FACULTAD DE INGENIERÍA

ESCUELA DE COMPUTACION

Desarrollo de Software para Móviles DSM104 G03L

Ciclo Académico: 01-2023

“Tercer Desafío Practico”

Docente:
Alexander Alberto Sigüenza Campos

Presentado por:

Nombre	Carnet
Olmedo López, Edwin Josué	OL150100

Soyapango, abril 29 de 2023

Índice

Introducción

El patrón Model-View-ViewModel (MVVM) se ha convertido en una de las arquitecturas más populares para el desarrollo de aplicaciones móviles. MVVM es un patrón de arquitectura de software que ayuda a los desarrolladores a separar la lógica de presentación de la lógica de negocio en una aplicación. Este patrón se centra en la separación de responsabilidades y la creación de un código escalable, mantenible y fácilmente de hacer pruebas.

En este trabajo de investigación, se explorará el patrón MVVM en el contexto del desarrollo de aplicaciones móviles. Se examinarán los componentes principales del patrón MVVM y cómo se relacionan entre sí. Además, se describirán las ventajas y desventajas de utilizar el patrón MVVM en el desarrollo de aplicaciones móviles.

El trabajo de investigación también analizará cómo se aplica el patrón MVVM en Android con Kotlin, uno de los lenguajes de programación más populares para el desarrollo de aplicaciones móviles. Se discutirán las mejores prácticas y se proporcionarán ejemplos de código para ilustrar cómo se puede utilizar el patrón MVVM en el desarrollo de aplicaciones móviles en Android.

Patrón MVVM

El patrón MVVM (Model-View-ViewModel) es un patrón arquitectónico utilizado en el desarrollo de aplicaciones de software. Fue introducido por primera vez por Microsoft en 2005 como una variante del patrón MVC (Modelo-Vista-Controlador) y se utiliza ampliamente en el desarrollo de aplicaciones de Windows y aplicaciones móviles.

El patrón MVVM separa la lógica de presentación de la interfaz de usuario (UI) de la lógica de negocio y datos subyacentes, lo que facilita la gestión y el mantenimiento del código. En este patrón, la vista es responsable de mostrar los datos en la interfaz de usuario, mientras que el modelo es responsable de almacenar los datos y proporcionar la lógica de negocio. El ViewModel actúa como un intermediario entre la vista y el modelo, y se encarga de la presentación y transformación de los datos para que puedan ser mostrados correctamente en la vista.

Además, el patrón MVVM facilita la implementación de pruebas unitarias, ya que la separación de responsabilidades permite probar cada componente por separado. Esto hace que sea más fácil detectar y corregir errores en el código y aumenta la calidad del software.

Componentes principales y cómo se relacionan entre sí

Los componentes principales del patrón MVVM son los siguientes:

- Modelo (Model): es la capa encargada de representar la estructura y los datos de la aplicación. Esta capa incluye lógica de negocio, acceso a datos y cualquier otra tarea relacionada con el procesamiento y almacenamiento de información.
- Vista (View): es la capa encargada de presentar la interfaz de usuario de la aplicación. Esta capa incluye todos los elementos visuales, como botones, cuadros de texto, gráficos y cualquier otro elemento con el que el usuario interactúa.
- ViewModel: es la capa que actúa como intermediario entre el modelo y la vista. El ViewModel recibe los datos del modelo y los prepara para ser mostrados en la vista. Además, también recibe las interacciones del usuario desde la vista y las procesa para actualizar los datos en el modelo.

La relación entre estos componentes se describe a continuación:

- La vista se enlaza al ViewModel a través de un enlace de datos. La vista muestra los datos que el ViewModel proporciona y envía eventos al ViewModel cuando el usuario interactúa con ella.
- El ViewModel se enlaza al modelo y se encarga de recopilar los datos del modelo y prepararlos para su presentación en la vista. También recibe eventos de la vista y los procesa para actualizar el modelo si es necesario..
- El modelo se enlaza al ViewModel y proporciona los datos que se van a mostrar en la vista. También recibe actualizaciones del ViewModel y las procesa para actualizar los datos del modelo si es necesario.

En general, la vista no interactúa directamente con el modelo, sino que lo hace a través del ViewModel. Esta separación de responsabilidades permite una mejor modularidad y escalabilidad de la aplicación.

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

1- Crear las clases para cada uno de los componentes del patrón MVVM. Por ejemplo, se podría crear una clase para el modelo, una clase para la vista y una clase para el ViewModel.
2- Enlazar la vista con el ViewModel. Esto se puede hacer utilizando el Data Binding, una característica de Android que permite enlazar los datos de la vista con el ViewModel de forma declarativa. Para usar el Data Binding, se debe habilitar en el archivo build.gradle del módulo de la aplicación y definir los enlaces de datos en el diseño de la vista.
3- Enlazar el ViewModel con el modelo. Esto se puede hacer utilizando repositorios, que son clases que se encargan de interactuar con el modelo y proporcionar los datos al ViewModel. Los repositorios pueden utilizar patrones de diseño como el patrón Repository o el patrón DAO (Data Access Object).
4- Implementar la lógica de negocio en el ViewModel. El ViewModel debe proporcionar los datos y la lógica necesaria para que la vista muestre la interfaz de usuario. También debe proporcionar los métodos necesarios para que la vista envíe eventos al ViewModel y actualice los datos del modelo.
5- Implementar la lógica del modelo. El modelo debe proporcionar los datos y la lógica de negocio necesarios para que el ViewModel pueda procesarlos y proporcionarlos a la vista. Además, el modelo debe proporcionar los métodos necesarios para actualizar los datos y realizar operaciones de lectura y escritura en el almacenamiento persistente.

Ventajas y desventajas de utilizar el patrón MVVM en el desarrollo de aplicaciones móviles

Las ventajas de utilizar el patrón MVVM en el desarrollo de aplicaciones móviles son las siguientes:

1- Separación de responsabilidades: el patrón MVVM permite separar claramente las responsabilidades de cada componente de la aplicación, lo que facilita el mantenimiento y la escalabilidad del código.
2- Pruebas unitarias: el patrón MVVM facilita la creación de pruebas unitarias ya que los componentes están separados y se pueden probar de forma independiente.
3- Flexibilidad: el patrón MVVM proporciona una estructura flexible que se puede adaptar a las necesidades específicas de la aplicación.
4- Enfoque en la interfaz de usuario: el patrón MVVM se centra en la presentación de la interfaz de usuario, lo que facilita el diseño y la implementación de la experiencia del usuario.

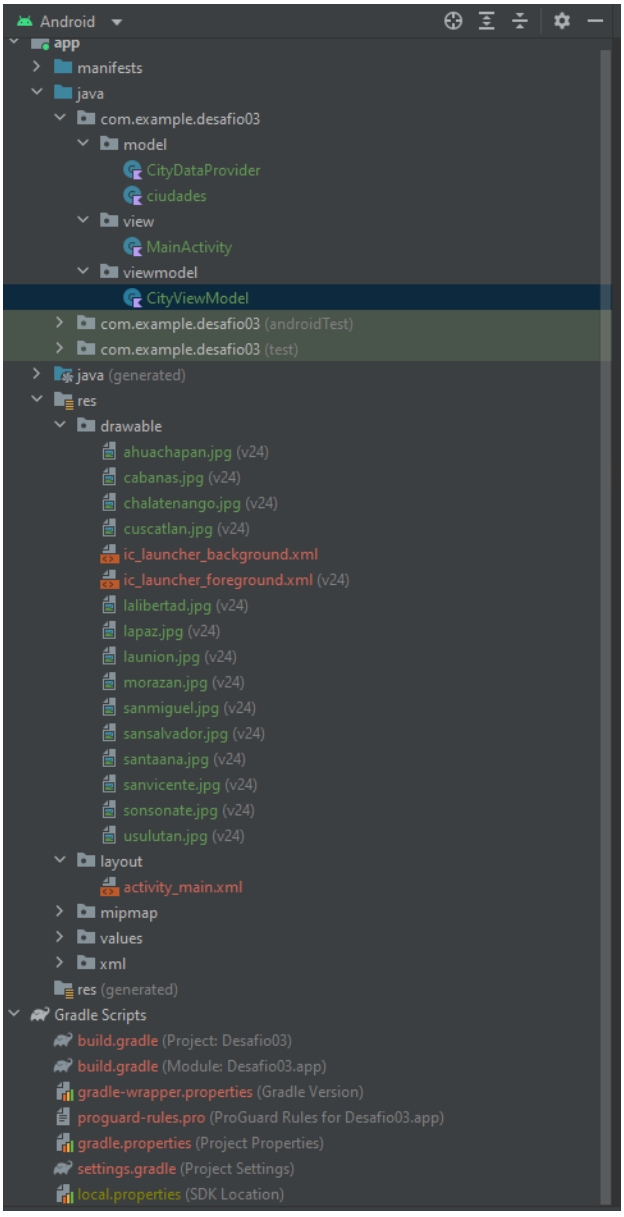
Las desventajas de utilizar el patrón MVVM en el desarrollo de aplicaciones móviles son las siguientes:

1- Curva de aprendizaje: el patrón MVVM puede tener una curva de aprendizaje más pronunciada para los desarrolladores que no estén familiarizados con él.
2- Complejidad: el patrón MVVM puede añadir complejidad al código si no se aplica de forma adecuada.
3- Sobrecarga de código: el patrón MVVM puede requerir la creación de clases adicionales, lo que puede aumentar la cantidad de código necesario para la aplicación.
4- Sobrecarga de procesamiento: el patrón MVVM puede aumentar la carga de procesamiento en el dispositivo, especialmente en dispositivos más antiguos o con recursos limitados.

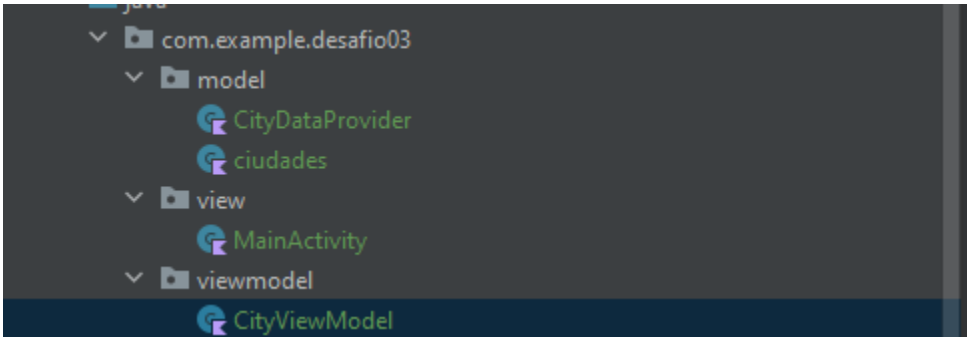
Anexos

Capturas del ejemplo practico

Estructura general del proyecto



Ubicación de clases donde se encuentra el modelo



CityDataProvider

```
package com.example.desafio03.model

import com.example.desafio03.R

class CityDataProvider {

    private val citys = arrayListOf<ciudades>()

    init {
        citys.add(ciudades( name: "Ahuachapán", R.drawable.ahuachapan,  poblacion: 333406))
        citys.add(ciudades( name: "Cabañas", R.drawable.cabanas,  poblacion: 164945))
        citys.add(ciudades( name: "Chalatenango", R.drawable.chalatenango,  poblacion: 204808))
        citys.add(ciudades( name: "Cuscatlán", R.drawable.cuscatlan,  poblacion: 252528))
        citys.add(ciudades( name: "La Libertad", R.drawable.lalibertad,  poblacion: 747662))
        citys.add(ciudades( name: "Morazán", R.drawable.morazan,  poblacion: 199519))
        citys.add(ciudades( name: "La Paz", R.drawable.lapaz,  poblacion: 328221))
        citys.add(ciudades( name: "Santa Ana", R.drawable.santaana,  poblacion: 572081))
        citys.add(ciudades( name: "San Miguel", R.drawable.sanmiguel,  poblacion: 478792))
        citys.add(ciudades( name: "San Salvador", R.drawable.sansalvador,  poblacion: 1740847))
        citys.add(ciudades( name: "San Vicente", R.drawable.sanvicente,  poblacion: 174561))
        citys.add(ciudades( name: "Sonsonate", R.drawable.sonsonate,  poblacion: 463732))
        citys.add(ciudades( name: "La Unión", R.drawable.launion,  poblacion: 263271))
        citys.add(ciudades( name: "Usulután", R.drawable.usulutan,  poblacion: 366040))

    }

    fun getCities() = citys
```

Ciudades

```
package com.example.desafio03.model

data class ciudades(
    val name: String,
    val img: Int,
    val poblacion: Int
)
```

MainActivity

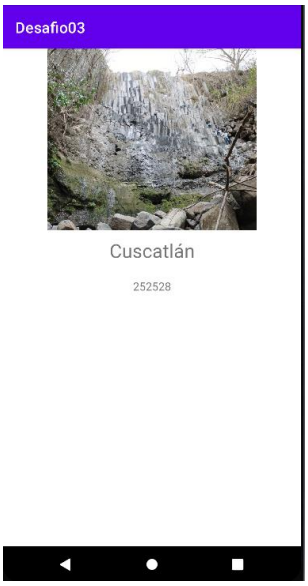
```kotlin
package com.example.desafio03.view

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import androidx.activity.viewModels
import androidx.core.content.res.ResourcesCompat
import androidx.lifecycle.Observer
import com.example.desafio03.R
import com.example.desafio03.databinding.ActivityMainBinding
import com.example.desafio03.model.ciudades
import com.example.desafio03.viewmodel.CityViewModel

class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    private val model: CityViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }

    override fun onResume() {
        super.onResume()

        model.getCityData().observe( owner: this, Observer { ciudades ->
            binding.cityImage.setImageDrawable(
                ResourcesCompat.getDrawable(resources, ciudades.img, applicationContext.theme)
            )
            binding.cityNameTV.text = ciudades.name
            binding.cityPopulation.text = ciudades.poblacion.toString()
        })
    }
}
```

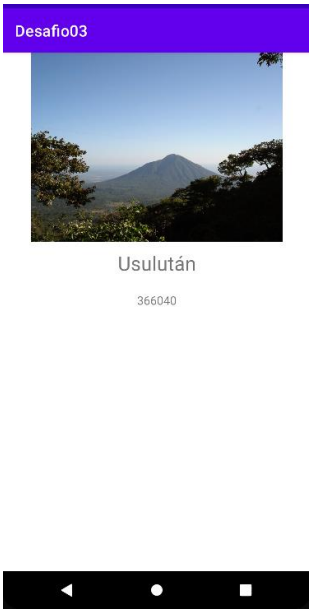
CityViewModel

```kotlin
package com.example.desafio03.viewmodel

import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import com.example.desafio03.model.CityDataProvider
import com.example.desafio03.model.ciudades
import android.os.Handler
import  android.os.Looper
import androidx.lifecycle.LiveData


class CityViewModel: ViewModel() {
    private val cityData =MutableLiveData<ciudades>()
    private val cities = CityDataProvider().getCities()
    private var currentIndex=0
    private val delay= 2000L

    init {
        loop()
    }

    fun getCityData(): LiveData<ciudades> = cityData

    private fun loop(){
        Handler(Looper.getMainLooper()).postDelayed({updateCity()},delay)
    }

    private fun updateCity(){
        currentIndex++
        currentIndex %= cities.size

        cityData.value = cities[currentIndex]

        loop()
    }
}
```

Capturas del funcionamiento en general

Cada 2 segundo se va refrescando la pantalla de la aplicación mostrando un departamento de El Salvador con la cantidad de población que tiene





Enlace de Git donde se encuentra el repositorio

https://github.com/EdwinJosue0124/EjemploMVVM.git

Conclusión

En conclusión, el patrón MVVM es una arquitectura poderosa y eficaz para el desarrollo de aplicaciones móviles. A través de la separación de responsabilidades y la creación de un código escalable y mantenible, el patrón MVVM ayuda a los desarrolladores a crear aplicaciones móviles de alta calidad que satisfacen las necesidades de los usuarios.

En este trabajo de investigación, hemos examinado los componentes principales del patrón MVVM y cómo se relacionan entre sí. También hemos descrito las ventajas y desventajas de utilizar el patrón MVVM en el desarrollo de aplicaciones móviles, y hemos proporcionado ejemplos de su aplicación en Android con Kotlin.

A través de nuestro análisis, hemos demostrado que el patrón MVVM es una arquitectura sólida y confiable que puede mejorar significativamente la calidad del código y la experiencia del usuario final en el desarrollo de aplicaciones móviles. Al adoptar las mejores prácticas y los estándares de codificación consistentes, los desarrolladores pueden aprovechar al máximo el patrón MVVM y crear aplicaciones móviles de alta calidad y éxito.

Bibliografía

- Documentación oficial de Android sobre Arquitectura de aplicaciones móviles: https://developer.android.com/jetpack/guide
- Guía de Google sobre el patrón MVVM en Android: https://developer.android.com/jetpack/docs/guide#recommended-app-arch
- Artículo de Medium sobre el patrón MVVM en Android: https://medium.com/upday-devs/android-architecture-patterns-part-3-model-view-viewmodel-e7eeee76b73b
- Documentación oficial de Microsoft sobre el patrón MVVM en Xamarin.Forms: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm