

SALESIANOS

UNIVERSIDAD DON BOSCO  
FACULTAD DE INGENIERIA  
ESCUELA DE INGENIERÍA EN COMPUTACIÓN  
DISEÑO DE SOFTWARE PARA MOVILES



## **“Patrón MVVM: Componentes y aplicaciones en Android con Kotlin”**

<b>Presentado por:</b>	<b>Código de carné:</b>
Kevin Ángel Galdámez Majano	GM190372
Benjamín Aramis Ruíz Iraheta	RI190244
Osmaro Alfonso Bonilla Mestizo	BM190742

**Docente:**  
Ing. Alexander Sigüenza

Ciclo 01-2023

Soyapango, 25/Abril/2023

# Índice

<b>1.0 Introducción .....</b>	<b>3</b>
<b>2.0 Contenido .....</b>	<b>4</b>
<b>2.1 ¿Qué es el patrón MVVM? .....</b>	<b>4</b>
<b>2.2 Componentes principales del patrón MVVM .....</b>	<b>4</b>
<b>2.3 Relación entre los componentes del patrón MVVM .....</b>	<b>5</b>
<b>2.4 Ventajas y desventajas del patrón MVVM en el desarrollo de aplicaciones móviles.....</b>	<b>6</b>
<b>3.0 Anexos.....</b>	<b>7</b>
<b>4.0 Bibliografía .....</b>	<b>13</b>

## 1.0 Introducción

El desarrollo de aplicaciones móviles es un proceso complejo que implica la implementación de lógicas de negocio específicas para cada tipo de aplicación. Por esta razón, es importante seguir patrones y buenas prácticas que permitan garantizar la calidad del código y la eficiencia del proceso de desarrollo.

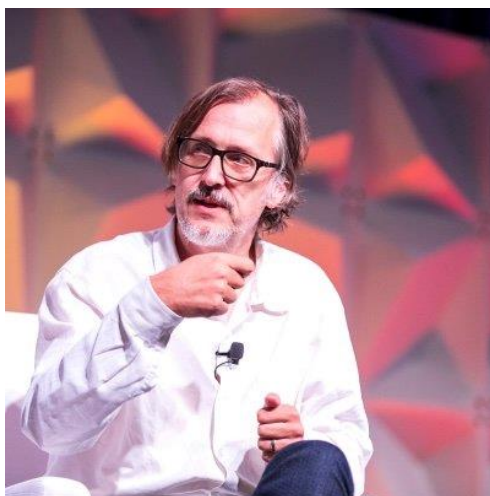
Uno de los patrones más utilizados en el desarrollo de aplicaciones móviles es el patrón MVVM (Modelo-Vista-Modelo de Vista), que permite separar de manera clara la lógica de la interfaz de usuario y la lógica de negocio de la aplicación. La arquitectura MVVM se basa en la separación de responsabilidades entre los componentes del patrón y en la comunicación entre ellos.

En este documento, se presentará una descripción detallada de los componentes principales del patrón MVVM, su relación entre sí y su aplicación en Android con Kotlin. Además, se analizarán las ventajas y desventajas de utilizar este patrón en el desarrollo de aplicaciones móviles.

## 2.0 Contenido

### 2.1 ¿Qué es el patrón MVVM?

El patrón MVVM (Modelo-Vista-Modelo de Vista) es un patrón arquitectónico de software que se utiliza para separar la lógica de la interfaz de usuario y la lógica de negocio de la aplicación. Este patrón fue introducido por John Gossman de Microsoft en el año 2005 y es ampliamente utilizado en el desarrollo de aplicaciones móviles.



### 2.2 Componentes principales del patrón MVVM

Los componentes principales del patrón MVVM son los siguientes:

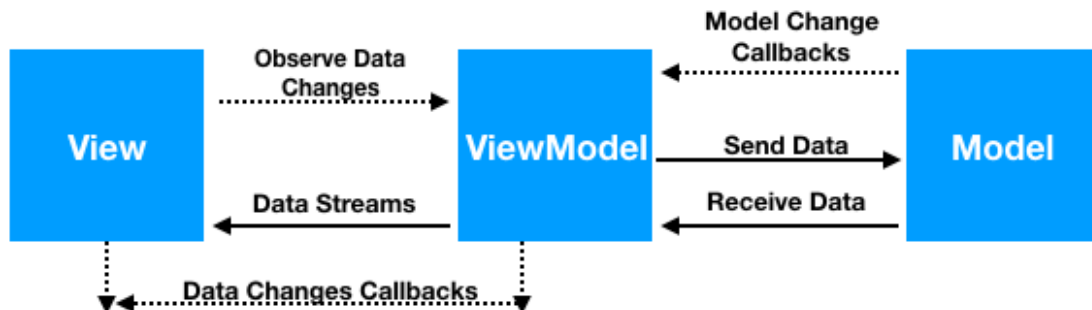
**Modelo (Model):** es la capa encargada de la lógica de negocio de la aplicación, es decir, las operaciones que se realizan en los datos. El modelo se encarga de interactuar con la base de datos, la red, el sistema de archivos u otras fuentes de datos.

**Vista (View):** es la capa encargada de la interfaz de usuario de la aplicación. La vista se encarga de mostrar los datos al usuario y de recibir las acciones que este realiza.

Modelo de vista (View-Model): es la capa que actúa como intermediario entre el modelo y la vista. El modelo de vista se encarga de traducir los datos del modelo a un formato que pueda ser mostrado en la vista y de recibir las acciones del usuario para enviarlas al modelo.

### 2.3 Relación entre los componentes del patrón MVVM

La relación entre los componentes del patrón MVVM se puede representar en un diagrama como el siguiente:



La vista se comunica con el modelo de vista a través de enlaces de datos (data bindings), los cuales permiten que los cambios en los datos se reflejen automáticamente en la vista y viceversa. El modelo de vista se comunica con el modelo a través de métodos y eventos.

### 5. Aplicación del patrón MVVM en Android con Kotlin

En Android con Kotlin, el patrón MVVM se puede aplicar de la siguiente manera:

Modelo (Model): en Android, el modelo se puede implementar utilizando la arquitectura de componentes de Android, la cual incluye componentes como ViewModel, Room y Retrofit, que permiten interactuar con la base de datos, la red y otros servicios.

Vista (View): en Android, la vista se puede implementar utilizando los elementos de la interfaz de usuario de Android, como actividades (Activities), fragmentos (Fragments) y vistas personalizadas (Custom Views).

Modelo de vista (View-Model): en Android, el modelo de vista se puede implementar utilizando la clase ViewModel, la cual se encarga de mantener los datos relacionados con la vista y de manejar los cambios de configuración de la actividad o el fragmento. La clase ViewModel también se encarga de manejar la comunicación con el modelo y de exponer los datos a la vista a través de LiveData.

En resumen, en Android con Kotlin, el modelo se implementa utilizando los componentes de la arquitectura de componentes de Android, la vista se implementa utilizando los elementos de la interfaz de usuario de Android y el modelo de vista se implementa utilizando la clase ViewModel.

## **2.4 Ventajas y desventajas del patrón MVVM en el desarrollo de aplicaciones móviles.**

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

Separación de responsabilidades: el patrón MVVM permite separar la lógica de negocio de la aplicación de la interfaz de usuario, lo cual facilita la mantenibilidad y evolución de la aplicación.

Facilidad de prueba: el patrón MVVM permite separar la lógica de negocio de la interfaz de usuario, lo cual facilita la creación de pruebas unitarias y de integración.

Reutilización de código: el patrón MVVM permite reutilizar el código del modelo y el modelo de vista en diferentes vistas, lo cual reduce la cantidad de código repetitivo en la aplicación.

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

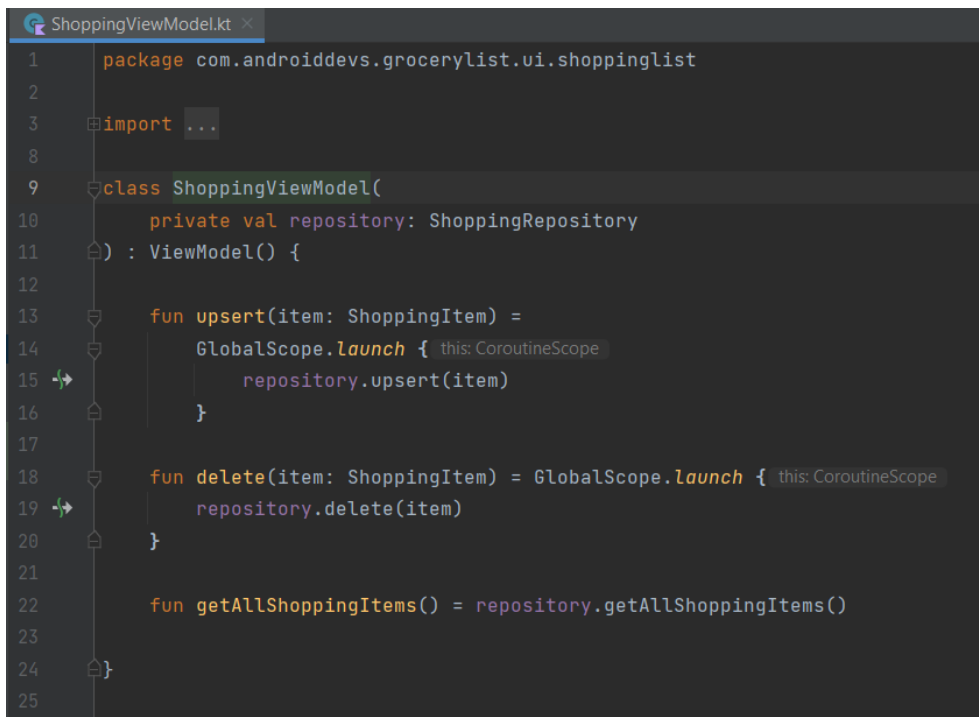
Complejidad: el patrón MVVM puede aumentar la complejidad de la aplicación, especialmente en aplicaciones pequeñas y simples.

Curva de aprendizaje: el patrón MVVM puede requerir una curva de aprendizaje para aquellos desarrolladores que no estén familiarizados con él.

Aumento de la cantidad de clases: el patrón MVVM puede aumentar la cantidad de clases en la aplicación, lo cual puede aumentar la complejidad y el tiempo de desarrollo.

## 3.0 Anexos

Código de ejemplo de aplicación Android con Kotlin implementando el patrón MVVM:



```
1 package com.androiddevs.grocerylist.ui.shoppinglist
2
3 import ...
4
5
6
7
8
9 class ShoppingViewModel(
10     private val repository: ShoppingRepository
11 ) : ViewModel() {
12
13     fun upsert(item: ShoppingItem) =
14         GlobalScope.launch { this: CoroutineScope
15             repository.upsert(item)
16         }
17
18     fun delete(item: ShoppingItem) = GlobalScope.launch { this: CoroutineScope
19         repository.delete(item)
20     }
21
22     fun getAllShoppingItems() = repository.getAllShoppingItems()
23
24 }
25
```

```

ShoppingViewModelFactory.kt
1  package com.androiddevs.grocerylist.ui.shoppinglist
2
3  import ...
4
5
6
7  @Suppress("UNCHECKED_CAST")
8  class ShoppingViewModelFactory(
9      private val repository: ShoppingRepository
10 ) : ViewModelProvider.NewInstanceFactory() {
11
12      override fun <T : ViewModel?> create(modelClass: Class<T>): T {
13          return ShoppingViewModel(repository) as T
14      }
15 }

```

```

ShoppingActivity.kt
1  package com.androiddevs.grocerylist.ui.shoppinglist
2
3  import ...
4
5
6
7
8
9
10
11
12
13
14
15
16
17 class ShoppingActivity : AppCompatActivity(), KodeinAware {
18
19     override val kodein by kodein()
20     private val factory: ShoppingViewModelFactory by instance()
21
22     lateinit var viewModel: ShoppingViewModel
23
24     override fun onCreate(savedInstanceState: Bundle?) {
25         super.onCreate(savedInstanceState)
26         setContentView(R.layout.activity_shopping)
27
28         viewModel = ViewModelProvider(owner, this, factory).get(ShoppingViewModel::class.java)
29
30         val adapter = ShoppingItemAdapter(listOf(), viewModel)
31
32         rvShoppingItems.layoutManager = LinearLayoutManager(context, this)
33         rvShoppingItems.adapter = adapter
34
35         viewModel.getAllShoppingItems().observe(owner, this, Observer { it: List<ShoppingItem>! }) {
36             adapter.items = it
37             adapter.notifyDataSetChanged()
38         })
39
40     }

```



```

ShoppingDao.kt
1 package com.androiddevs.grocerylist.data.db
2
3 import ...
4
5
6
7 @Dao
8 interface ShoppingDao {
9
10     @Insert(onConflict = OnConflictStrategy.REPLACE)
11     suspend fun upsert(item: ShoppingItem)
12
13     @Delete
14     suspend fun delete(item: ShoppingItem)
15
16     @Query("SELECT * FROM shopping_items")
17     fun getAllShoppingItems(): LiveData<List<ShoppingItem>>
18 }

```

```

ShoppingDatabase.kt
1 package com.androiddevs.grocerylist.data.db
2
3 import ...
4
5
6
7
8
9 @Database(
10     entities = [ShoppingItem::class],
11     version = 1
12 )
13 abstract class ShoppingDatabase: RoomDatabase() {
14
15     abstract fun getShoppingDao(): ShoppingDao
16
17     companion object {
18         @Volatile
19         private var instance: ShoppingDatabase? = null
20         private val LOCK = Any()
21
22         operator fun invoke(context: Context) = instance
23             ?: synchronized(LOCK) {
24                 instance
25                 ?: createDatabase(
26                     context
27                 ).also { instance = it }
28             }
29     }
30 }

```

```

ShoppingItem.kt
1 package com.androiddevs.grocerylist.data.db.entities
2
3 import ...
4
5
6
7 @Entity(tableName = "shopping_items")
8 data class ShoppingItem(
9     @ColumnInfo(name = "item_name")
10    var name: String,
11    @ColumnInfo(name = "item_amount")
12    var amount: Int
13 ) {
14     @PrimaryKey(autoGenerate = true)
15     var id: Int? = null
16 }

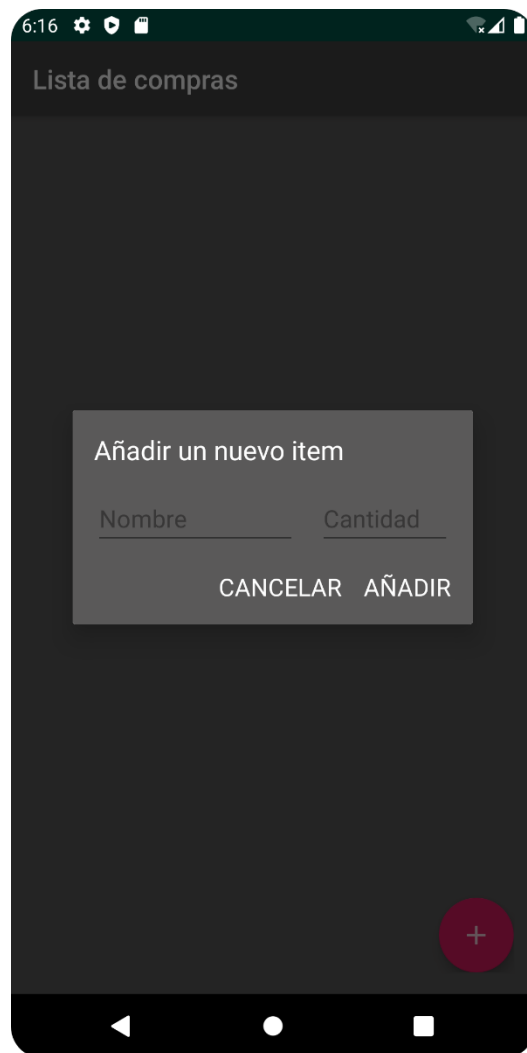
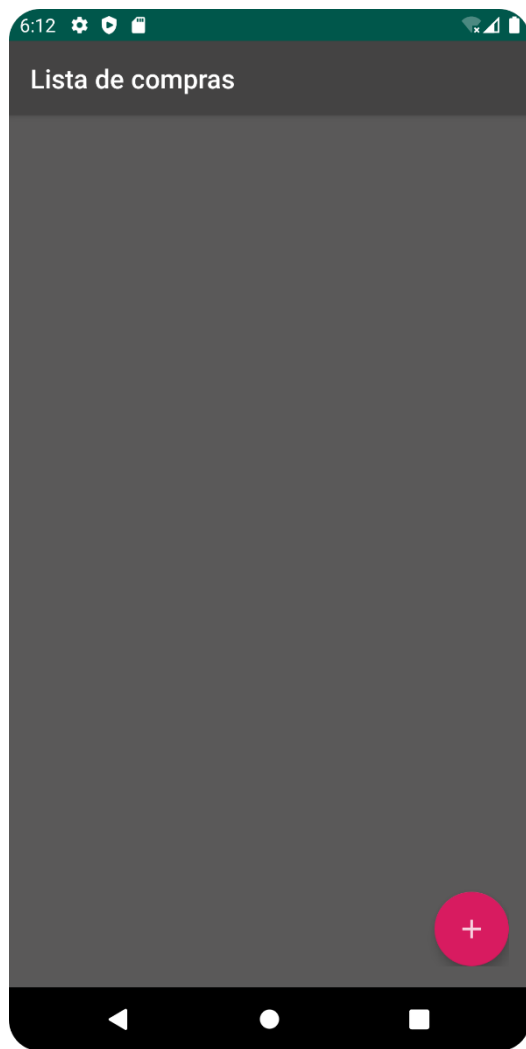
```

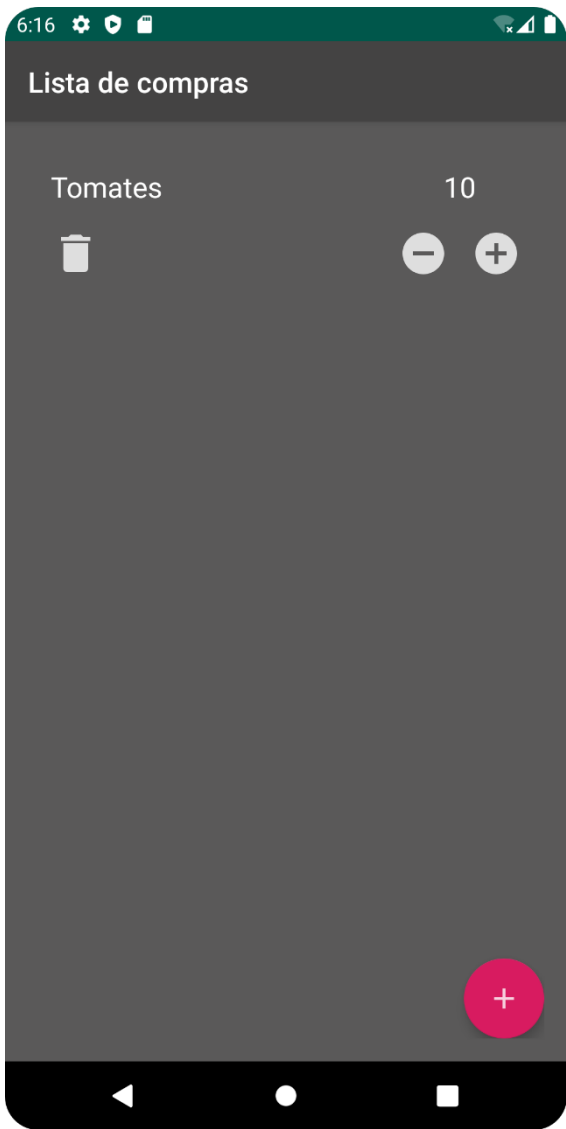
```

ShoppingItemAdapter.kt
1 package com.androiddevs.grocerylist.other
2
3 import ...
4
5
6
7 class ShoppingItemAdapter(
8     var items: List<ShoppingItem>,
9     private val viewModel: ShoppingViewModel
10 ): RecyclerView.Adapter<ShoppingItemAdapter.ShoppingViewHolder>() {
11
12     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ShoppingViewHolder {
13         val view = LayoutInflater.from(parent.context).inflate(R.layout.shopping_item, parent, attachToRoot: false)
14         return ShoppingViewHolder(view)
15     }
16
17     override fun getItemCount(): Int {
18         return items.size
19     }
20
21     override fun onBindViewHolder(holder: ShoppingViewHolder, position: Int) {
22         val curShoppingItem = items[position]
23
24         holder.itemView.tvName.text = curShoppingItem.name
25         holder.itemView.tvAmount.text = "${curShoppingItem.amount}"
26
27         holder.itemView.ivDelete.setOnClickListener { it View!
28             viewModel.delete(curShoppingItem)
29         }
30     }
31 }

```

## Interfaz de la aplicación





url del repositorio:

<https://github.com/KevinMajano/Patron-MVVM>

## 4.0 Bibliografia

Microsoft. (2005). Introduction to Model/View/ViewModel pattern for building WPF apps. <https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern>

Google. (s.f.). Android Architecture Components. <https://developer.android.com/topic/libraries/architecture>

Fernandez, J. (2021). The Pros and Cons of the MVVM Architecture in Android. <https://betterprogramming.pub>