UNIVERSIDAD DON BOSCO
Escuela de Computación
FACULTAD DE INGENIERÍA
CICLO I– 2023

TRABAJO DE INVESTIGACIÓN

Carlos Moisés Pérez Cabrera G04L
Victoria Margarita Sura Jimenez G03L

INTRODUCCIÓN

El patrón MVVM (Model-View-ViewModel) es un patrón arquitectónico utilizado en el desarrollo de aplicaciones móviles, que proporciona una separación clara de responsabilidades entre la vista, la lógica del negocio y los datos de la aplicación. En Android con Kotlin, el patrón MVVM es ampliamente utilizado y ofrece una serie de ventajas, como una mayor organización del código, facilidad de prueba y reutilización de código.

En este informe se explicará con mayor detalle cómo se aplica el patrón MVVM en Android con Kotlin, sus ventajas y desventajas.

PATRÓN DE DISEÑO

MVVM - (Model-View-ViewModel)

MVVM (Model-View-ViewModel) es un patrón arquitectónico que se usa comúnmente en el desarrollo de aplicaciones móviles. La lógica de vista está separada de la lógica de negocios en este patrón.

En el contexto de Android con Kotlin, el patrón MVVM se utiliza para crear aplicaciones fáciles de probar y escalables. La siguiente es una breve descripción de cada uno de los componentes y cómo se utilizan.

La capa de datos es el modelo. Los datos y la lógica empresarial son administrados por él. El modelo se puede implementar usando una base de datos local o una combinación de los dos. La capa del modelo puede ser manejada por bibliotecas populares.

La interfaz de usuario se llama la vista. Maneja las interacciones del usuario y muestra datos al usuario. La vista se puede implementar utilizando los dos lenguajes de programación. La capa de vista contiene varias bibliotecas populares.

ViewModel actúa como un conducto entre la vista y el modelo. Para visualizar y gestionar las interacciones de los usuarios, es responsable de proporcionar los datos necesarios. ViewModel se puede implementar usando la biblioteca Kotlin.

APLICACIÓN

MVVM - (Model-View-ViewModel)

En Android con Kotlin, el patrón MVVM (Model-View-ViewModel) se aplica siguiendo los siguientes pasos:

DEFINIR EL MODELO

El modelo representa la capa de datos de la aplicación y es responsable de manejar la lógica del negocio y los datos de la aplicación. Para definir el modelo, se pueden utilizar tecnologías como Room para almacenamiento local, Retrofit para llamadas a API remota, y Firebase para el almacenamiento en la nube.

DEFINIR LA VISTA

La vista representa la interfaz de usuario de la aplicación y es responsable de mostrar los datos al usuario y de manejar las interacciones del usuario. Para definir la vista, se pueden utilizar tecnologías como XML y Kotlin. En la vista, se deben evitar incluir lógica de negocio y en su lugar centrarse en mostrar los datos.

DEFINIR LA VM

El ViewModel es responsable de manejar la comunicación entre la vista y el modelo. El ViewModel proporciona los datos necesarios a la vista y maneja las interacciones del usuario, pero no conoce detalles de la implementación de la vista. El ViewModel también puede manejar la lógica de negocio y la validación de datos. Para definir el ViewModel, se pueden utilizar tecnologías como la biblioteca de arquitectura ViewModel de Android Jetpack.

APLICACIÓN

MVVM - (Model-View-ViewModel)

ESTABLECER LA COMUNICACIÓN ENTRE LA VISTA Y EL VM

La comunicación entre la vista y el ViewModel se establece mediante la observación de los cambios en los datos. En la vista, se observa el ViewModel y se actualiza automáticamente cuando los datos cambian. En el ViewModel, se notifica a la vista cuando los datos han cambiado.

MANEJAR LOS EVENTOS DEL USUARIO

El ViewModel es responsable de manejar los eventos del usuario, como hacer clic en un botón o deslizar una pantalla. En lugar de incluir la lógica de evento en la vista, se puede enviar un mensaje al ViewModel, que manejará la lógica de evento y actualizará el modelo o la vista según sea necesario.

VENTAJAS

MVVM - (Model-View-ViewModel)

Separación clara de responsabilidades: **MVVM separa la lógica de la vista de la lógica del negocio, lo que permite una mejor organización y mantenimiento del código.**

Facilita la prueba unitaria: **El patrón MVVM permite una fácil prueba unitaria de la lógica del negocio y del ViewModel, ya que el ViewModel no depende de la vista.**

Comunicación bidireccional: **La comunicación bidireccional entre la vista y el ViewModel se realiza mediante observables, lo que permite que los cambios en los datos se reflejen automáticamente en la vista sin la necesidad de actualizar manualmente la interfaz de usuario.**

Reutilización de código: **El patrón MVVM facilita la reutilización de código, ya que la lógica de negocio se encuentra en el ViewModel y puede ser utilizada por varias vistas.**

DESVENTAJAS

MVVM - (Model-View-ViewModel)

Curva de aprendizaje: **El patrón MVVM puede tener una curva de aprendizaje empinada, especialmente para aquellos que no están familiarizados con la programación reactiva.**

Mayor complejidad: **La implementación del patrón MVVM puede aumentar la complejidad del código en comparación con otros patrones arquitectónicos más simples.**

Potencial sobrecarga de ViewModel: **Si no se maneja adecuadamente, el ViewModel puede acumular una gran cantidad de lógica de negocio, lo que puede dificultar su mantenimiento.**

No es una solución universal: **El patrón MVVM puede no ser adecuado para todas las aplicaciones y situaciones, y puede ser necesario utilizar otros patrones arquitectónicos según las necesidades específicas de la aplicación.**