

android

Nathaly Roxana Hidalgo Gómez

William Obed Gutierrez Flores

Kevin Miguel Jiménez Hernández

HG201730

GF190083

JH200303

# Arquitectura CLEAN

Una arquitectura limpia es aquella que pretende lograr unas construcciones modulares bien separadas, de simple lectura, limpieza del código y testabilidad.

Basándonos en el artículo de Uncle Bob, los sistemas construidos con una arquitectura limpia han de ser :

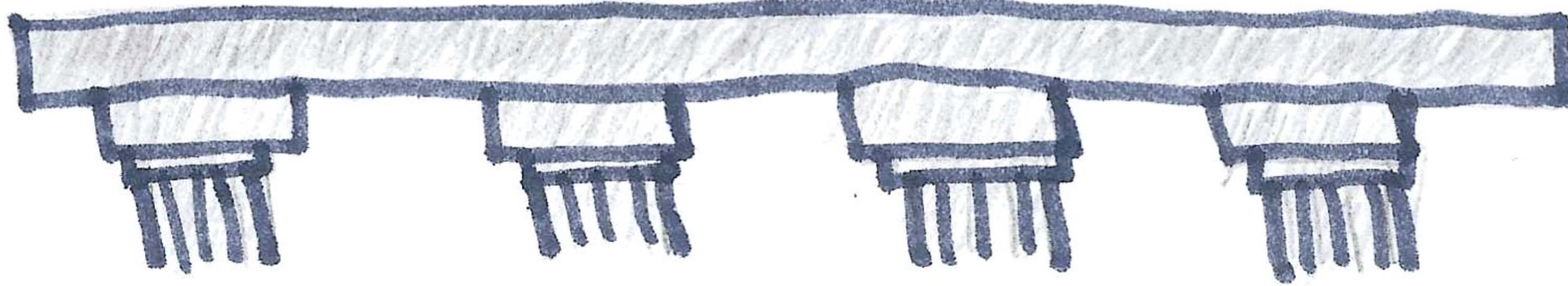
- Independientes del framework usado.
- Independientes de la interfaz gráfica.
- Independientes de la procedencia de datos.
- Independientes de componentes externos.

Se divide en las siguientes capas:

- ★ Dominio → entidades y casos de uso
- ★ Adaptadores
- ★ Detalles de implementación → frameworks y drivers
  - \* Comunicación entre las capas \*



# PRINCIPIOS SOLID



**SOLID** es el acrónimo que acuñó Michael Feathers, basándose en los principios de la programación Orientada a objetos.

**S** Single Responsibility Principle (SRP)



**O** Open Closed Principle (OCP)

**L** Liskov Substitution Principle (LSP)

**I** Interface Segregation Principle (ISP)

**D** Dependency Inversion Principle (DIP)



**S** Principio de Responsabilidad Única

**O** Principio de Abierto / Cerrado

**L** Principio de Sustitución de Liskov

**I** Principio de Segregación de la Interfaz

**D** Principio de Inversión de Dependencias

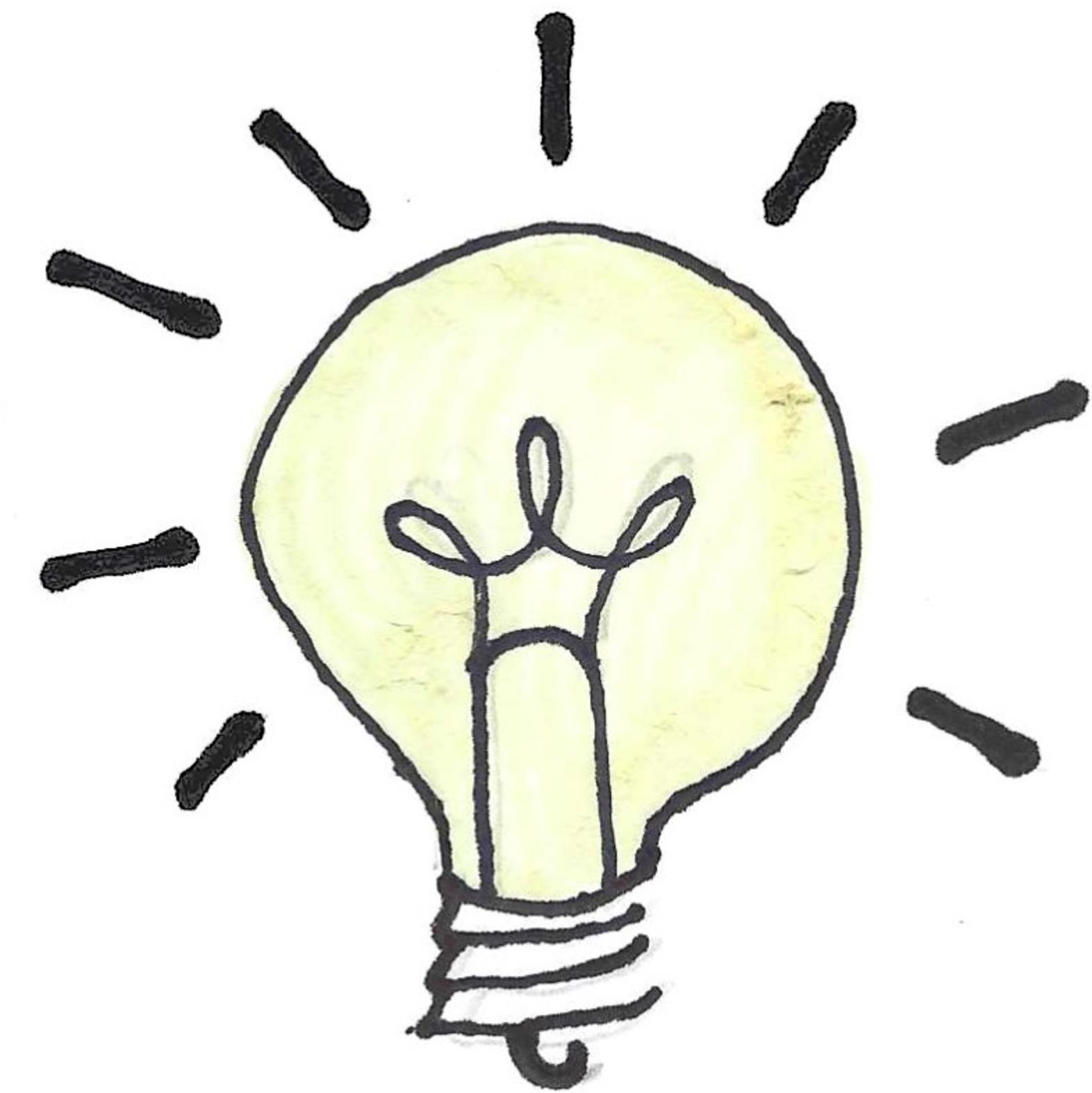




Los principios **SOLID** son eso: **principios**, es decir, buenas prácticas que nos pueden ayudar a escribir un mejor código :

- + Limpio
- + Mantenible
- + Escalable

Como indica el propio Robert C. Martin en su artículo "Getting a **Solid** Start," no se trata de reglas, ni leyes, ni verdades absolutas, sino más bien de soluciones de sentido común a problemas comunes.



El tío Bob nos dice:

**SOLID** nos ayuda a categorizar lo que es un buen o mal código y es innegable que un código limpio tendrá más a salir airosa del «control de calidad de código» **WTF/Minute.**

Aplicar estos cinco principios puede parecer algo tedioso, pero a la larga, mediante la práctica, se volverán parte de nuestra forma de programar.

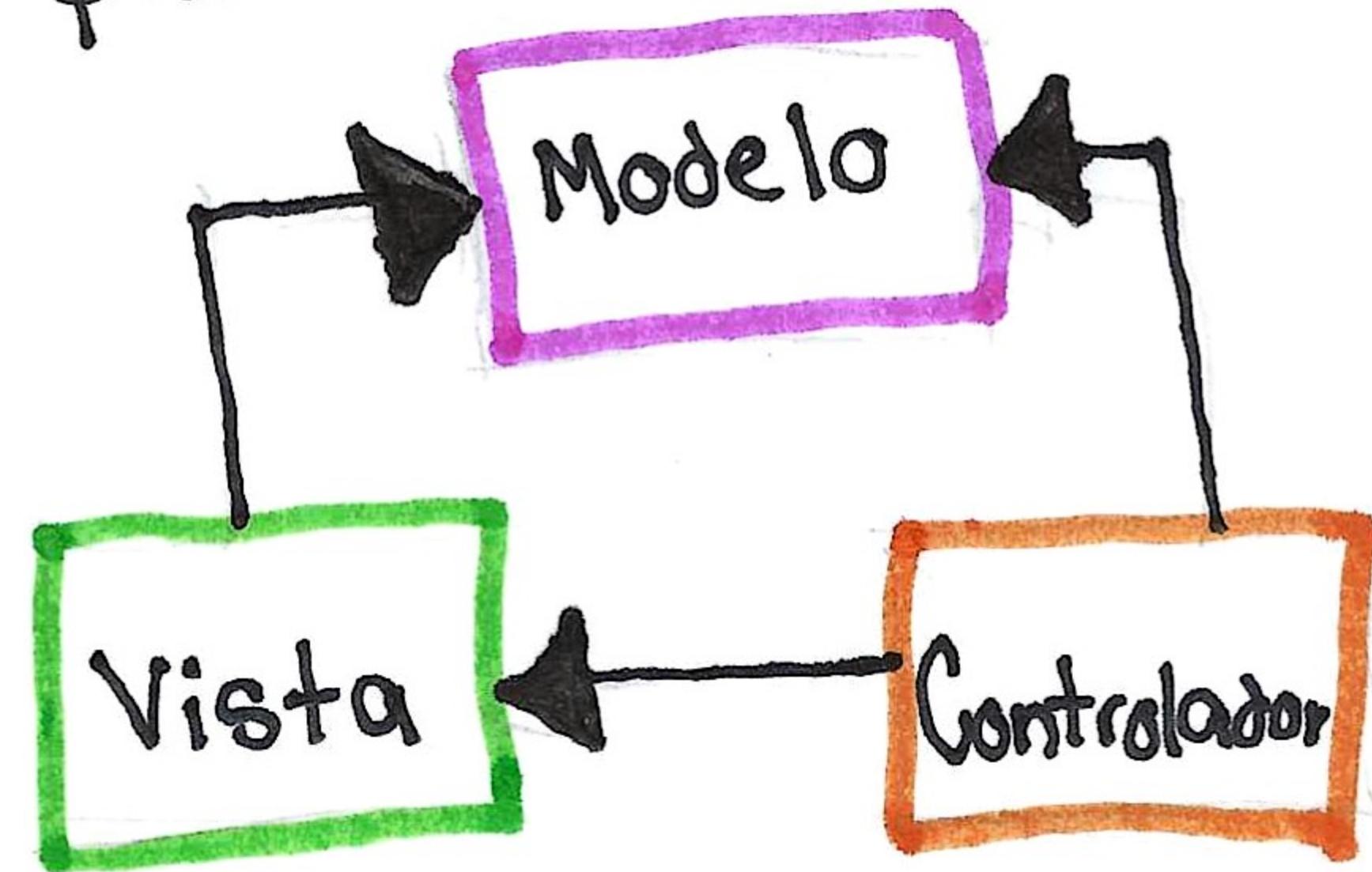
**CONCLUSION**

# Patrones de Diseño

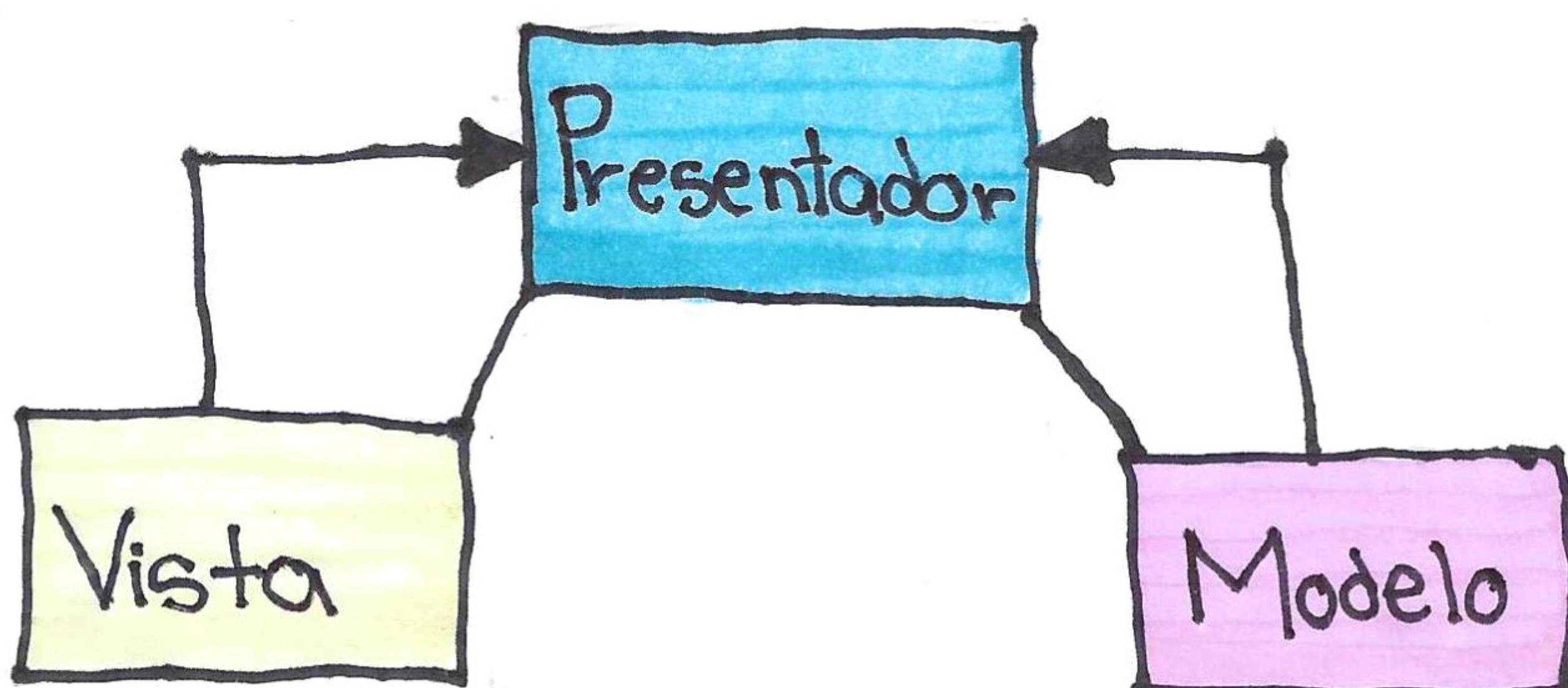
El uso de patrones facilita la solución de problemas comunes existentes en el desarrollo de software.

Dos de los patrones más utilizados para la resolución de este tipo de problemas son:

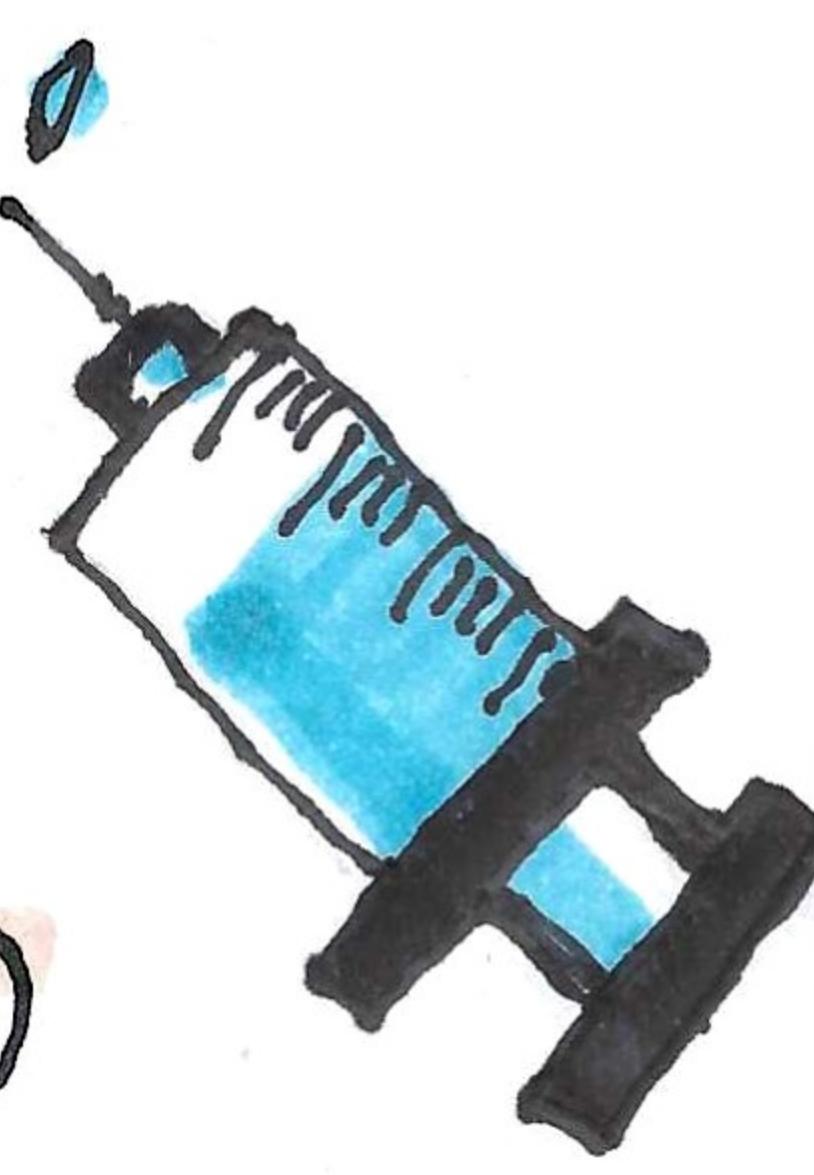
- El **MVC** es uno de los más conocidos por la comunidad de desarrolladores de software. Plantea el uso de tres capas para separar la interfaz de usuario de nuestra aplicación.: **Modelo, Vista, Controlador.**



- El **MVP** tiene como objetivo separar la interfaz de usuario de la lógica de las aplicaciones y nos permite separar aún más la vista de la lógica de negocio y de los datos.



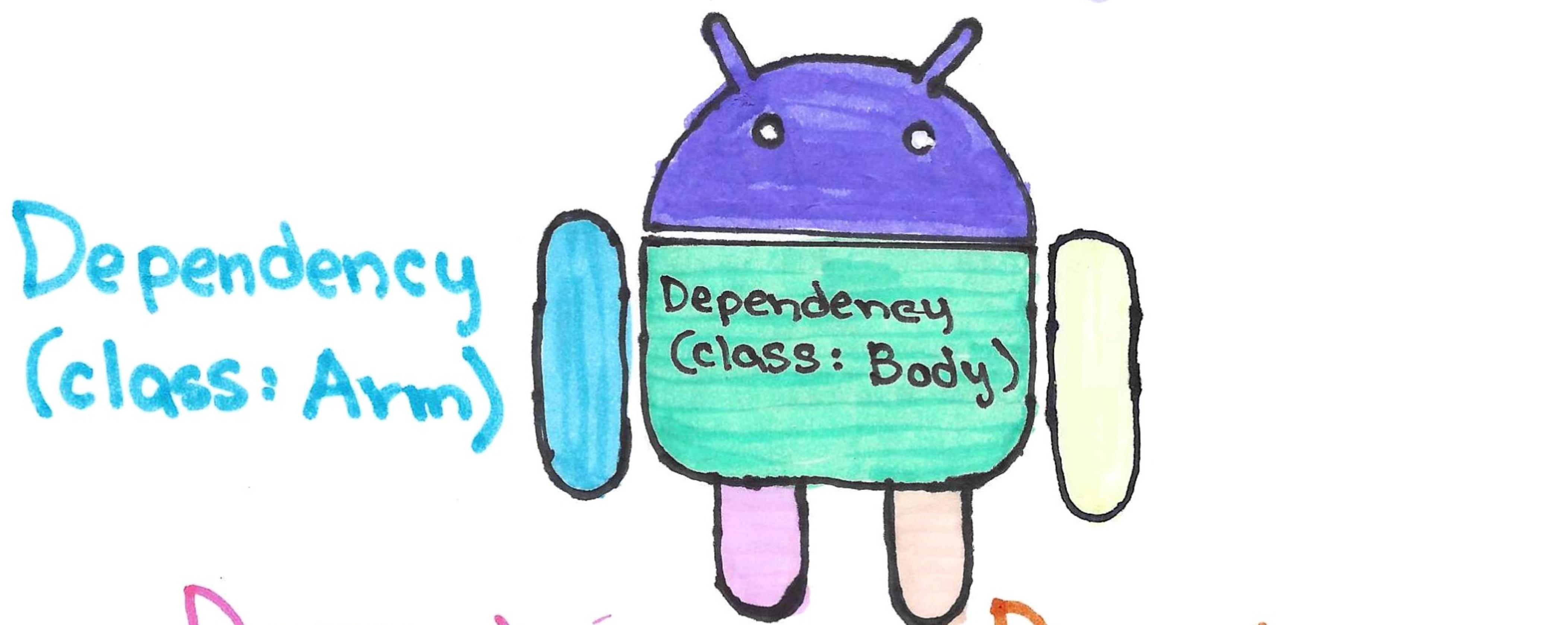
# Inyección de Dependencias



La inyección de dependencias surge para solucionar el problema de dependencia de objetos del dominio del sistema.

La inyección consiste en pasar o injectar por referencia al constructor de una clase, el objeto del que anteriormente dependía, en lugar de crear la instancia del objeto en el constructor de la clase.

Dependency (class : Head)



Dependency  
(class: Leg)

Dependency  
(class: Leg)

Flexible, Testable, Mantenible