Patrón MVP

El patrón MVP (Model View Presenter) es un derivado del conocido MVC (Model View Controller), y uno de los patrones más populares para organizar la capa de presentación en las aplicaciones de Android.

Este artículo se publicó lo publiqué por primera vez en 2014 en mi blog en inglés, y ha sido el más popular desde entonces. Así que he decidido traducirlo al español para que lo tengas aquí también.

Desde entonces se han producido cambios importantes en los patrones de arquitectura, como [MVVM con architecture components](https://devexperto.com/mvvm-vs-mvp/), pero MVP sigue siendo válido y es una opción a tener en cuenta.

## **¿Qué es MVP?**

El patrón MVP permite **separar la capa de presentación de la lógica** para que todo sobre cómo funciona la interfaz de usuario sea independiente de cómo lo representamos en la pantalla. Idealmente, el patrón MVP lograría que la misma lógica pudiera tener vistas completamente diferentes e intercambiables.

Lo primero que hay que aclarar es que MVP **no es una arquitectura en sí misma**, solo es responsable de la capa de presentación. Varias personas me han rebatido esto, por lo que quiero explicarlo un poco más a fondo.

Puedes ver en muchos sitios que MVP se define como un patrón de arquitectura porque puede convertirse en parte de la arquitectura de su aplicación, pero no consideres que solo porque estás usando MVP, tu arquitectura está completa. **MVP solo modela la capa de presentación**, pero el resto de capas aún requerirá una buena arquitectura si quieres una aplicación flexible y escalable.

Un ejemplo de una arquitectura completa podría ser [Clean Architecture](https://devexperto.com/clean-architecture-android/), aunque hay muchas otras opciones.

En cualquier caso, siempre es mejor usarlo para su arquitectura que no usarlo en absoluto.

## **¿Por qué usar MVP?**

En Android, tenemos un problema derivado del hecho de que las actividades de Android están estrechamente relacionadas con la UI y los mecanismos de acceso a datos. Podemos encontrar ejemplos extremos como *CursorAdapter*, que mezcla adaptadores, que son parte de la vista, con cursores, algo que debería relegarse a las profundidades de la capa de acceso a datos.

**Para que una aplicación sea fácilmente extensible y fácil de mantener, necesitamos definir capas bien separadas**. ¿Qué haremos mañana si, en lugar de recuperar los mismos datos de una base de datos, necesitamos hacerlo desde un servicio web? Tendríamos que rehacer toda nuestra visión.

MVP hace que las vistas sean independientes de nuestra fuente de datos. Dividimos la aplicación en al menos tres capas diferentes, lo que nos permite probarlas de forma independiente. Con MVP **sacamos la mayor parte de la lógica de las actividades para poder probarla sin usar tests de instrumentación**.

## **Cómo implementar MVP para Android**

Bueno, aquí es donde todo comienza a volverse más difuso. Hay muchas variaciones de MVP y todos pueden ajustar el patrón a sus necesidades y la forma en que se sienten más cómodos. Varía dependiendo básicamente del número de responsabilidades que delegamos al presentador.

¿Es la vista responsable de habilitar o deshabilitar una barra de progreso, o debe ser realizada por el presentador? ¿Y quién decide qué acciones deben mostrarse en la ActionBar? Ahí es donde comienzan las decisiones difíciles. Mostraré cómo trabajo habitualmente, pero quiero que lo tomes con espíritu crítico, porque no hay una forma “estándar” de implementarlo.

Para representar lo que cuento aquí, he implementado un ejemplo muy simple que [puedes encontrar en mi Github](https://github.com/antoniolg/androidmvp) con una pantalla de inicio de sesión y una pantalla principal. Por razones de simplicidad, el código en el artículo está en Kotlin, pero también puede verificar el código en Java 8 [en el repositorio](https://github.com/antoniolg/androidmvp).

### **El modelo**

En una aplicación con una arquitectura por capas completa, este modelo solo sería la puerta de entrada a la capa de dominio o la lógica empresarial. Si estuviéramos usando la [clean architecture de Uncle Bob](http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html), el modelo probablemente sería un interactor que implementa un caso de uso. Pero para el propósito de este artículo, es suficiente verlo como el proveedor de los datos que queremos mostrar en la vista.

Si echas un ojo al código, verás que he creado dos mocks de interactors con delays artificiales para simular peticiones a un servidor. Esta es la estructura de uno de estos interactors:

class LoginInteractor {

...

fun login(username: String, password: String, listener: OnLoginFinishedListener) {

// Mock login. I'm creating a handler to delay the answer a couple of seconds

postDelayed(2000) {

when {

username.isEmpty() -> listener.onUsernameError()

password.isEmpty() -> listener.onPasswordError()

**else** -> listener.onSuccess()

}

}

}

}

Es una función simple que recibe el nombre de usuario y la contraseña, y realiza algunas validaciones.

### **The View**

La vista, generalmente implementada por una Activity (puede ser un Fragment, una View… dependiendo de cómo esté estructurada la aplicación), contendrá una referencia al presentador. El presentador será idealmente provisto por un inyector de dependencias como [Dagger](https://dagger.dev/), pero en caso de que no use algo como esto, esta vista será la responsable de crear el objeto presentador. Lo único que hará la vista es llamar a un método de presentador cada vez que haya una acción del usuario (por ejemplo, hacer clic en un botón).

Como el presentador debe ser independiente de la vista, utiliza una interfaz que debe ser implementada. Aquí está la interfaz que usa el ejemplo:

interface LoginView {

fun showProgress()

fun hideProgress()

fun setUsernameError()

fun setPasswordError()

fun navigateToHome()

}

Tiene algunos métodos de utilidad para mostrar y ocultar el progreso, mostrar errores, navegar a la siguiente pantalla… Como se mencionó anteriormente, hay muchas maneras de hacer esto, pero prefiero mostrar la más simple.

Entonces, la actividad puede implementar esos métodos. Aquí te muestro algunos, para que tengas una idea:

class LoginActivity : AppCompatActivity(), LoginView {

...

override fun showProgress() {

progress.visibility = View.VISIBLE

}

override fun hideProgress() {

progress.visibility = View.GONE

}

override fun setUsernameError() {

username.error = getString(R.string.username\_error)

}

}

Pero si recuerdas, también te dije que la vista usa el presentador para notificar sobre las interacciones del usuario. Así es como se usa:

class LoginActivity : AppCompatActivity(), LoginView {

private val presenter = LoginPresenter(**this**, LoginInteractor())

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_login)

button.setOnClickListener { validateCredentials() }

}

private fun validateCredentials() {

presenter.validateCredentials(username.text.toString(), password.text.toString())

}

override fun onDestroy() {

presenter.onDestroy()

super.onDestroy()

}

...

}

El Presenter se define como una property para la actividad, y cuando se hace clic en el botón, llama a validateCredentials(), que notificará al Presenter.

Lo mismo sucede con onDestroy(). Más adelante veremos por qué se necesita notificar en ese caso.

### **El Presenter**

El Presenter es responsable de actuar **como intermediario entre la vista y el modelo**. Recupera datos del modelo y los devuelve formateados a la vista.

Además, a diferencia del MVC típico, decide qué sucede cuando se interactúa con la vista. Por lo tanto, tendrá un método para cada posible acción que el usuario pueda hacer. Lo vimos en la vista, pero aquí está la implementación:

class LoginPresenter(var loginView: LoginView?, val loginInteractor: LoginInteractor) :

LoginInteractor.OnLoginFinishedListener {

fun validateCredentials(username: String, password: String) {

loginView?.showProgress()

loginInteractor.login(username, password, **this**)

}

...

}

**MVP tiene algunos riesgos**, y lo más importante de lo que nos solemos olvidar es que el presentador está conectado a la vista para siempre. Y la vista es una actividad, lo que significa que:

* Podemos producir un leak de la actividad si ejecutamos tareas largas en segundo plano
* Podemos intentar actualizar una activity que ya haya muerto

Para el primer punto, si puedes asegurarte de que tus tareas en segundo plano finalicen en un período de tiempo razonable, no le daría muchos vueltas. Producir un leak de una actividad durante 5-10 segundos no hará que tu aplicación sea mucho peor, y las soluciones para esto suelen ser complejas.

El segundo punto es más preocupante. Imagina que envías una solicitud a un servidor que tarda 10 segundos, pero el usuario cierra la actividad después de 5 segundos. Cuando se llama al callback y se actualiza la IU, lanzará un error porque la actividad está finalizando.

Para resolver esto, llamamos al método onDestroy() que limpia la vista:

fun onDestroy() {

loginView = **null**

}

De esta forma evitamos llamar a una vista que esté en un estado inconsistente.

## **Conclusión**

Separar la interfaz de la lógica en Android no es fácil, pero el patrón MVP hace que sea más sencillo evitar que nuestras actividades terminen degradando en clases muy acopladas que consisten en cientos o incluso miles de líneas. En aplicaciones grandes, es esencial organizar bien nuestro código. De lo contrario, se hace imposible mantenerlas y extenderlas.

Recuerda que [tienes el repositorio](https://github.com/antoniolg/androidmvp), donde puedes ver el código tanto en Kotlin como en Java