¡Te doy la bienvenida!

Muchas gracias por tu interés en esta guía. Para mí es un honor poder ayudarte a avanzar en tus conocimientos para que puedas crear código de calidad.

La guía está basada en tres artículos que escribí en el blog, y que tiene mucho sentido aprender uno detrás de otro.

Tómate tu tiempo para digerirla y releerla. Si nunca habías visto estos conceptos antes, te va a llevar un tiempo.

¿Quién soy yo?

Mi nombre es Antonio Leiva, y llevo desde el 2012 ayudando a otros desarrolladores a evolucionar en su carrera profesional. Soy formador y Kotlin Certified Trainer por JetBrains.

Escribí hace un tiempo un libro llamado **Kotlin for Android Developers**, en el que explico cómo pasar de Java a Kotlin. También existe el curso online y la formación para empresas.

Además **soy el creador del programa Architect Coders**, donde ayudo a desarrolladores Android a convertirse en verdaderos ingenieros, dominando todos los conocimientos que las grandes empresas y startups del sector demandan. Puedes apuntarte aquí para que te informe de la próxima edición.

Si quieres conectar conmigo en las redes sociales, puedes encontrarme en:

- Instagram
- YouTube
- Canal de Telegram
- Twitter
- Facebook
- Podcast

Y si has llegado a esta página por otra vía, puedes suscribirte aquí y llevarte de regalo la guía para iniciarte en Kotlin.

¡Empezamos!

MVP para Android: Cómo organizar la capa de presentación

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, 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, 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 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.

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, 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:

```
1. class LoginInteractor {
2.
3.     ...
4.
5.     fun login(username: String, password: String,
   listener: OnLoginFinishedListener) {
6.         // Mock login. I'm creating a handler to delay the
   answer a couple of seconds
7.         postDelayed(2000) {
8.             when {
9.                 username.isEmpty() ->
   listener.onUsernameError()
10.                password.isEmpty() ->
   listener.onPasswordError()
11.                else -> listener.onSuccess()
12.            }
13.        }
14.    }
```

```
15.     }
```

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

La Vista

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, 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:

```
1. interface LoginView {
2.     fun showProgress()
3.     fun hideProgress()
4.     fun setUsernameError()
5.     fun setPasswordError()
6.     fun navigateToHome()
7. }
```

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:

```
1. class LoginActivity : AppCompatActivity(), LoginView {
2.     ...
3.
4.     override fun showProgress() {
5.         progress.visibility = View.VISIBLE
6.     }
7.
8.     override fun hideProgress() {
9.         progress.visibility = View.GONE
10.      }
11.
12.      override fun setUsernameError() {
13.          username.error =
   getString(R.string.username_error)
14.      }
15.  }
```

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

```
1. class LoginActivity : AppCompatActivity(), LoginView {
2.
3.     private val presenter = LoginPresenter(this,
   LoginInteractor())
4.
5.     override fun onCreate(savedInstanceState: Bundle?) {
6.         super.onCreate(savedInstanceState)
```

```
7.          setContentView(R.layout.activity_login)
8.
9.          button.setOnClickListener { validateCredentials()
    }
10.       }
11.
12.       private fun validateCredentials() {
13.
    presenter.validateCredentials(username.text.toString(),
    password.text.toString())
14.       }
15.
16.       override fun onDestroy() {
17.           presenter.onDestroy()
18.           super.onDestroy()
19.       }
20.       ...
21.   }
```

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:

```
1. class LoginPresenter(var loginView: LoginView?, val
   loginInteractor: LoginInteractor) :
2.    LoginInteractor.OnLoginFinishedListener {
3.
4.    fun validateCredentials(username: String, password:
   String) {
5.        loginView?.showProgress()
6.        loginInteractor.login(username, password, this)
7.    }
8.    ...
9. }
```

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:

```
1. fun onDestroy() {
2.     loginView = null
3. }
```

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, donde puedes ver el código tanto en Kotlin como en Java

De patrones de presentación, arquitecturas, testing… hablamos a fondo en mi programa Architect Coders. Así que si te interesa, puedes unirte para la próxima edición.

MVVM con Architecture Components: Una guía paso a paso para los amantes de MVP

Bien, ahora que MVVM es el estándar para implementar aplicaciones de Android desde que Google lanzó su Guía de arquitectura de aplicaciones, creo que es el momento de **proporcionar información fácil para comprender el patrón de MVVM desde la perspectiva de un usuario de MVP.**

Si has llegado aquí por casualidad, pero no sabes lo que es MVP o cómo usarlo en Android, te recomiendo que primero eches un vistazo a este artículo sobre el tema que escribí hace algún tiempo.

También tienes la opción de ver este contenido en YouTube (en inglés):

MVVM vs MVP – ¿Necesito refactorizar todas mis app ahora?

Durante mucho tiempo, MVP parece ha sido el patrón de presentación más utilizado para aislar a la interfaz de usuario de la lógica de negocio, pero ahora hay un nuevo sheriff en la ciudad.

Muchas personas me preguntaron si deberían huir de MVP o qué hacer cuando inician una nueva aplicación. Estos son algunas ideas al azar al respecto:

- **MVP no esta muerto.** Sigue siendo un patrón perfectamente válido que puede seguir usando como antes.
- **MVVM como patrón, no es necesariamente mejor**. La implementación específica que hizo Google tiene sentido, pero hay una razón por la cual MVP se usaba antes: simplemente encaja muy bien con el framework de Android y con poca complejidad.
- **Usar MVP no significa que no se puedan usar el resto de los componentes de la arquitectura.** Probablemente el `ViewModel` no tiene mucho sentido (ya que es el sustituto natural del presentador), pero el resto de los componentes se pueden usar de una forma u otra.
- **No necesitas refactorizar tu aplicación de inmediato.** Si estás contento con MVP, continúa con ello. En general, es mejor mantener una arquitectura sólida en lugar de tener todas las nuevas tendencias implementadas en diferentes pantallas de la aplicación. Refactorizaciones como esta no vienen sin coste añadido.

Diferencias entre MVVM y MVP

Afortunadamente, si ya conocse MVP **¡Aprender MVVM es extremadamente fácil!** Solo hay una pequeña diferencia, al menos en la forma en que generalmente se aplica a Android:

En MVP, el presentador se comunica con la vista a través de una interfaz. En MVVM, el ViewModel se comunica con la vista usando el patrón Observer.

Sé que, si lees la definición original del patrón MVVM, no coincidirá exactamente con lo que dije antes. Pero en el caso particular de Android, y si dejamos databiding a un lado, en mi opinión **esta es la mejor manera de entender cómo funciona**.

Migrando de MVP a MVVM *sin Arch Components*

Lo que estoy haciendo aquí es adaptar el ejemplo que hice para MVP (puedes echar un vistazo al repositorio aquí) para usar MVVM. El nuevo repositorio está aquí.

Estoy excluyendo los Architecture Components por ahora, para que absorbamos la idea primero. Luego podemos ver cómo funciona el nuevo «framework» que Google creó, los puntos en los que facilita las cosas.

Creando una clase Observable

Como estamos usando el patrón *Observer*, necesitamos una clase que se pueda observar. Esta clase contendrá los observadores y un tipo genérico para el valor que se enviará a estos observadores. Cuando el valor cambia, los observadores son notificados:

```
1. class Observable<T> {
2.
3.     private var observers = emptyList<(T) -> Unit>()
4.
5.     fun addObserver(observer: (T) -> Unit) {
```

```
        observers += observer
    }

    fun clearObservers() {
        observers = emptyList()
    }

    fun callObservers(newValue: T) {
        observers.forEach { it(newValue) }
    }

}
```

Usando estados que representan las UI

Como ahora no tenemos una manera de comunicarnos directamente con la vista, no podemos decirle qué hacer. La forma en que encontré más flexible es tener modelos que representen el estado de la interfaz de usuario.

Por ejemplo, si queremos que muestre su progreso, enviaremos el estado `Loading`. La forma de consumir ese estado depende totalmente de la vista.

Para este caso en particular, creé una clase llamada `ScreenState`, el cual acepta un tipo genérico que representará el estado específico que necesita la view.

Puede haber algunos estados genéricos que se apliquen a todas las pantallas, como `Loading` (también podría pensar acerca de un estado de `Error`) y luego uno específico por pantalla.

El `ScreenState` general se puede modelar usando una sealed class como esta:

```
sealed class ScreenState<out T> {
```

```
2.     object Loading : ScreenState<Nothing>()
3.     class Render<T>(val renderState: T) :
   ScreenState<T>()
4. }
```

Entonces los estados específicos pueden tener cualquier estructura que necesitemos. Para el estado de inicio de sesión, un enumerado es suficiente:

```
1. enum class LoginState {
2.     Success, WrongUserName, WrongPassword
3. }
```

Pero para `MainState`, ya que estamos mostrando una lista de elementos y un mensaje, el enum no nos daría suficiente flexibilidad. Así que la sealed class es de nuevo extremadamente útil (veremos más adelante por qué):

```
1. sealed class MainState {
2.     class ShowItems(val items: List<String>) :
   MainState()
3.     class ShowMessage(val message: String) : MainState()
4. }
```

Convirtiendo los presentadores a ViewModels

Lo primero que ya no necesitamos es la interfaz `View`. Puedes deshacerte de ella (y del argumento `View`) , porque usaremos un observable en su lugar.

Para definirlo:

```
1. val stateObservable =
   Observable<ScreenState<LoginState>>()
```

Luego, cuando queremos mostrar un progreso que indica que se está ejecutando un proceso, simplemente llamamos a los observadores con el estado `Loading`.

```
1. fun validateCredentials(username: String, password:
   String) {
2.     stateObservable.callObservers(ScreenState.Loading)
3.     loginInteractor.login(username, password, this)
4. }
```

Cuando el inicio de sesión finaliza, le pedimos que muestre el éxito:

```
1. override fun onSuccess() {
2.
   stateObservable.callObservers(ScreenState.Render(LoginSta
   te.Success))
3. }
```

Realmente este estado anterior se puede modelar de muchas maneras diferentes. Si queremos ser más explícitos, podríamos decir que puede navegar a la pantalla principal con un `LoginState.NavigateToMain` o similar.

Pero como esto depende de muchos factores dependiendo de la estructura de la aplicación, lo dejaré así.

Luego, en el `onDestroy` del ViewModel, eliminamos a los observadores para no leakearlos.

```
1. fun onDestroy() {
```

```
2.     stateObservable.clearObservers()
3. }
```

Usando el ViewModel de la Actividad

La Actividad ahora no puede actuar como la View de ViewModel, de modo que ahí es donde el patrón de observador se pone en acción.
Primero, crea una propiedad que contenga el ViewModel:

```
1. private val viewModel =
   LoginViewModel(LoginInteractor())
```

Luego, en `onCreate`, puedes comenzar a observar el estado. Cuando el estado se actualice, llamará al método `updateUI`:

```
1. override fun onCreate(savedInstanceState: Bundle?) {
2.     super.onCreate(savedInstanceState)
3.     ...
4.     viewModel.stateObservable.addObserver(::updateUI)
5. }
```

Aquí, gracias a las *sealed classes* y las *enums*, al usar la expresión `when`, todo se vuelve bastante fácil. Estoy procesando el estado en dos pasos: primero los estados generales y luego el `LoginState` particular.

El primer `when` mostrará un progreso con el estado de `Loading`, y llamará a otra función si tiene que representar el estado específico:

```
1. private fun updateUI(screenState:
   ScreenState<LoginState>) {
2.     when (screenState) {
```

```
3.          ScreenState.Loading -> progress.visibility =
    View.VISIBLE
4.          is ScreenState.Render ->
    processLoginState(screenState.renderState)
5.      }
6.  }
```

Este segundo oculta el progreso (en caso de que estuviera visible), navega a la siguiente actividad si el inicio de sesión fue exitoso o muestra un error según el tipo de error:

```
1.  private fun processLoginState(renderState: LoginState) {
2.      progress.visibility = View.GONE
3.      when (renderState) {
4.          LoginState.Success -> startActivity(Intent(this,
    MainActivity::class.java))
5.          LoginState.WrongUserName -> username.error =
    getString(R.string.username_error)
6.          LoginState.WrongPassword -> password.error =
    getString(R.string.password_error)
7.      }
8.  }
```

Cuando se hace clic en el botón, llamamos a `ViewModel` para que pueda hacer su trabajo:

```
1.  private fun onLoginClicked() {
2.      viewModel.onLoginClicked(username.text.toString(),
    password.text.toString())
3.  }
```

Y luego, en `onDestroy()`, llamamos a destruir el ViewModel (para que pueda limpiar los observadores):

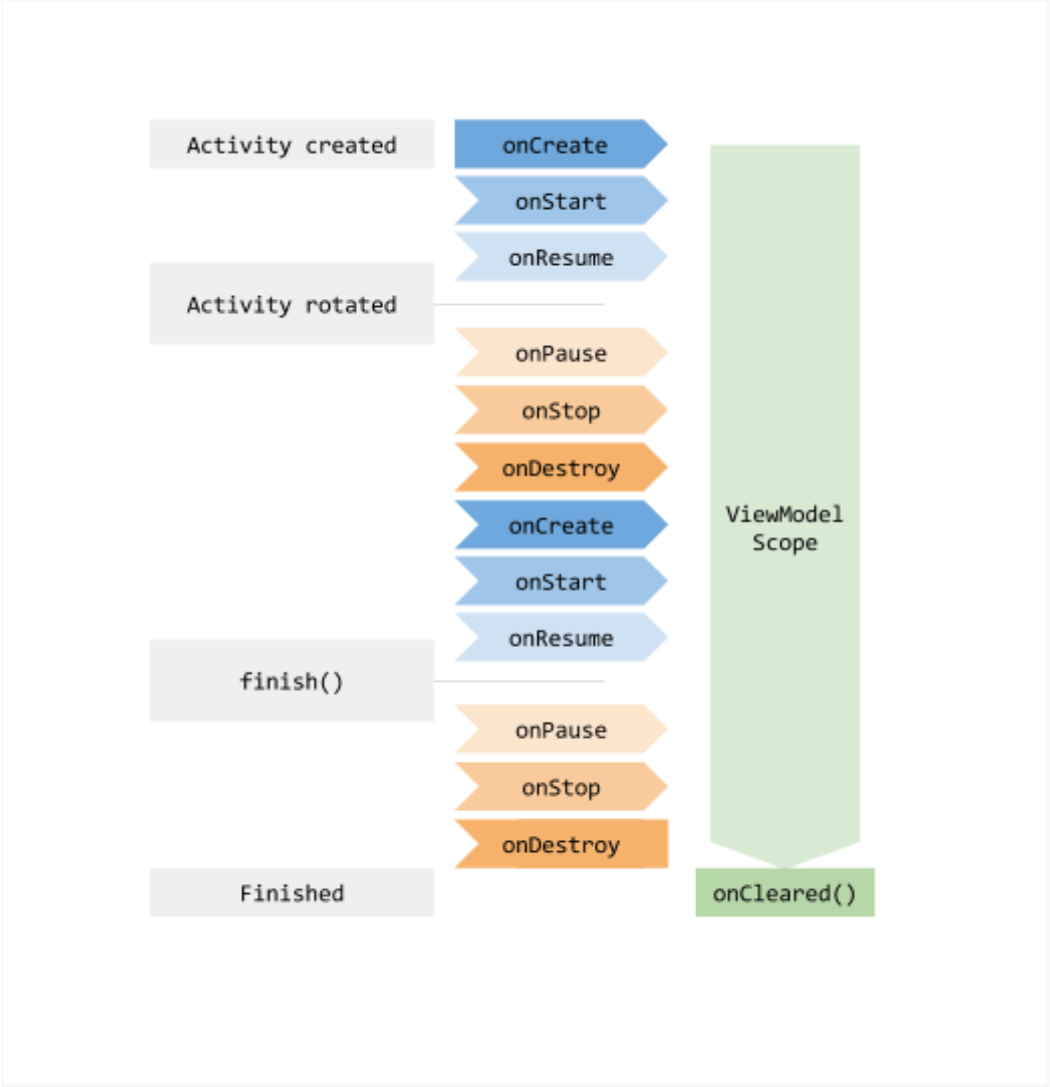
```
override fun onDestroy() {
    viewModel.onDestroy()
    super.onDestroy()
}
```

Cambiando el código para usar los Architecture Components

Por ahora, hemos tomado una solución a medida como un paso intermedio a MVVM, para que pueda ver las diferencias fácilmente. Hasta ahora, no hay muchos beneficios en comparación con MVP.

Pero sí que hay algunos. El más importante es te puedes olvidar de si la actividad se destruye o no, por lo que puedes desvincularte de su ciclo de vida y hacer tu trabajo en cualquier momento. Gracias a `ViewModel` y `LiveData`, no necesitas preocuparte cuando la actividad se recrea o cuando se destruye.

Así es como funciona: mientras se recrea la actividad, `ViewModel` se mantiene vivo. Es justo cuando la actividad termina para siempre, cuando se llama al método `onCleared()` de `ViewModel`.



Tomado de developers.android.com

Como `LiveData` también es consciente del ciclo de vida, sabe cuándo debe engancharse y desconectarse del propietario del ciclo de vida, por lo que no es necesario que lo hagas tú.

No quiero profundizar más en cómo funcionan los Architecture Components (se explica detalladamente en la guía para desarrolladores, pero puedo escribir otro artículo si crees que puede ser de ayuda), así que continuemos con la implementación.

Para usar los Architecture Components en nuestro proyecto, necesitamos agregar esta dependencia:

```
implementation "android.arch.lifecycle:extensions:1.1.1"
```

Hay varias bibliotecas para los architecture components y diferentes formas de incluirlos (dependiendo de si usas AndroidX o no, por ejemplo). Así que si tienes necesidades especiales, echa un vistazo aquí.

Architecture Components ViewModel

Para cambiar a `ViewModel`, solo necesitas extender la clase de la librería:

```
class LoginViewModel(private val loginInteractor:
LoginInteractor) : ViewModel()
```

Elimine `onDestroy()`, ya no hace falta. Podemos mover su código a `onCleared()`. Así que de esa manera no necesitamos observar en `onCreate`y dejar de observar en `onDestroy`, sino que lo hacemos justo cuando se está borrando el `ViewModel`.

```
override fun onCleared() {
    stateObservable.clearObservers()
    super.onCleared()
}
```

Ahora, volviendo a la actividad, crea una property para el `ViewModel`. Es necesario que se inicie *lateinit* porque se asignará en `onCreate`:

```
private lateinit var viewModel : LoginViewModel

override fun onCreate(savedInstanceState: Bundle?) {
    ...
    viewModel =
ViewModelProviders.of(this)[LoginViewModel::class.java]
```

```
6. }
```

Ese es el caso ideal, cuando el `ViewModel` no recibe argumentos. Pero si queremos que el `ViewModel` reciba argumentos a través del constructor, debes declarar una `Factory`. Este es el camino (un poco complicado, lo sé ...):

```
1. class LoginViewModelFactory(private val loginInteractor:
   LoginInteractor) :
2.     ViewModelProvider.NewInstanceFactory() {
3.
4.     override fun <T : ViewModel?> create(modelClass:
   Class<T>): T {
5.         return LoginViewModel(loginInteractor) as T
6.     }
7. }
```

Y para recuperar el `ViewModel`, el código cambia un poco:

```
1. ViewModelProviders.of(
2.     this,
3.     LoginViewModelFactory(LoginInteractor())
4. )[LoginViewModel::class.java]
```

Reemplazar Observable por LiveData

El `LiveData` puede sustituir nuestra clase `Observable` sin problema. Una cosa a tener en cuenta es que `LiveData` es inmutable por defecto (no puedes cambiar su valor).

Esto es genial, porque queremos que sea público para que los observadores puedan suscribirse, pero no queremos que otras partes del código cambien el valor.

Pero, por otro lado, los datos deben ser mutables, ¿para qué lo observaríamos si no va a cambiar? Para conseguir esto, el truco es usar el equivalente a un campo privado más un getter público con un tipo de retorno más restrictivo. **En el caso de Kotlin, sería una propiedad privada más una pública:**

```
1. private val _loginState:
   MutableLiveData<ScreenState<LoginState>> =
   MutableLiveData()

1. val loginState: LiveData<ScreenState<LoginState>>
2.     get() = _loginState
```

Y ya no necesitamos `onCleared()`, ya que `LiveData` también está suscrito al ciclo de vida. Sabrá el momento adecuado para dejar de ser observado.

Para observarlo, la forma más limpia es la siguiente:

```
1. viewModel.loginState.observe(::getLifecycle, ::updateUI)
```

Echa un vistazo a mi artículo sobre referencias de funciones si encuentras esta línea confusa.

La definición de `updateUI` requiere un `ScreenState` como argumento, para que se ajuste al valor de retorno de `LiveData` y puedo usarlo como referencia de función:

```
1. private fun updateUI(screenState:
   ScreenState<LoginState>?) {
```

```
    ...
}
```

El `MainViewModel` tampoco requiere `onResume()`. En su lugar, podemos anular el getter de la property y ejecutar la solicitud la primera vez que se observe el `LiveData`:

```
private lateinit var _mainState:
MutableLiveData<ScreenState<MainState>>

val mainState: LiveData<ScreenState<MainState>>
    get() {
        if (!::_mainState.isInitialized) {
            _mainState = MutableLiveData()
            _mainState.value = ScreenState.Loading
            findItemsInteractor.findItems(::onItemsLoaded)
        }
        return _mainState
    }
```

Y el código para esta actividad es muy similar al otro:

```
viewModel.mainState.observe(::getLifecycle, ::updateUI)
```

Para terminar

El código anterior parece un poco más complicado, pero eso se debe principalmente al uso de un nuevo framework y hasta que haces a su uso.

Es verdad que hay nuevo boilerplate, como el de `ViewModelFactory`, la búsqueda de `ViewModel`, o las dos properties necesarias para evitar la edición de `LiveData` desde fuera. **Simplifiqué algunas de ellas en este**

artículo **usando algunas funciones de Kotlin** y pueden ayudarte a sentirte más cómodo con el código. Pero no quería usarlos aquí por simplicidad.

Como mencioné al principio, si decides usar **MVP** o **MVVM** es totalmente tu decisión. **No creo que haya una necesidad de migrar si tu arquitectura está ya resueta utilizando MVP, pero es interesante tener una idea de cómo funciona MVVM** porque lo necesitarás tarde o temprano.

Creo que todavía estamos en un punto en el que estamos tratando de descubrir las mejores formas de usar **MVVM** con Architecture Components en nuestras aplicaciones Android y estoy seguro de que mi solución no es perfecta. Así que hazme saber todo lo que te preocupa o con lo que no estás de acuerdo, y estaré encantado de actualizar el artículo con tus comentarios.

Recuerda que tienes acceso al código completo en mi Github (las estrellas siempre se agradecen 😁)

Clean architecture para Android con Kotlin: una visión pragmática para iniciarse

Clean architecture es un tema que nunca pasa de moda en el mundo Android, y a partir de los comentarios y preguntas que recibo, me da la sensación de que todavía no está muy claro.

Sé que hay decenas (o probablemente cientos) de artículos relacionados con clean architecture pero aquí he querido dar **un enfoque más pragmático / simplista** que puede ayudar en tu primera incursión a la clean architecture. Es por eso que voy omitir conceptos que pueden parecer ineludibles para los puristas de la arquitectura.

Mi principal objetivo aquí es que entiendas lo que considero el punto principal (y más complicado) en clean architecture: **la inversión de dependencias**. Una vez que comprendas esó, puedes ir a otros artículos para rellenar los pequeños huecos que puedan haber quedado fuera.

Clean architecture: ¿por qué debería importarme?

Incluso si decides no utilizar arquitecturas en tus aplicaciones, creo que aprenderlas es realmente interesante, ya que te

ayudará a entender conceptos muy importantes de la programación orientada a objetos.

Las arquitecturas permmiten desacoplar diferentes unidades de tu código de manera organizada. De esta forma el código se hace más fácil de entender, modificar y testear.

Pero las arquitecturas complejas, como la clean architecture pura, también **pueden tener el efecto contrario**: desacoplar el código también significa crear un montón de fronteras, modelos, transformaciones de datos... que pueden terminar aumentando la curva de aprendizaje de tu código hasta un punto en el que no merezca la pena.

Así que, como se debe hacer con todo lo que se aprende, pruébalo en el mundo real y decide qué nivel de complejidad estás dispuesto a introducir. Dependerá del equipo, el tamaño de la aplicación, el tipo de problemas que resuelve...

¡Así que vamos a empezar! En primer lugar, vamos a definir las capas que va a usar nuestra aplicación.

Las capas de clean architecture

Este punto se puede enfocar de maneras muy diferentes. Sin embargo, por simplicidad, me voy a limitar a 5 capas (que ya es lo suficientemente complejo de todas formas 😂):

1. Presentación

Es la capa que interactúa con la interfaz de usuario. Es probable que veas esta capa dividida en dos en otros ejemplos, ya que

técnicamente se podría extraer todo menos las clases de la arquitectura a otra capa. Pero en la práctica, casi nunca se le saca partido, y complica las cosas.

Esta capa de presentación por lo general consiste en la interfaz de usuario de Android (activities, fragments, views) y presenters o view models, según el patrón de presentación que decidas utilizar. Si usa MVP, tengo un artículo donde explico en profundidad (y me gustaría escribir uno sobre MVVM en breve).

2. Casos de uso

A veces también se les llama *interactors*. Se trata principalmente de las **acciones que el usuario puede desencadenar.** Estos pueden ser acciones activas (el usuario hace clic en un botón) o acciones implícitas (la App navega a una pantalla).

Si quieres ser extra-pragmático, puedes incluso quitarte esta capa. A mí gusta porque por lo general es el punto en el que me cambio de hilo. A partir de este punto, puedo ejecutar todo lo demás en un un hilo secundario y olvidarme de tener cuidado con el hilo de UI. De esta forma, no necesito preguntarme más si algo se está ejecutando en el hilo de interfaz de usuario o un hilo de background.

3. Dominio

También conocida como la lógica de negocio. **Estas son las reglas de tu negocio**.

Contiene todos los modelos de negocio. Por ejemplo, en una App de películas, podría ser la clase `Movie`, la clase de `Subtitles`, etc.

Idealmente, debería ser la capa más grande, aunque es cierto que Apps Android por lo general lo único que hacen es dibujar una API en la pantalla de un teléfono, por lo que la mayor parte de la lógica va a consistir simplemente en la solicitud y la persistencia de datos.

4. Datos

En esta capa se encuentra **una definición abstracta de las diferentes fuentes de datos**, y la forma en que se debe utilizar. Aquí se suele usar un patrón repositorio que, para una determinada solicitud, es capaz de decidir dónde encontrar la información.

Una App típica podría guardar sus datos localmente y recuperarlos desde la red. Así que esta capa puede comprobar si los datos están en una base de datos local. Si están ahí y no están caducadon, devolverlos como resultado, y de lo contrario pedirlos a la API y guardarlos localmente.

Pero los datos no tienen que provenir solamente de una petición a un API. Es posible, por ejemplo, utilizar los datos de los sensores del dispositivo, o de un BroadcastReceiver (¡aunque la capa de datos nunca debe saber acerca de este concepto! Lo veremos más adelante)

5. Framework

Puedes encontrar esta capa con muchos nombres distintos. Básicamente encapsula **la interacción con el framework**, por lo que el resto del código puede ser agnóstico y reutilizable en caso de que quieras desarrollar la misma aplicación para otra plataforma (una opción real hoy en día con los proyectos multi-plataforma en Kotlin). Con «framework» no sólo me refiero al framework de Android, sino a cualquier biblioteca externa que queremos ser capaces de reemplazar fácilmente en el futuro.

Por ejemplo, si la capa de datos debe persistir algo, aquí se podría utilizar Room para hacerlo. O si tienes que hacer una petición, se usaría Retrofit. O se puede acceder a los sensores para solicitar alguna información. ¡Lo que sea que necesites!

Esta capa debe ser tan simple como sea posible, ya que toda la lógica debería ser abstraída en la capa de datos.

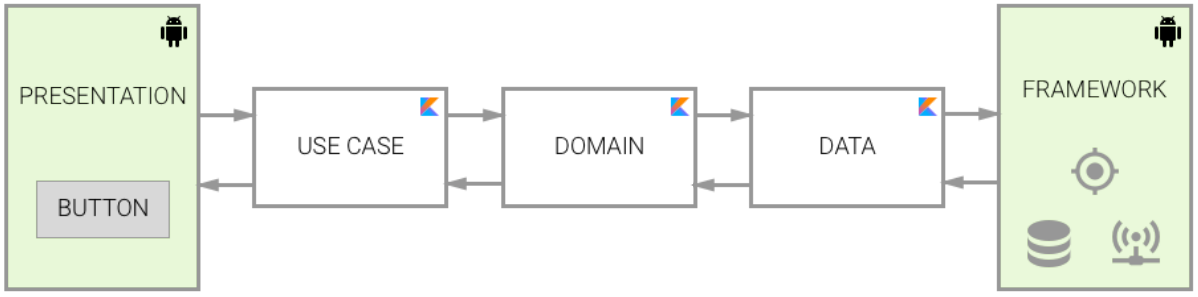
¡Recuerda! Estas son las capas sugeridas, pero algunas de ellas se pueden combinar. Incluso se puede simplemente tener tres capas: presentación – dominio – framework. Esto probablemente no puede llamarse estrictamente clean architecture, pero sinceramente me dan un poco igual los nombres. Dejaré 5 capas, ya que ayuda a explicar el punto siguiente, que creo que es el importante.

Interacción entre capas

Esta es la parte más difícil de explicar y entender. Voy a tratar de ser lo más claro posible, porque creo que este es también el punto más importante si se quiere entender la clean architecture.

Pero no dudes en escribirme si no entiendes algo, y actualizaré el artículo para aclararlo.

Cuando se piensa en una forma lógica de interacción, se diría que la presentación utiliza la capa de casos de uso, que a su vez va a utilizar el dominio para acceder a la capa de datos, la que finalmente va a utilizar el framework para obtener acceso a los datos solicitados. A continuación, estos datos van de vuelta por la estructura de capas hasta llegar a la capa de presentación, que actualiza la interfaz de usuario. Esto sería un gráfico sencillo de lo que está sucediendo:

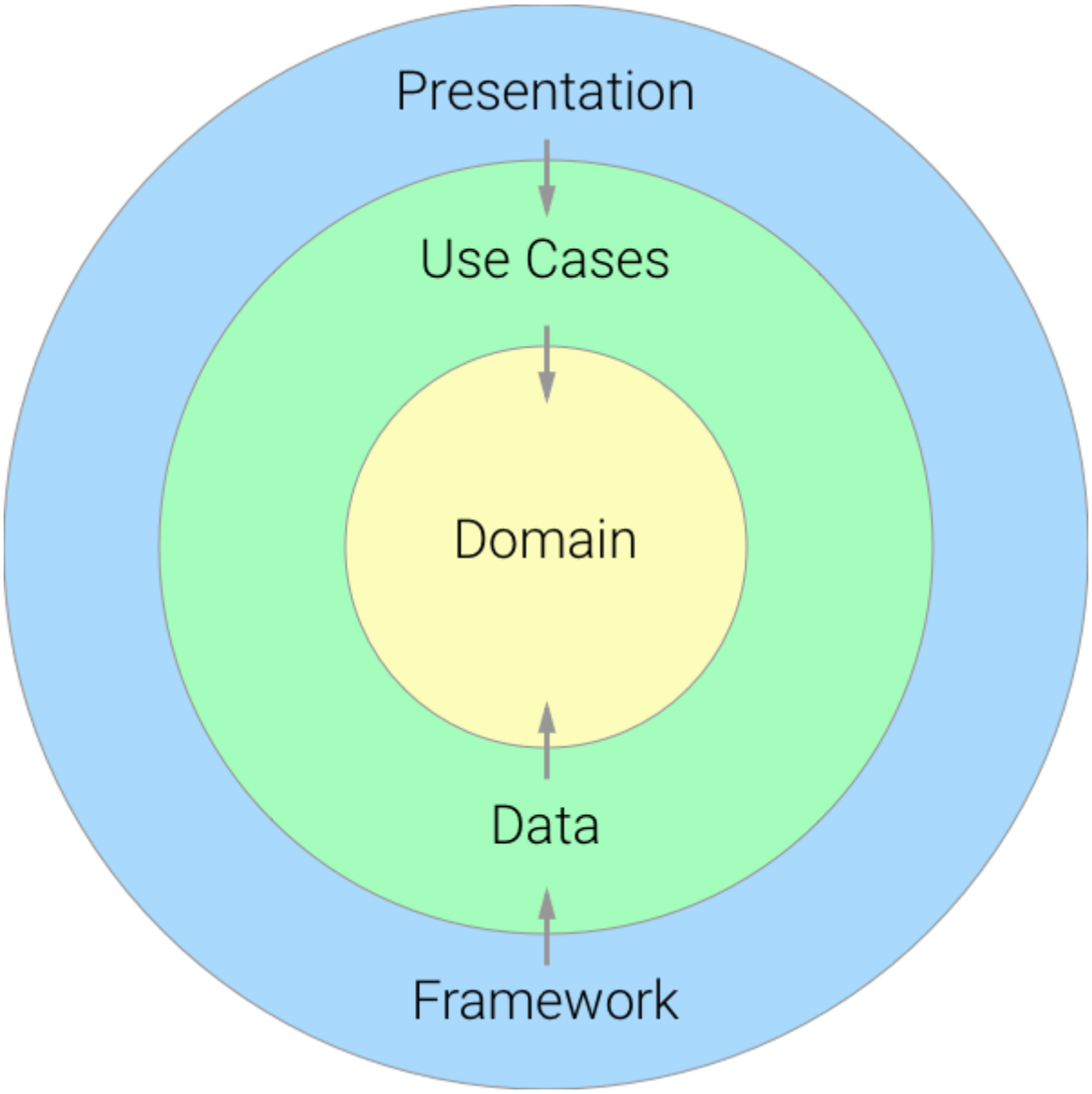


Como puedes ver, los dos límites del flujo dependen del framework, por lo que requieren el uso de la dependencia de Android, mientras que el resto de las capas sólo requieren Kotlin. Esto es realmente interesante si desea dividir cada capa en un sub-módulo por separado. **Si fueses a reutilizar este mismo código para (digamos) una aplicación web, simplemente necesitarías reimplementar las capas de presentación y framework.**

Pero no mezcles el flujo de la aplicación con la dirección de las dependencias entre las capas. Si has leído acerca de la clean architecture antes, es probable que vieras este gráfico:

Que es un poco diferente de la imagen anterior. Los nombres también son diferentes, pero vemos esto en un minuto. Básicamente, la clean architecture dice que tenemos capas exteriores e interiores, y que las capas internas no deben saber nada sobre las externas. Esto significa que una clase externa puede tener una dependencia explícita de una clase interna, pero no al revés.

Vamos a recrear el gráfico anterior con nuestras propias capas:



Así que desde la interfaz de usuario hacia la de dominio, todo es bastante simple, ¿verdad? La capa de presentación tiene una dependencia de casos de uso, y es capaz de llamar para iniciar el flujo. A continuación, el caso de uso tiene una dependencia al dominio.

Sin embargo, los problemas aparecen cuando vamos desde el interior hacia el exterior. Por ejemplo, cuando la capa de datos necesita algo del framework. Como se trata de una capa interna,

la capa de datos no sabe nada acerca de las capas externas, por lo que ¿cómo puede comunicarse con ellos?

Presta atención, que aquí viene lo importante.

Principio de Inversión de Dependencia

Si has oído hablar de los principios SOLID, puede que hayas leído acerca del principio de inversión de dependencias. Pero, como pasa con muchos de estos conceptos, es posible que no que no te quedara claro cómo aplicarlo a un ejemplo real. La inversión de la dependencia es la «D» de los principios SOLID, y esto es lo que dice:

A. Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.

B. Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.

Sinceramente esto a mí no me dice mucho mucho en mi trabajo del día a día. Sin embargo, con este ejemplo, te va a ser más fácil entenderlo:

Un ejemplo de inversión de dependencia

Digamos que tenemos un `DataRepository` en la capa de datos que requiere una `RoomDatabase` para recuperar unos elementos guardados en un base de datos. El primer enfoque sería tener algo como esto.

Este es nuestro `RoomDatabase`:

```
1. class RoomDatabase {
2.     fun requestItems(): List<Item> { ... }
3. }
```

Y el `DataRepository` utilizaría una instancia de la misma:

```
1. class DataRepository {
2.
3.     private val roomDatabase = RoomDatabase()
4.
5.     fun requestItems(): List<Item> {
6.         val items = roomDatabase.requestItems()
7.         ...
8.         return result
9.     }
10.    }
```

¡Pero esto no es posible! **La capa de datos no sabe nada de las clases en Framework porque es una capa más interna.**

Así que el primer paso es hacer una **inversión de control** (no mezclar con la inversión de dependencias, no son lo mismo), lo que significa que en lugar de crear instancias de la clase nosotros mismos, dejamos nos vengan dadas del exterior (a través del constructor):

```
1. class DataRepository(private val roomDatabase:
   RoomDatabase) {
2.     fun requestItems(): List<Item> {
3.         val items = roomDatabase.requestItems()
```

```
        ...
        return result
    }
}
```

Fácil ¿verdad? Pero aquí es donde tenemos que hacer la inversión de dependencias. En vez de depender de la implementación específica, vamos a depender de una abstracción (una interfaz). Por lo que el módulo de dominio tendrá la siguiente interfaz:

```
interface DataPersistence {
    fun requestItems(): List<Item>
}
```

Ahora el `DataRepository` puede utilizar la interfaz (que se encuentra en su misma capa):

```
class DataRepository(private val dataPersistence:
DataPersistence) {

    fun requestItems(): List<Item> {
        val items = dataPersistence.requestItems()
        ...
        return result
    }
}
```

Y ya que la capa de datos puede utilizar la capa de dominio, puede implementar esa interfaz:

```
class RoomDatabase : DataPersistence {
```

```
2.
3.     override fun requestItems(): List<Item> { ... }
4.
5. }
```

El único punto que faltaría es buscar una forma de proporcionar la dependencia al `DataRepository`. **Eso se hace mediante inyección de dependencias.** Las capas externas se harán cargo de ella.

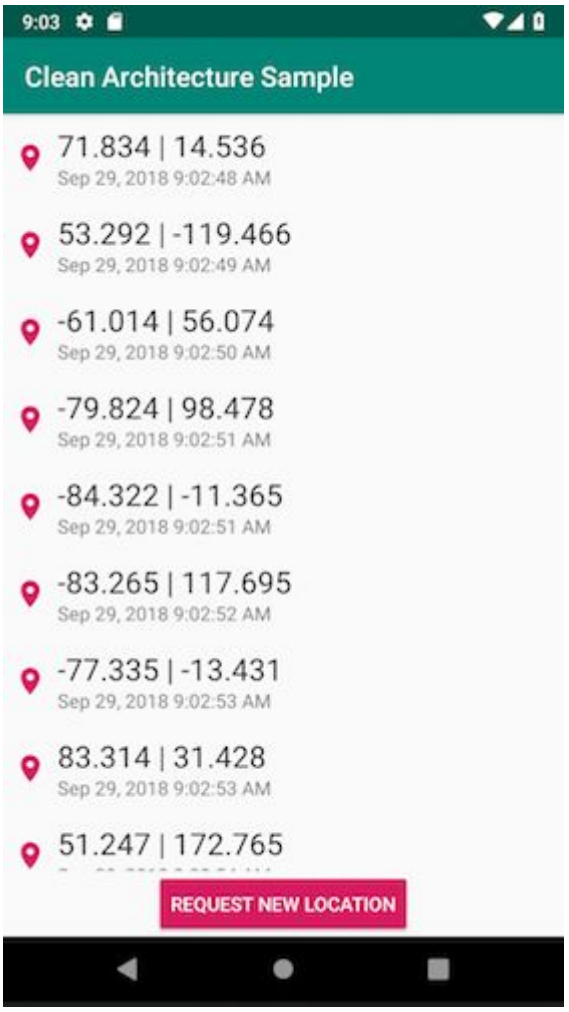
No voy a profundizar en la inyección de dependencia en este artículo, ya que no quiero añadir muchos conceptos complejos. tengo un unos cuantos artículos que hablan sobre ello y sobre Dagger, por si quieres ampliar sobre el tema.

Implementando un ejemplo

Lo primero de todo es que puedes encontrar el ejemplo completo en este repositorio. Voy a omitir algunos detalles, así que puedes ir allí a echarle un ojo, hacer *fork* y jugar un poco con el ejemplo.

Todo esto está muy bien, pero hay que ponerlo en práctica si queremos entenderlo por completo.

Para ello, vamos a crear una App que permitirá solicitar la ubicación actual gracias a un botón y mantener un registro de las ubicaciones previamente solicitadas, mostrándolas en una lista.



Creación de un proyecto de ejemplo

El proyecto consistirá en un conjunto de 5 módulos.

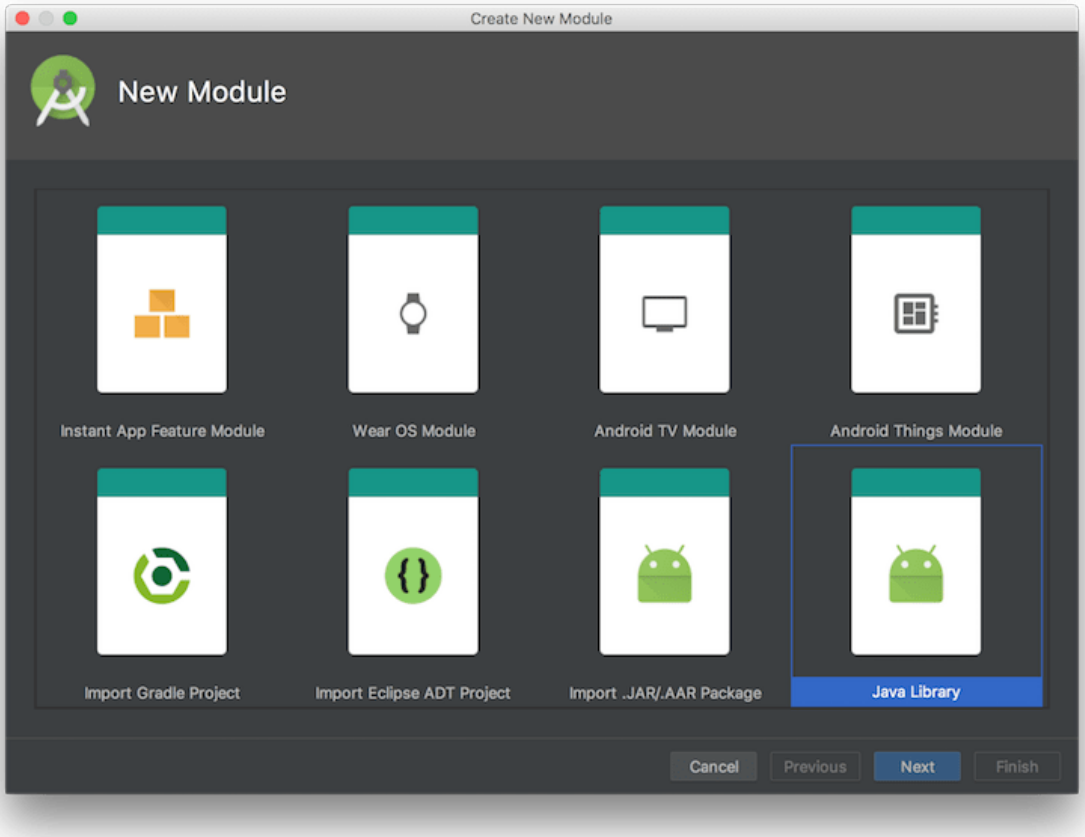
Por simplicidad, sólo voy a crear 4:

- **app**: Será el único proyecto que utiliza el framework de Android, que incluirá las capas de presentación y de Framework.
- **usecases**: Será un módulo Kotlin (no necesita el framework de Android).

- **dominio**: Otro módulo Kotlin.
- **datos**: Un módulo Kotlin también.

Se podría hacer perfectamente todo en un único módulo, y utilizar paquetes en su lugar. Pero es más fácil de no violar el flujo de dependencias si se hace así. Así que recomiendo te lo recomiendo si te estás iniciando.

Crea un nuevo proyecto, que creará automáticamente el módulo de aplicación, y luego crea los módulos adicionales de Java:



No hay una opción «Kotlin library», por lo que tendrás que añadir el soporte para Kotlin después.

La capa de dominio

Necesitamos una clase que representa la ubicación. En Android, ya tenemos una clase Location. Pero recordemos que queremos dejar los detalles de implementación en las capas exteriores.

Imagina que quieres utilizar este código para una aplicación web escrita en KotlinJS. Aquí no tendrías acceso a las clases de Android.

Así que esta es la clase:

```
1. data class Location(val latitude: Double, val longitude:
   Double, val date: Date)
```

Si has leído sobre esto antes, una clean architecture pura tendría una representación del modelo por capa, que en nuestro caso implicaría tener una clase *Location* en cada capa. A continuación, se usarían transformaciones de datos para convertirlos al pasar de una capa a otra. Eso hace que las capas estén menos acopladas, pero también todo lo más complejo. En este ejemplo, sólo voy a hacerlo cuando sea estrictamente necesario.

Esto es todo lo que necesitas en esta capa para este ejemplo sencillo.

La capa de datos

La capa de datos normalmente se modela usando repositorios que acceden a los datos que neceistamos. Podemos tener algo como esto:

```
1. class LocationsRepository {
2.
```

```
3.     fun getSavedLocations(): List<Location> { ... }
4.     fun requestNewLocation(): List<Location> { ... }
5.
6. }
```

Tiene una función para obtener las ubicaciones pedidas anteriormente, y otra función para solicitar una nueva.

Como se puede ver, esta capa está utilizando la capa de dominio. Una capa exterior puede utilizar las capas internas (pero no a la inversa). Para ello, es necesario agregar una nueva dependencia al `build.gradle`` del módulo:

```
1. dependencies {
2.     implementation project(':domain')
3.     ...
4. }
```

El repositorio se va a utilizar un par de orígenes (o sources):

```
1. class LocationsRepository(
2.     private val locationPersistenceSource:
   LocationPersistenceSource,
3.     private val deviceLocationSource:
   DeviceLocationSource
4. )
```

Uno de ellos tiene acceso a las ubicaciones almacenadas, y el otro a la ubicación actual del dispositivo.

Y aquí es donde sucede la magia inversión de dependencias. Estas dos fuentes son interfaces:

```
1. interface LocationPersistenceSource {
2.
3.     fun getPersistedLocations(): List<Location>
4.     fun saveNewLocation(location: Location)
5.
6. }
7.
8. interface DeviceLocationSource {
9.
10.       fun getDeviceLocation(): Location
11.
12.    }
```

Y la capa de datos no sabe (y no necesita saber) cuál es la implementación real de estas interfaces. Las ubicaciones almacenadas y del dispositivo deben ser gestionadas por el framework específico del dispositivo. Una vez más, volviendo al ejemplo de KotlinJS, una aplicación web implementaría esto de forma muy diferente a la de una App para Android.

Ahora, el `LocationsRepository` puede utilizar estas fuentes sin necesidad de conocer la implementación final:

```
1. fun getSavedLocations(): List<Location> =
   locationPersistenceSource.getPersistedLocations()
2.
3. fun requestNewLocation(): List<Location> {
4.     val newLocation =
   deviceLocationSource.getDeviceLocation()
5.     locationPersistenceSource.saveNewLocation(newLocation)
```

```
6.     return getSavedLocations()
7. }
```

La capa de Casos de Uso

Esta es por lo general una capa muy simple, que simplemente convierte las acciones del usuario en las interacciones con el resto de capas internas. En nuestro caso, vamos a tener un par de casos de uso:

- `GetLocations`: Devuelve las ubicaciones que ya han sido registradas por la aplicación.
- `RequestNewLocation`: Le dirá al `LocationsRepository` que pida la ubicación actual.

Estos casos de uso tendrán una dependencia al `LocationRepository`:

```
1. class GetLocations(private val locationsRepository:
   LocationsRepository) {
2.
3.     operator fun invoke(): List<Location> =
   locationsRepository.getSavedLocations()
4.
5. }
```

```
1. class RequestNewLocation(private val
   locationsRepository: LocationsRepository) {
2.
3.     operator fun invoke(): List<Location> =
   locationsRepository.requestNewLocation()
4.
```

```
5. }
```

La capa del Framework

Esta va a ser parte del módulo de `app`, e implementará principalmente las dependencias que se ofrecen al resto de capas. En nuestro caso particular, será `LocationPersistenceSource` y `DeviceLocationSource`.

El primero podría ser implementado con *Room*, por ejemplo, y el segundo con el `LocationManager`. Pero con la intención de hacer esta explicación más simple, voy a utilizar implementaciones fake. Podría implementar la solución real en algún momento, pero esto sólo añadiría complejidad a la explicación, por lo que prefiero que nos olvidemos de ello por ahora.

Para la persistencia, voy a usar una implementación en memoria sencilla:

```
1. class InMemoryLocationPersistenceSource :
   LocationPersistenceSource {
2.
3.     private var locations: List<Location> = emptyList()
4.
5.     override fun getPersistedLocations(): List<Location>
   = locations
6.
7.     override fun saveNewLocation(location: Location) {
8.         locations += location
9.     }
```

```
10.    }
```

Y un generador aleatorio para el otro:

```
1. class FakeLocationSource : DeviceLocationSource {
2.
3.     private val random =
   Random(System.currentTimeMillis())
4.
5.     override fun getDeviceLocation(): Location =
6.         Location(random.nextDouble() * 180 - 90,
7.             random.nextDouble() * 360 - 180, Date())
8. }
```

Piensa en esto en un proyecto real. Gracias a las interfaces, **durante la implementación de una nueva funcionalidad, se pueden proporcionar dependencias fake** mientras se trabaja en el resto del flujo, y olvidarse de los detalles de implementación hasta el final.

Esto también demuestra que **estos detalles de implementación son fácilmente intercambiables**. Así podrías hacer que tu App trabaje inicialmente con persistencia en memoria, y luego más adelante usar una base de datos. Se pueden implementar como queramos, y luego reemplazarlos.

O imagina que una nueva biblioteca aparece (como Room 😂) y quieres probarlo y considerar una posible migración. Sólo tienes que implementar la interfaz utilizando la nueva librería, sustituir la dependencia, ¡y ya lo tienes funcionando!

Y, por supuesto, **esto también ayuda en los tests, en el que podemos sustituir estos componentes por fakes o mocks.**

La capa de presentación

Y ahora podemos implementar la interfaz de usuario. Para este ejemplo, voy a usar MVP, porque este artículo se originó por las preguntas en el artículo original sobre MVP y porque creo que es más fácil de entender que el uso de MVVM con architecture components. Sin embargo, ambos enfoques son muy similares.

En primer lugar, tenemos que escribir el Presenter, el cual recibirá una vista como dependencia (la interfaz del Presenter para interactuar con su vista) y los dos casos de uso:

```
1. class MainPresenter(
2.     private var view: View?,
3.     private val getLocations: GetLocations,
4.     private val requestNewLocation: RequestNewLocation
5. ) {
6.     interface View {
7.         fun renderLocations(locations: List<Location>)
8.     }
9.
10.      fun onCreate() = launch(UI) {
11.          val locations = bg { getLocations() }.await()
12.          view?.renderLocations(locations)
13.      }
14.
15.      fun newLocationClicked() = launch(UI) {
16.          val locations = bg { requestNewLocation() }
17.          view?.renderLocations(locations.await())
18.      }
```

```
19.
20.      fun onDestroy() {
21.          view = null
22.      }
23.  }
```

Todo muy sencillo, dejando a un lado la forma de realizar tareas en segundo plano. He usado corrutinas (de hecho, la librería Anko, para utilizar `bg`). No estoy seguro de si es la mejor decisión, porque quería mantener este ejemplo tan fácil como sea posible. Así que si no se entiendo bien, indícamelo en los comentarios. Puedes saber más sobre las corrutinas en este artículo que escribí hace algún tiempo.

Por último, el `MainActivity`. Con el fin de evitar el uso de un inyector de dependencias, he declarado aquí las dependencias:

```
1. private val presenter: MainPresenter
2.
3. init {
4.   // This would be done by a dependency injector in a
   complex App
5.   //
6.    val persistence = InMemoryLocationPersistenceSource()
7.    val deviceLocation = FakeLocationSource()
8.    val locationsRepository =
   LocationsRepository(persistence, deviceLocation)
9.    presenter = MainPresenter(
10.          this,
11.          GetLocations(locationsRepository),
12.          RequestNewLocation(locationsRepository)
13.      )
```

```
14.    }
```

No recomiendo usar esto para una App grande, ya que se podrían reemplazar las dependencias en los tests de UI, por ejemplo, pero es suficiente para este ejemplo.

Y el resto del código no necesita mucha explicación. Tenemos un `RecyclerView` y un botón. Cuando se hace clic en el botón, se llama al Presenter para que se solicite una nueva ubicación:

```
1. newLocationBtn.setOnClickListener {
   presenter.newLocationClicked() }
```

Y cuando el Presenter termina, se llama al método de la `View`. Esta interfaz la implementa la Activity de esta manera:

```
1. override fun renderLocations(locations: List<Location>)
   {
2.     locationsAdapter.items = locations
3. }
```

Conclusión

¡Esto es todo! Los fundamentos de la clean architecture son de hecho bastante simples.

Sólo es necesario entender **cómo funciona la inversión de dependencias, y luego enlazar las capas correctamente**. Sólo recuerda no agregar dependencias de los módulos externos a los

internos, y tendrás mucha ayuda del IDE para hacer las cosas bien.

Sé que esto es una mucha información de golpe. Mi objetivo es que esto sirve como un punto de entrada para las personas que nunca vieron la clean architecture antes. Así que todavía tienes dudas, házmelo saber en los comentarios y reescribiré este artículo tantas veces como sea necesario.

Y también recuerda que con el fin de hacer este simple artículo, he omitido algunas complejidades que encontrarías en una clean architecture habitual. Una vez que tengas esto claro, te sugiero que leas otros ejemplos más completos. Siempre me gusta recomendar éste de Fernando Cejas.

El enlace al repositorio de Github está aquí. Puedes ir allí para echar un ojo a los pequeños detalles, y si te gusta, por favor, házmelo saber con una estrella 🙂

¡Espero que te haya gustado!

Siempre puedes enviarme tu feedback a contacto@devexperto.com y estaré encantado de leerlo.

Recuerda que si estás buscando amplicar tus conocimientos, tengo las siguientes opciones:

- El libro **Kotlin for Android Developers**,
- El curso online
- La formación para empresas.
- El programa Architect Coders (¡incluye un training gratuito de 2 semanas!)

Muchas gracias, y seguimos en contacto por:

- Instagram
- YouTube
- Canal de Telegram
- Twitter
- Facebook
- Podcast

Y si has llegado a esta página por otra vía, puedes suscribirte aquí y llevarte de regalo la guía para iniciarte en Kotlin.

¡Un abrazo!
Antonio Leiva