Arquitectura Limpia + MVVM

- Mesias Mariscal
- Brayan Quispe
- Gabriel Murillo

Arquitectura Limpia

Robert C. Martin

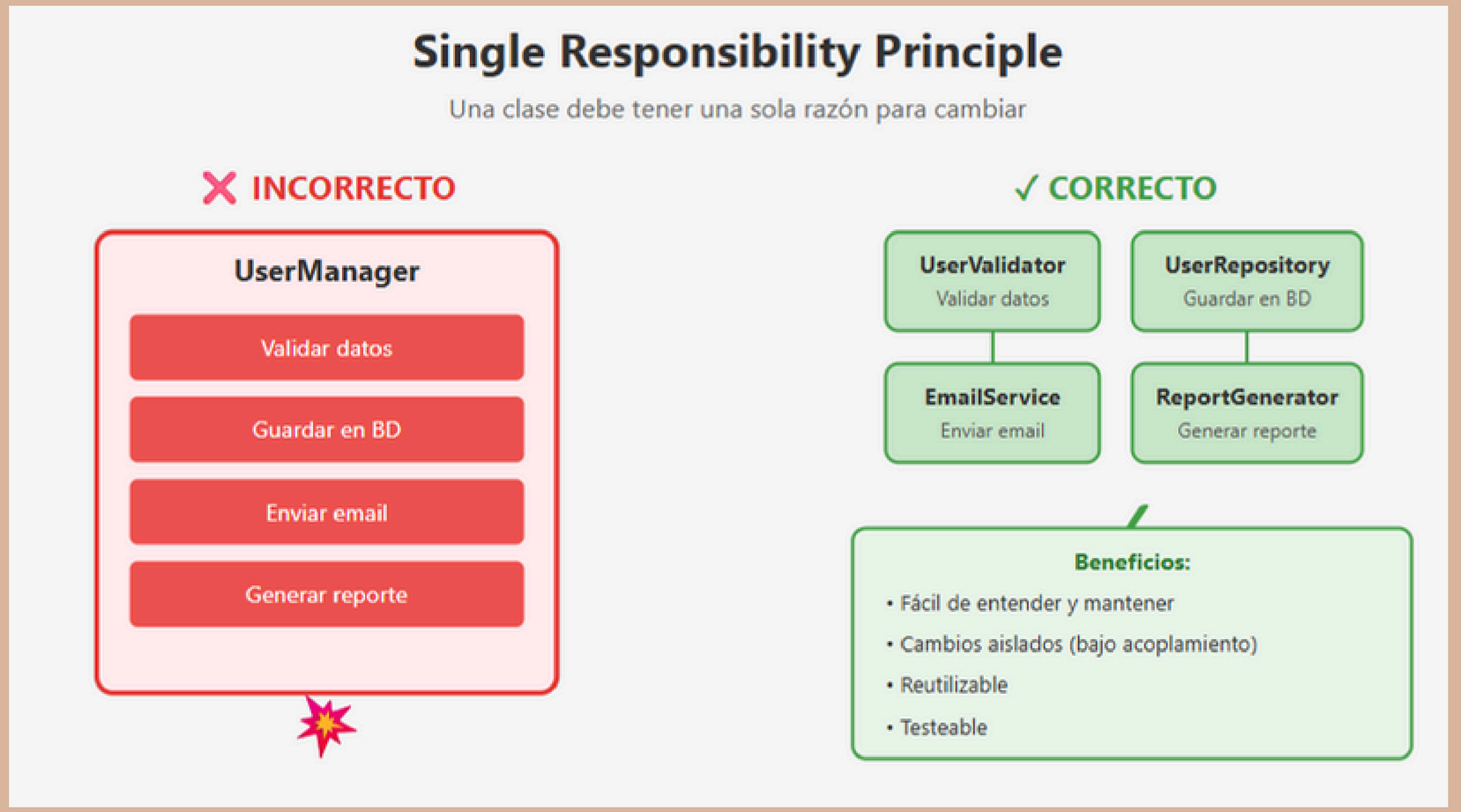
La Arquitectura Limpia, propuesta por Robert C. Martin, no es meramente un patrón de organización de código, sino un enfoque de diseño de sistemas que prioriza la mantenibilidad, escalabilidad y la independencia de las implementaciones técnicas.

Pricipios:

La solidez de la Arquitectura Limpia se cimienta directamente en los Principios de Diseño Orientado a Objetos (OOD), especialmente aquellos recopilados en el acrónimo SOLID, introducidos por Robert C. Martin en su paper de 2000 sobre Principios de Diseño.

El Principio de Responsabilidad Única (SRP)

Las funcionalidades implementadas en una parte del sistema no deben interferir ni concernir a otras partes.

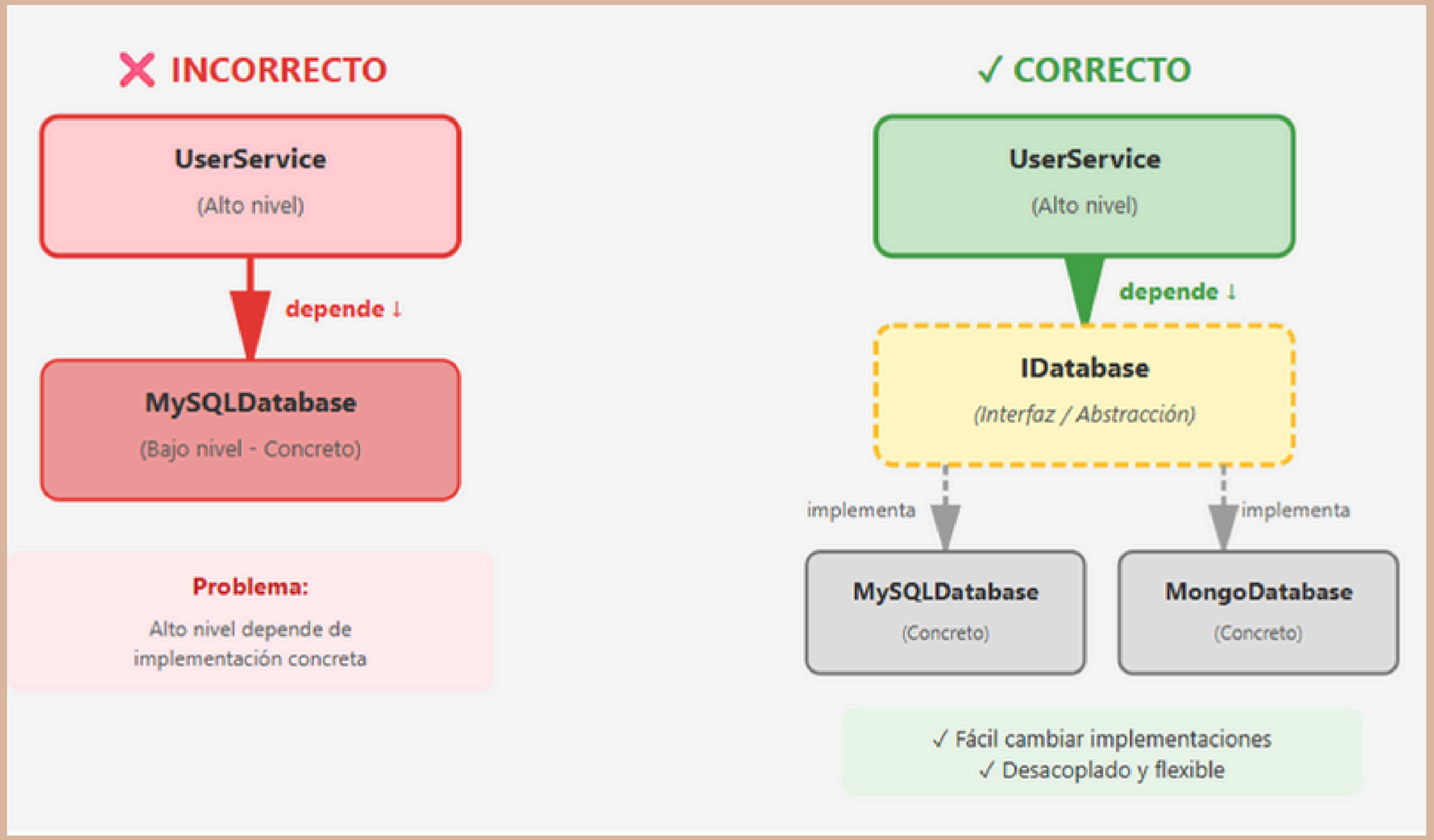


Pricipios:

Principio de Inversión de Dependencias (DIP)

1. Los módulos de alto nivel (la lógica de negocio central) no deben depender de los módulos de bajo nivel (la infraestructura o frameworks).
2. Ambos deben depender de abstracciones (interfaces).

Aislar la lógica que es menos propensa a cambiar es lo que garantiza la Estabilidad y Cambiabilidad del sistema, criterios fundamentales para la mantenibilidad del software según la norma ISO/IEC 25010.



Definición y Rol de las Capas Concéntricas de CA

La arquitectura limpia organiza el código en capas concéntricas, donde la lógica de negocio central reside en el centro, y la infraestructura se encuentra en la periferia. La regla de dependencia establece que el código de las capas interiores no debe tener conocimiento del código en las capas exteriores.

1

Entities (Entidades): Contienen las reglas de negocio más generales y estables de la empresa. Son los objetos centrales del dominio y son las menos probables de cambiar, por lo que residen en el centro, libres de cualquier dependencia externa.

2

Use Cases (Casos de Uso): Contienen las reglas de negocio específicas de la aplicación (application-specific business rules). Los Casos de Uso, también conocidos como Interactors, orquestan el flujo de datos para lograr un objetivo específico.

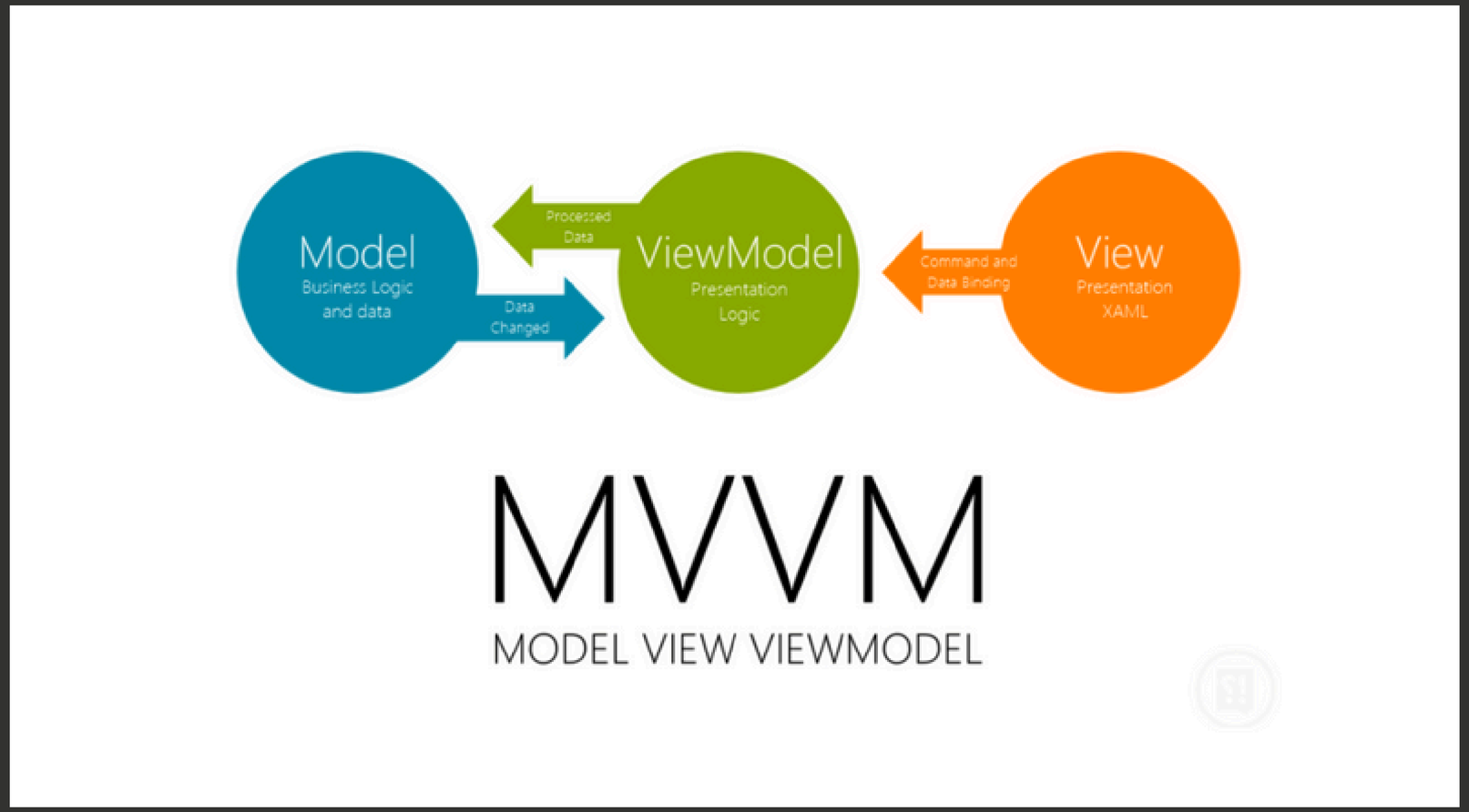
3

I**nterface Adapters (Adaptadores de Interfaz):** Esta capa actúa como un puente, adaptando los datos entre el formato interno (Entidades y Casos de Uso) y el formato requerido por las capas externas, como la UI o la base de datos. Componentes como Controladores, Presenters, Gateways y, crucialmente, los ViewModels, residen aquí.

4

Frameworks & Drivers: La capa más externa. Contiene todos los detalles técnicos y frameworks específicos: la Interfaz de Usuario (UI), los controladores web, la base de datos y cualquier dispositivo o interfaz externa.

Model View ViewMode



MVVM (Model - View - ViewModel)

Es un patrón de diseño de Presentación (UI) que separa la lógica de la interfaz de usuario de la lógica de negocio. Su objetivo es maximizar la separación de preocupaciones y facilitar las pruebas (testing).

Los 3 Componentes Clave

1. Model (Modelo)

Qué es: El "Motor" de la aplicación.

Responsabilidad: Contiene la lógica de negocio pura y el acceso a datos (API, Base de Datos, Repositorios).

Regla: No sabe nada de la UI (ni del ViewModel ni de la Vista).

2. View (Vista)

Qué es: La Interfaz de Usuario (UI) (lo que el usuario ve).

Responsabilidad: Es "tonta". Solo muestra los datos que recibe y reporta las acciones del usuario (clics, texto).

Regla: No contiene lógica de decisión.

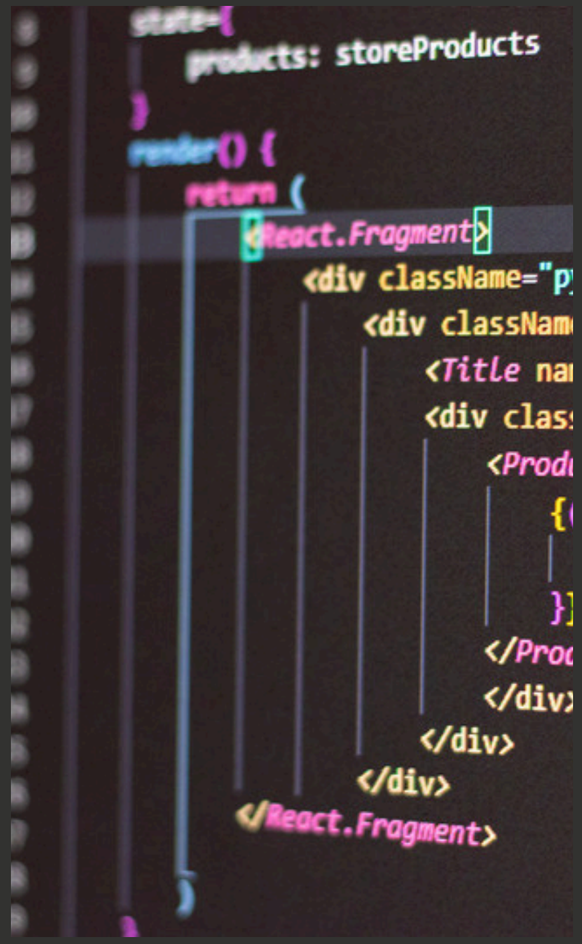
3. ViewModel (Modelo de Vista)

Qué es: El "Cerebro" de la Vista.

Responsabilidad: Contiene el Estado de la UI (ej. isLoading, listaDeDatos, error) y la lógica de presentación (qué hacer con las acciones del usuario).

Regla: No tiene referencia directa a la Vista (no sabe de botones, XML, etc.).

El Flujo de Interacción



Observación (Data Binding)

La Vista observa el Estado expuesto por el ViewModel.

Acción del Usuario

La Vista recibe un clic y notifica al ViewModel (ej. viewModel.botonPresionado()).

Lógica

El ViewModel ejecuta la lógica (p.ej., llamar al Modelo para cargar datos).

Actualización de Estado

El ViewModel actualiza su Estado (ej. isLoading = false, datos = [...]).

Reacción

La Vista, al estar observando, detecta el cambio de estado y se redibuja automáticamente para reflejar los nuevos datos.

Provider y Riverpod:

El Pegamento de MVVM

Provider fue la herramienta estándar para estado en Flutter, pero Riverpod es más moderno, flexible, seguro, fácil de testear y no depende del contexto

Provider y Riverpod son bibliotecas de gestión de estado (los "utensilios") que nos permiten implementar esa receta en Flutter. Resuelven los dos problemas clave de MVVM:

1) Proveer el ViewModel a la View (Inyección de Dependencias)

2) Observar los cambios de estado para que la View se redibuje automáticamente.

1

Provider (El Enfoque Clásico)

Cómo funciona: Depende fuertemente del BuildContext para acceder a los ViewModels (ej. context.watch<MiViewModel>()).

Estructura: Requiere que los "providers" se declaren en la parte superior del árbol de widgets (ej. MultiProvider).

Desventaja: Su dependencia del context puede ser limitante y llevar a errores comunes como ProviderNotFoundException si se usa un context incorrecto.

2

Provider (El Enfoque Clásico)

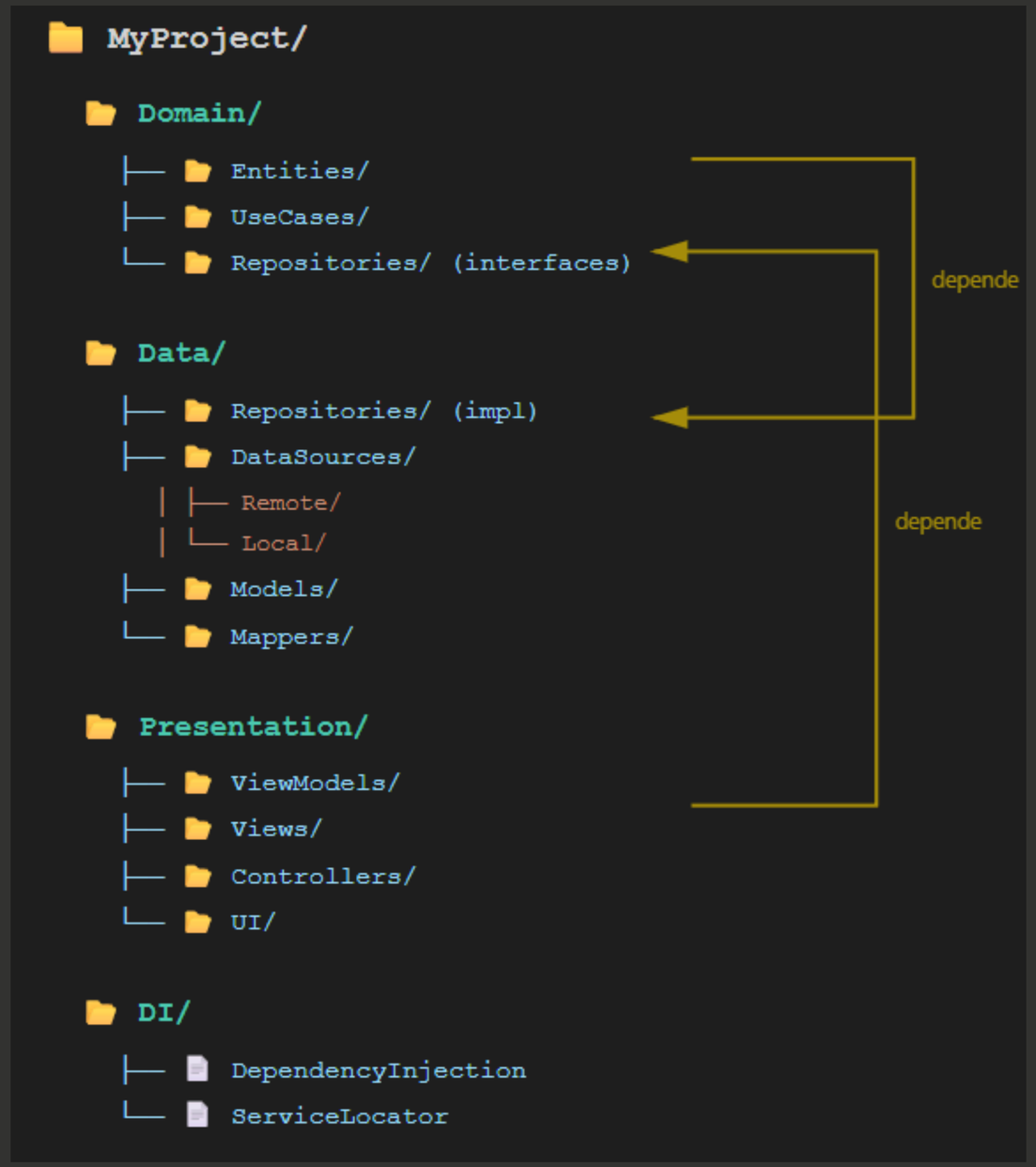
Cómo funciona: Depende fuertemente del BuildContext para acceder a los ViewModels (ej. context.watch<MiViewModel>()).

Estructura: Requiere que los "providers" se declaren en la parte superior del árbol de widgets (ej. MultiProvider).

Desventaja: Su dependencia del context puede ser limitante y llevar a errores comunes como ProviderNotFoundException si se usa un context incorrecto.

Link del repositorio:

Estructura de Archivos



Muchas Gracias