dificultar la lectura y mantenimiento del código.
- **Aprendizaje requerido:** Para trabajar con MVVM, los desarrolladores necesitan tener una comprensión completa del patrón y su funcionamiento. Esto puede requerir tiempo y recursos adicionales para aprender y entrenar a los desarrolladores.
- **Dependencias adicionales:** MVVM a menudo requiere bibliotecas adicionales para que funcione correctamente. Esto puede aumentar la complejidad y el costo de la aplicación.

Anexos

Estructura – El mainActivity es nuestro view o vista



ViewModel

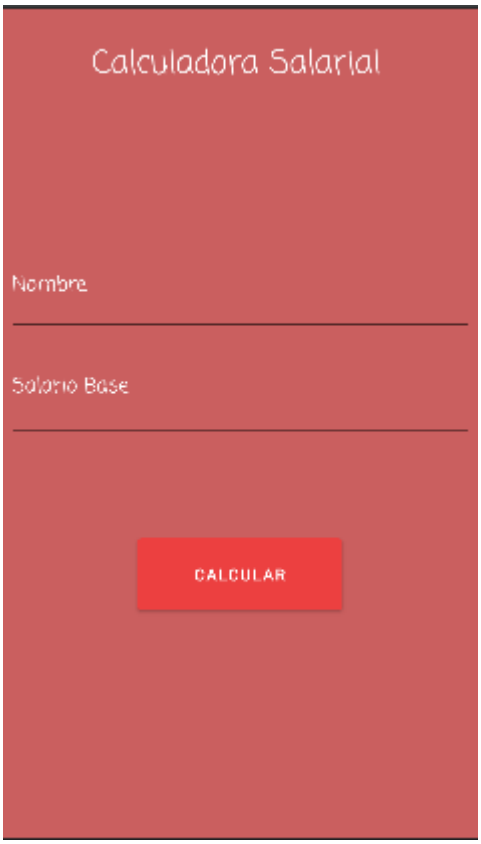
```
class SalarioViewModel: ViewModel() {
    private val _salario = MutableLiveData<Salario>()
    val salario: LiveData<Salario>
        get() = _salario

    fun calcularSalario(nombre: String, salarioBase: Double) {
        val descuento = salarioBase * 0.07
        val salarioNeto = salarioBase - descuento
        val salario = Salario(nombre, salarioBase, descuento, salarioNeto)
        _salario.value = salario
    }
}
```

Model

```
data class Salario(val nombre: String, val salarioBase: Double, val descuento: Double, val salarioNeto: Double)
```

Vista




```
calcularButton.setOnClickListener { it: View!
    val nombre = nombreEditText.text.toString()
    val salarioBase = salarioBaseEditText.text.toString().toDoubleOrNull() ?: 0.0
    salarioViewModel.calcularSalario(nombre, salarioBase)
}
salarioViewModel.salario.observe( owner: this, { salario ->
    salarioNetoTextView.text =
        "${salario.nombre}, tu salario Neto es ${salario.salarioNeto}"
})
```

Bibliografía

David Britch, N. S. (07 de Agosto de 2021). *Patrones de Modelo*. Obtenido de Microsoft: https://learn.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

MVVM en Android con Kotlin, LiveData y View Binding. (22 de Abril de 2021). Obtenido de CursoKotlin: https://cursokotlin.com/mvvm-en-android-con-kotlin-livedata-y-view-binding-android-architecture-components/

Pasos para la implementación de MVVM en Android. (23 de Diciembre de 2022). Obtenido de KeepCoding: https://keepcoding.io/blog/pasos-para-la-implementacion-de-mvvm-en-android/#:~:text=A%20grandes%20rasgos%2C%20la%20arquitectura,para%20el%20almacenamiento%20de%20datos.