

Trabajo de investigación 2

- Arquitectura CLEAN
- Principios Solid
- Patrones de Diseño



Arquitectura CLEAN

Se basa en la premisa de estructurar el código en capas contiguas, es decir, que solo tienen comunicación con las capas que están inmediatamente a sus lados.

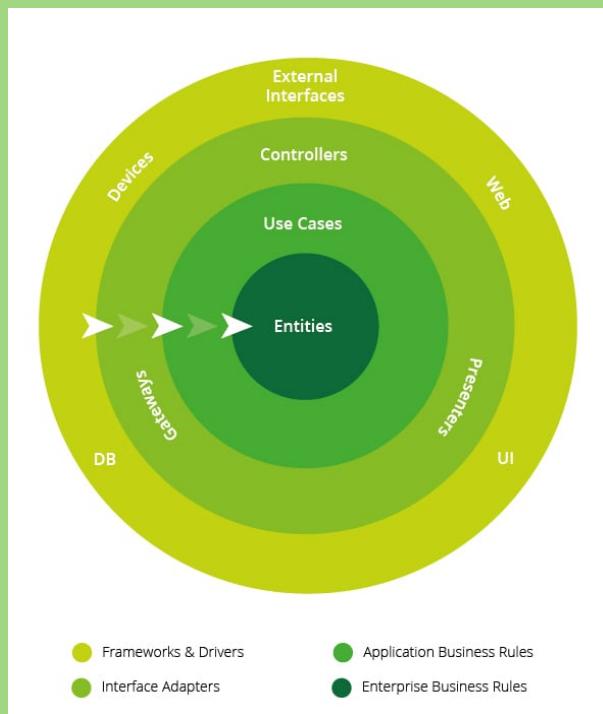
Los niveles de los que se compone Clean Architecture son los siguientes:

UI: la interfaz de usuario propiamente dicha (html, xml, etc).

Presenters: clases que se suscriben a los eventos generados por la interfaz y que responden en consecuencia, también realizan el pintado de la información en la interfaz.

Use Cases: evaluación de reglas de negocio y tomas de decisiones.

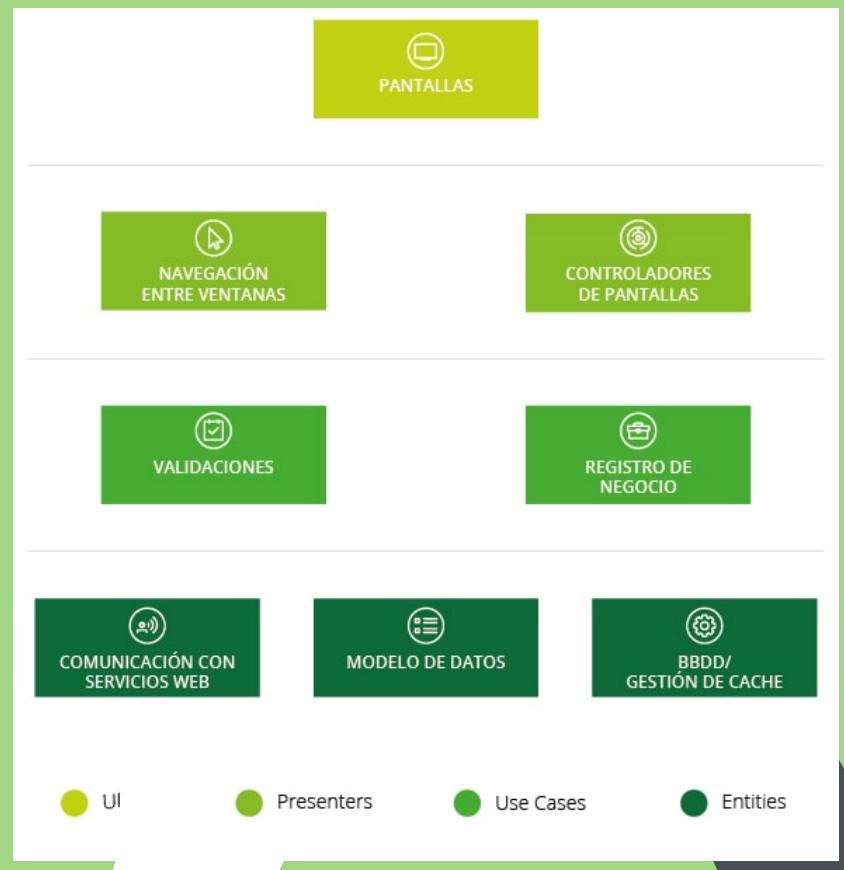
Entities: modelo de datos de la aplicación, comunicación con servicios web, cache de datos.



Cada uno de los niveles aquí representados podría considerarse dentro de la estructura de nuestra aplicación como carpetas independientes, diferentes módulos o librerías que se incluyen como dependencias del proyecto principal o incluso podríamos aplicar este tipo de estructuración sin la necesidad de reflejarlo de forma explícita en la estructura del proyecto.

Este es el gráfico original que propone Uncle Bob y se puede apreciar como los diferentes niveles anteriormente descritos sólo se comunican con los inmediatamente contiguos.

A continuación, tenemos otra interpretación del gráfico tradicional, en la que se han añadido descripción de las tareas que se suelen realizar en cada uno de los niveles.



Otra visión de las capas, parecida a la anteriormente vista es:

Capa de Modelo.

También se utiliza el nombre de Dominio o Lógica de Negocio. Está formada por las clases que representan la lógica interna de la aplicación y cómo representamos los datos con los que vamos a trabajar. Muchas clases de esta capa se conocen como POJO (Plain Old Java Object) al tratarse de clases Java puras.

Capa de Datos.

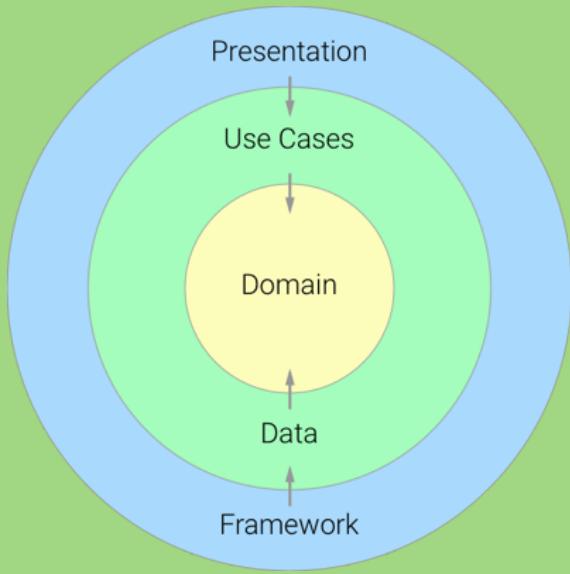
En esta capa estarían las clases encargadas de guardar de forma permanente los datos y cómo acceder a ellos. Suelen representar bases de datos, servicios Web, preferencias, ficheros JSON, etc. También es conocida como capa de almacenamiento o persistencia.

Capa de Casos de Uso.

Los casos de uso son clases que van a definir las operaciones que el usuario puede realizar con nuestra aplicación. Esta capa no sería estrictamente necesaria (por ejemplo, en Asteroides no la vamos a utilizar), pero resulta muy interesante para tener enumeradas las diferentes acciones que vamos a implementar.

Capa de Presentación.

Representa la interfaz de usuario, por lo que está formada por las actividades, fragments, vistas y otros elementos con los que interactúa el usuario.



Capa de Presentación.

Representa la interfaz de usuario, por lo que está formada por las actividades, fragments, vistas y otros elementos con los que interactúa el usuario.

Ventajas

- **Independencia:** cada capa tiene su propio paradigma o modelo arquitectónico como si se tratara de una aplicación en sí misma sin afectar al resto de los niveles.
- **Estructuración:** mejor organización del código, facilitando la búsqueda de funcionalidades y navegación por el mismo.

- **Desacoplamiento:** cada capa es independiente de las demás por lo que podríamos reemplazarla o incluso desarrollar en diferentes tecnologías. Además de reutilizar alguna de ellas en diferentes proyectos.
- **Facilidad de testeo:** podremos realizar test unitarios de cada una de las capas y test de integración de las diferentes capas entre sí, pudiendo reemplazarlas por objetos temporales que simulen su comportamiento de forma sencilla.

Principio SOLID

SOLID es un acrónimo de Single responsibility, Open-closed, Liskov substitution, Interface segregation y Dependency inversion.

SOLID tiene bastante relación con los patrones de diseño



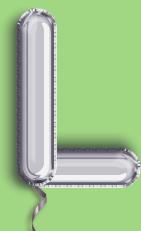
S: Single Responsibility Principle (SRP)

El principio de responsabilidad única se basa en que cada clase o método sólo debe hacer una cosa, sencilla y concreta. Si un objeto tiene un sólo cometido, éste será más fácil de mantener.



O: Open / Closed Principle

El principio Open/Closed dice que una clase/método debe estar abierto a extensiones pero cerrado a modificaciones.



L: Liskov Substitution Principle

Este principio trata sobre que los objetos de un desarrollo deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del desarrollo. Dicho de otro modo: cualquier subclase debería poder ser sustituible por la clase padre.



I: Interface Segregation Principle

Es mejor definir una serie de métodos abstractos a través de una serie de interfaces para que implementen nuestras clases. Cada interface tendrá una única responsabilidad. Es preferible tener muchas interfaces que contengan pocos métodos que tener un interface con muchos métodos.



D: Dependency Inversion Principle

El objetivo de este principio es conseguir desacoplar las clases de nuestro desarrollo. Que una clase pueda funcionar por sí sola sin depender de otra. Es difícil, pero en todo diseño de software al final suele existir un acoplamiento, pero hay que evitarlo en la medida de lo posible.

Aplicar estos principios facilitará mucho el trabajo, tanto propio como ajeno (es muy probable que tu código lo acabe leyendo muchos otros desarrolladores a lo largo de su ciclo de vida). Algunas de las ventajas de aplicarlo son:

- Mantenimiento del código más fácil y rápido
- Permite añadir nuevas funcionalidades de forma más sencilla
- Favorece una mayor reusabilidad y calidad del código, así como la encapsulación

Patrones de diseño

El uso de patrones facilita la solución de problemas comunes en el desarrollo de software. Los patrones de diseño tratan de resolver los problemas relacionados con la interacción entre interfaz de usuario, lógica de negocio y los datos.

MVC (Modelo Vista Controlador)

Su fundamento es la separación del código en tres capas diferentes, acotadas por su responsabilidad, en lo que se llaman Modelos, Vistas y Controladores.

Modelos

Es la capa donde se trabaja con los datos, por tanto contendrá mecanismos para acceder a la información y también para actualizar su estado. Los datos los tendremos habitualmente en una base de datos, por lo que en los modelos tendremos todas las funciones que accederán a las tablas y harán los correspondientes selects, updates, inserts, etc.

Vistas

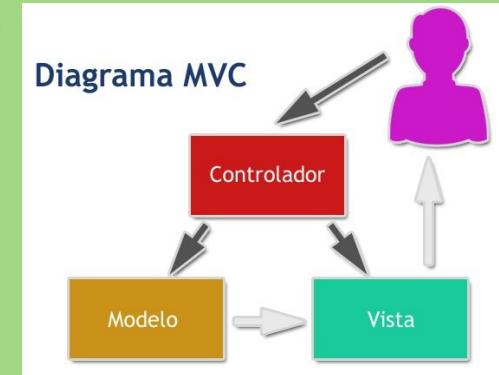
En la vista generalmente trabajamos con los datos, sin embargo, no se realiza un acceso directo a éstos. Las vistas requerirán los datos a los modelos y ellas se generará la salida, tal como nuestra aplicación requiera.

Controladores

En realidad es una capa que sirve de enlace entre las vistas y los modelos, respondiendo a los mecanismos que puedan requerirse para implementar las necesidades de nuestra aplicación.

MVP (Modelo Vista Presentador)

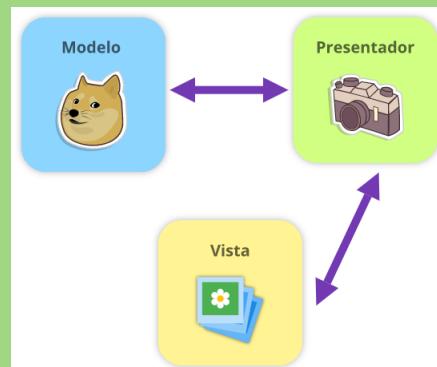
Es uno de los patrones de arquitectura de desarrollo más comunes y usados en el desarrollo nativo de Android. La necesidad de utilizar este patrón surge debido a lo complicado que puede llegar a ser el mantenimiento y escalamiento de un proyecto que va creciendo a lo largo del tiempo y líneas de código.



Modelo: Esta capa gestiona los datos. Son las clases que denominaríamos de lógica de negocio.

Vista: Se encarga de mostrar los datos. Aquí se encontrarían nuestros Fragmentos y Vistas.

Presentador: Se sitúa entre el modelo y la vista, permitiendo conectar la interfaz gráfica con los datos.



Diferencias entre ambos:

- 1- En MVC, el modelo tiene lógica extra para interactuar con la vista. En el MVP, esta lógica se encontraría en el presentador.
- 2- En el modelo MVC, la vista tiende a tener más lógica porque es responsable de manejar las notificaciones del modelo y de procesar los datos, en MVP su única función es representar la información que el presentador le ha proporcionado.