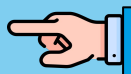
Arquitectura en Android

Componentes de arquitectura (Parte I)

{desafío} latam_

Utilizar patrones de arquitectura escalables para la construcción de una aplicación Android de acuerdo a los requerimientos

- Unidad 1: Kotlin para el desarrollo de aplicaciones.
- Unidad 2: Ciclo de vida de componentes.
- Unidad 3: Arquitectura en Android.
- Unidad 4: Programación asíncrona en Android.



Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Implementar un aplicativo utilizando el patrón MVVM de acuerdo a las buenas prácticas recomendadas*

¿Qué te ha parecido el ViewModel hasta ahora?

¿Viste alguno de los ejemplos sugeridos en la sesión anterior?

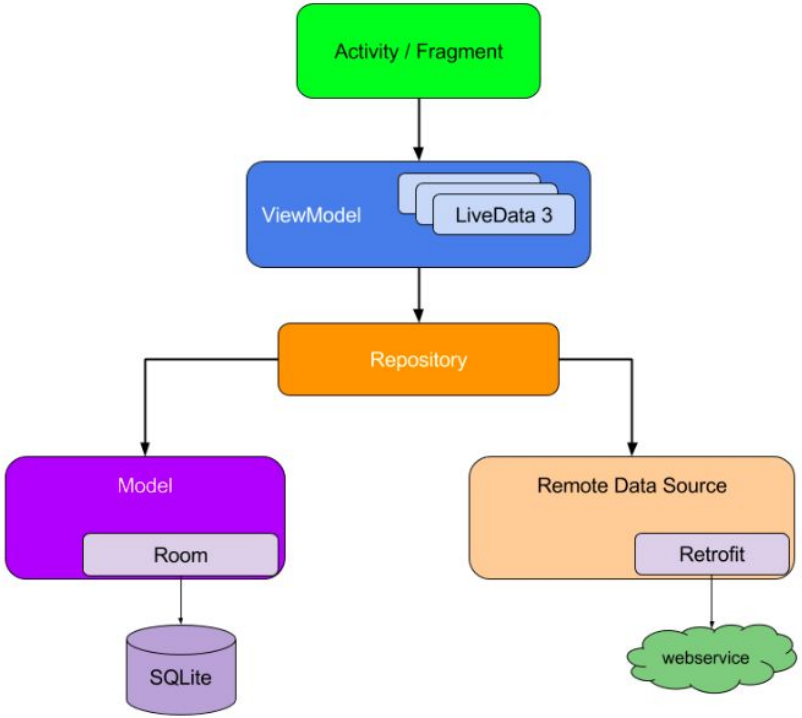
/* ViewModel */

Ejercicio - “ViewModel”

Ejercicio ViewModel

Veamos otro ejercicio de cómo usar el ViewModel, en el ejemplo anterior vimos cómo compartir datos entre Fragments, pero qué pasa si lo que necesitamos es recibir datos desde una API o desde una base de datos local.

Primero debemos saber que existe otro patrón de diseño llamado "**Repository**", el cual veremos en más detalle más adelante, por ahora solo debemos entender que funciona como SSOT (Single Source Of Truth o Fuente Única de Verdad)



Ejercicio ViewModel

Para el caso de nuestro, vamos a simular que recibimos datos desde una fuente externa, es decir, desde Internet (una API por ejemplo) o desde la base de datos local, y queremos mostrar esos datos en un Fragment.

Descarga el **Archivo complementario - Componentes de arquitectura (Parte I)** el cual contiene el proyecto iniciado.

Lo primero que debemos hacer es crear una instancia del ViewModel en `MainActivity()`, siguiendo lo aprendido anteriormente, lo haremos usando delegación de principios, para esto debemos revisar primero si `build.gradle` está configurado para esto.

Busca la siguiente línea en build.gradle

```
implementation "androidx.fragment:fragment-ktx:1.5.3"
```

Ejercicio ViewModel

Ahora que podemos crear el ViewModel, el cual estará vacío por ahora.

- Crea un nuevo paquete y nómbralo “viewmodel”, dentro del paquete, genera una nueva clase que extienda de ViewModel() y nómbrala MainViewModel.

```
class MainViewModel : ViewModel() {}
```

- Luego vamos a crear una instancia del ViewModel en el MainActivity, lo haremos de la siguiente forma.

```
private val viewModel: MainViewModel by viewModels()
```

Ejercicio ViewModel

Necesitamos llamar a nuestra clase `Repository` desde el ViewModel

- Creamos una instancia del Repository:

```
private val repository = StoreRepository()
```

- Ahora debemos poder consumir el repository, para esto creamos una función que retorne una lista de Store.

```
fun getStoreList(): List<Store> {
    return repository.getStoreList()
}
```

O más simple:

```
fun getStoreList(): List<Store> = repository.getStoreList()
```

{desafío} latam_

IMPORTANTE: Para el caso de este ejercicio vamos a ignorar el hecho de que Android no permite hacer llamados a API o bases de datos directamente desde el Main Thread.

Ejercicio ViewModel

- Casi hemos terminado, ahora lo que necesitamos es llamar la función que creamos en el ViewModel desde MainActivity, lo hacemos de la siguiente forma:

```
viewModel.getStoreList()
```

- Podemos ver el listado de tiendas usando Log,

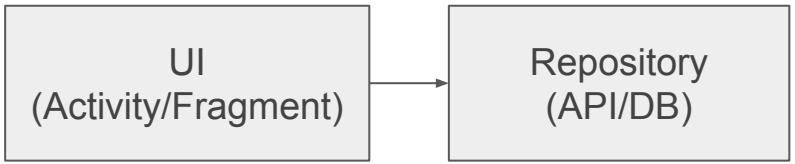
```
Log.d("TAG", "getStoreList: ${viewModel.getStoreList()}")
```

¿Qué podemos hacer ahora?, el objetivo del ejemplo es solamente mostrar cómo se puede llamar desde un activity, una función que está definida más allá del ViewModel.

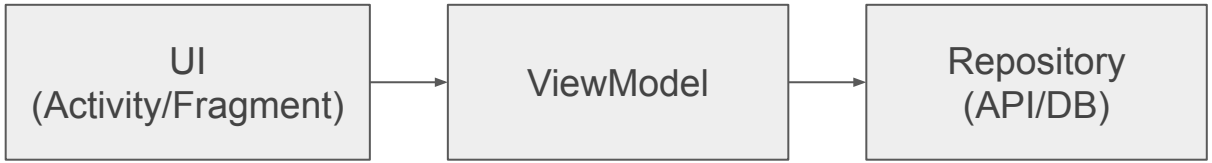
Ejercicio ViewModel

Para ahondar un poco más en el tema anterior a la implementación del ViewModel, estábamos llamando las funciones directamente desde la fuente, lo cual no es recomendable. Si bien la app puede funcionar, se vuelve inmantenible al momento de agregar más fuentes de datos, como por ejemplo: conexión a otras API o una base de datos más grande.

Antes



Ahora



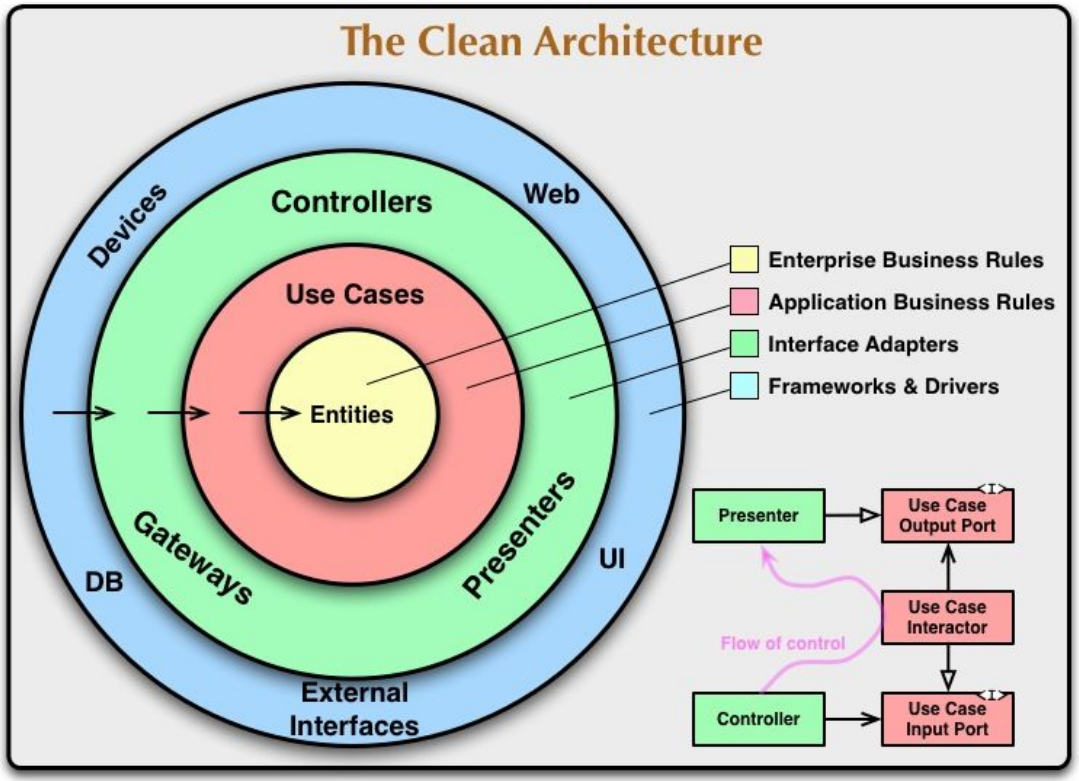
/* Clean Architecture */

¿Qué es Clean Architecture?

Clean Architecture es un método de desarrollo de software en el que se debería poder identificar lo que realiza el software simplemente mirando su código fuente. Al igual que otras filosofías de diseño de software, Clean Architecture tiene como objetivo proporcionar un proceso rentable para desarrollar código de calidad que funcione mejor, sea más fácil de modificar y tenga menos dependencias.

Dato: Robert C. Martin estableció Clean Architecture y lo promocionó en su blog, Uncle Bob, en 2011.

Diagrama oficial de Clean Architecture



Fuente: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Ventajas y Desventajas de Clean Architecture

Ventaja

- El código es más fácil de probar que con MVVM estándar.
- Separación perfectamente curada (la ventaja más considerable).
- Estructura de paquete fácil de usar.
- Fácil de mantener el proyecto en marcha.
- Los desarrolladores podrán implementar nuevas funciones aún más rápido.

Desventajas

- La curva de aprendizaje es un poco empinada. Puede tomar algún tiempo aprender cómo interactúan todos los niveles, especialmente si proviene de arquitecturas como MVVM o MVP simples.
- Contiene muchas clases adicionales, por lo que no es adecuado para aplicaciones con un bajo nivel de sofisticación.

¿Por qué MVVM con Clean Architecture?

MVVM separa su vista (es decir, actividades y fragmentos) de su lógica de negocio. MVVM es suficiente para proyectos pequeños, pero cuando su base de código se vuelve enorme, sus ViewModels comienzan a hincharse. Separar responsabilidades se vuelve difícil.

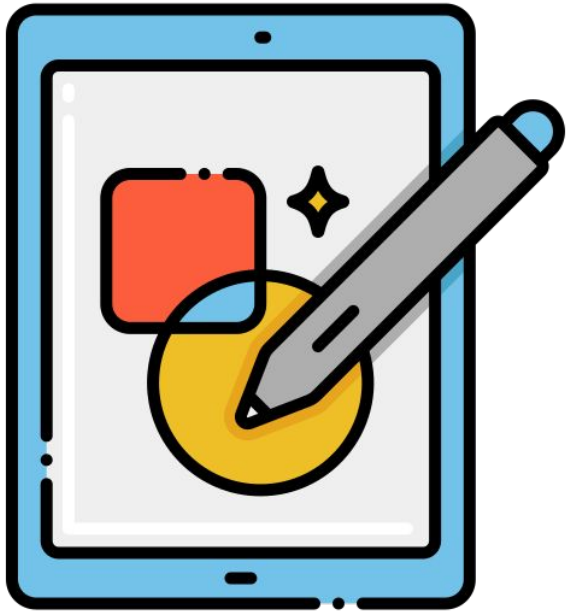
MVVM con Clean Architecture es bastante bueno en tales casos. Va un paso más allá al separar las responsabilidades de su base de código. Abstrae claramente la lógica de las acciones que se pueden realizar en su aplicación.

/* SOLID */

SOLID

En ingeniería de software, **SOLID** es un acrónimo mnemotécnico de cinco principios de diseño destinados a hacer que los diseños orientados a objetos sean más comprensibles, flexibles y mantenibles, los principios son:

- **S**ingle-responsibility
- **O**pen–closed
- **L**iskov substitution
- **I**nterface segregation
- **D**ependency inversion

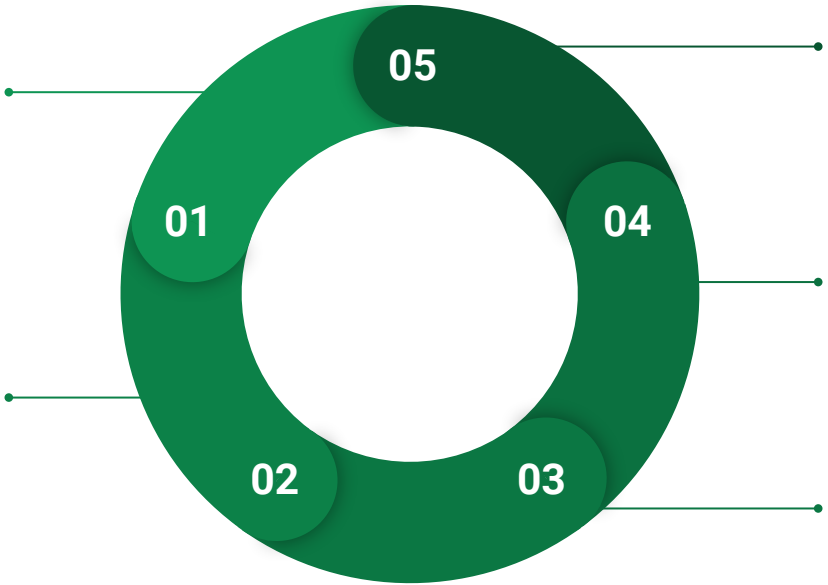


SOLID

Las ideas **SOLID** son:

El principio de responsabilidad única
Nunca debe haber más de una razón para que una clase cambie". En otras palabras, cada clase debe tener una sola responsabilidad.

El principio abierto-cerrado
Las entidades de software... deben estar abiertas para la extensión, pero cerradas para la modificación



El principio de sustitución de Liskov
Las funciones que usan punteros o referencias a clases base deben poder usar objetos de clases derivadas sin saberlo

El principio de segregación de interfaces
No se debe obligar a los clientes a depender de interfaces que no utilizan

El principio de inversión de dependencia
Depende de abstracciones, NO de concreciones

{desafío} latam_

Aunque los principios **SOLID** se aplican a cualquier diseño orientado a objetos, también pueden formar una filosofía central para metodologías como el desarrollo ágil o el desarrollo de software adaptativo.

RESUMEN

- El ViewModel resuelve de muy buena forma muchos problemas relacionados con el ciclo de vida de la aplicación, sin embargo, aun así es posible cometer errores, recuerda tener siempre presente donde se está llamando al ViewModel.
- Clean Architecture NO dice de qué forma debes distribuir los paquetes en una app, pero te puede dar una idea de cómo agruparlos.
- Haz lo posible por seguir los principios de SOLID, es posible que al inicio sea un poco complejo, sin embargo, si lo haces parte de tu rutina diaria como desarrollador te ahorrarás muchas horas de debugging o refactoring.

Próxima sesión...

- *Continuaremos implementando un aplicativo utilizando el patrón MVVM de acuerdo a las buenas prácticas recomendadas.*

{desafío}
latam_

Academia de
talentos digitales