

UNIVERSIDAD DON BOSCO



Desarrollo de Software para Android (DSA441)

GRUPO TEÓRICO N°G01T

ING. ALEXANDER SIGUENZACICLO 2-2021

INTEGRANTES

ALUMNO

CARNÉ

PATRICK ERNESTO ROSALES MENDOZA

RM181976

KEVIN ADONAY MARTÍNEZ CERÓN

MC200314

STANLEY ADONAY MEJIA AMAYA

MA212116

Arquitectura Clean

Este tiene su origen en la escritura del libro “**clean code**” en el que un ingeniero de software reflexiona sobre las buenas prácticas y el estudio de patrones a la hora de escribir un software, esto nos ayudaría a facilitar el proceso de construcción del software, así como su mantenimiento,

Una arquitectura limpia es aquella donde se pretende conseguir unas estructuras modulares bien separadas, de fácil lectura, limpieza del código y testabilidad de esta forma nuestro software podría ser reutilizable o de fácil lectura para otro desarrollares que lleguen a trabajar en este.

Para que un sistema sea construido con una arquitectura limpia debe cumplir con lo siguiente:



Independientes del framework utilizado: Es decir que las librerías agregadas a nuestro proyecto no deben condicionarlos, es decir que tiene que ser solo una herramienta más que vamos a implementar en nuestro proyecto.



Testeables: es decir que debe ser testeable indistintamente de la interfaz gráfica, modelo, base de datos o peticiones a una API empleadas.



Independientes de la interfaz gráfica: debemos utilizar patrones que nos permitan cambiar fácilmente lo que es la interfaz gráfica si así lo deseamos, es decir que no tenemos que acoplar el funcionamiento de la vista con el modelo implementado.



Independientes de los orígenes de datos: debemos ser capaces de sustituir nuestro origen de datos fácilmente sin importar si estos están alojados en una base de datos local, relacional, no relacional o si estos están siendo obtenidos desde una API.







Independientes de factores externos: Debemos aislar nuestras reglas de negocio en su mundo, de tal forma que no conozcan nada ajeno a ellas.



Regla de dependencia Arquitectura Limpia.

Los diferentes elementos que miramos en la figura son los siguientes:

-  **Entidades:** Son objetos de negocio de la aplicación que estamos desarrollando que poseen propiedades y métodos.
-  **Casos de uso:** Conocidos como “interactores”, se encargan de implementar las reglas de negocio de la aplicación, orquestando el flujo de datos desde y hacia las entidades.
-  **Adaptadores de interfaz:** Son adaptadores encargados de transformar los datos desde el formato más conveniente para los Casos de uso y Entidades al formato que mejor convenga a Base de datos o Interfaz de usuario
-  **Frameworks y Drivers:** En esta capa se encuentran plataformas externas, Interfaz de usuario y Base de datos. Es la capa más externa, que debe comunicarse hacia las capas interiores.

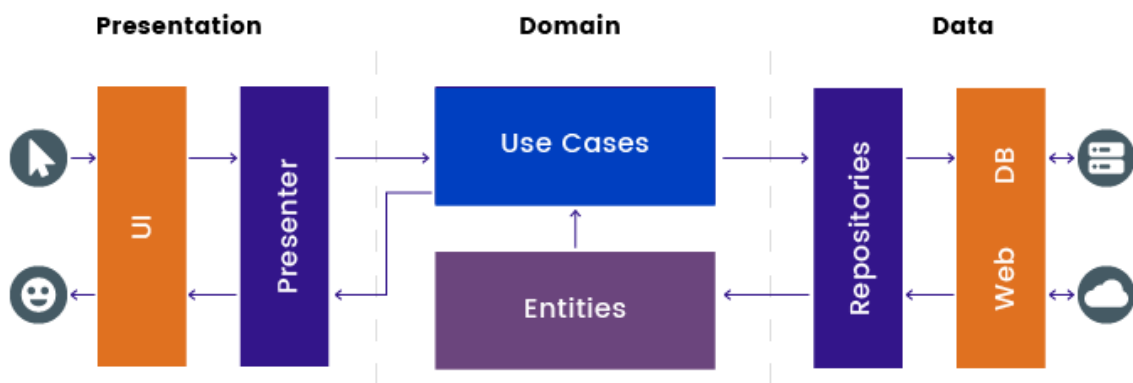



Figura 1 Ejemplo de arquitectura limpia para ser aplicada en el desarrollo de una aplicación Android.


Principios Solid




Es uno de los acrónimos más famosos en el mundo de la programación. Introducido por Robert C. Martin a principios del 2000, se compone de 5 principios de la programación orientada a objetos.

Lo más importante que se debe de saber sobre estos principios es lo siguiente.

 **Principio de responsabilidad única (S):** Una clase debe encapsular una única funcionalidad; en caso de encapsular más de una funcionalidad, sería necesario separar la clase en múltiples clases. Es un principio fundamental basado en que una clase ha de hacer aquello que debe hacer y nada más. De esta forma conseguimos que la clase sea más entendible y fácil de mantener.

 **Principio de ser abierto y cerrado (O):** Debemos preparar nuestro código para que esté abierto a extensiones y cerrado a modificaciones. Es decir, que aquello que funcione no se toque. Por otra parte, dado que nuestro sistema evolucionará con el tiempo, debe estar abierto a cambios, es decir, que podamos extender estas clases mediante el uso de clases abstractas.

 **Principio de sustitución de liskov (L):** Este principio nos que, si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido.





🔗 **Principio de segregación de interfaz (I):** ninguna clase debería depender de métodos que no usa. Por tanto, cuando creamos interfaces que definan comportamientos, es importante estar seguros de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. En caso contrario, es mejor tener varias interfaces más pequeñas.

🔗 **Principio de inversión de dependencia (D):** Gracias al principio de inversión de dependencias,

podemos hacer que el código que es el núcleo de nuestra aplicación no dependa de los detalles de implementación, como pueden ser el framework que utilices, la base de datos, cómo te conectes a tu servidor.

Ya que todos estos aspectos se especificarán mediante interfaces, y el núcleo no tendrá que conocer cuál es la implementación real para funcionar.

Beneficios de usar principios solid

Ayuda a crear
un software
mas flexible.

Ayudan a
entender mejor
las arquitecturas.

Hacen mas
simple la
creacion de tests.



PATRONES DE DISEÑO

Los patrones de diseño tratan de resolver los problemas relacionados con la interacción entre interfaz de usuario, lógica de negocio y los datos.

Dos de los patrones más utilizados para estos tipos de problemas son **MVC** (Modelo-Vista - Controlador) y **MVP** (Modelo-Vista-Presentador). Ambos tratan de separar la presentación de la lógica de negocio y de los datos. La idea principal de los patrones es que cada una de las capas tenga su propia responsabilidad.

Cada modelo o cada patrón permite a los desarrolladores no tener que reinventar la rueda, es decir, si ya existe una solución para un problema de diseño, hay que utilizarla.

Si nos familiarizamos con más patrones de diseño sabremos en qué momento aplicarlos dependiendo del problema de diseño software con el que nos encontremos.

Ahora pasaremos a ver un poco en profundidad los patrones más utilizados el primero de ellos:

MVC (Modelo Vista Controlador)

Este modelo es uno de los más conocidos y plantea el uso de 3 capas para separar la interfaz de usuario de los datos y la lógica de negocio. Estas capas son:

Modelo: Es la capa donde se trabaja con los datos, por tanto, contendrá mecanismos para acceder a la información y también para actualizar su estado.

Vista: Esta capa contiene la interfaz de usuario de nuestra aplicación. Maneja la interacción del usuario con la interfaz de usuario para enviar peticiones al controlador. Podemos tener múltiples vistas para representar un mismo modelo de datos.

Controlador: Esta capa es la intermediaria entre la Vista y el Modelo. Es capaz de responder a eventos, capturándolos por la interacción de usuarios en la interfaz, para posteriormente procesar la petición y solicitar datos o modificarlos en el modelo, retornando a la vista el modelo para representarlo en la interfaz.



Ilustración 1 Representación MVC (Modelo-Vista-Controlador)



MVP (Modelo Vista Presentador)

El patrón **MVP** deriva del **MVC** y nos permite separar aún más la vista de la lógica de negocio y de los datos. En este patrón, toda la lógica de la presentación de la interfaz reside en el Presentador, de forma que este da el formato necesario a los datos y los entrega a la vista para que esta simplemente pueda mostrarlos sin realizar ninguna lógica adicional.

Estas capas son:



Modelo: Es la capa encargada de gestionar los datos; su principal responsabilidad es la persistencia y almacenamiento de datos. En esta capa encontramos la lógica de negocio de nuestra aplicación, utilizando los interactores para realizar peticiones al servidor con el fin de obtener o actualizar los datos y devolvérselos al presentador.



Vista: La vista no es una Activity o un Fragment, simplemente es una interfaz de comportamiento de lo que podemos realizar con la vista. Sin embargo, son las Activity o Fragments los encargados de atender a la interacción del usuario por pantalla para comunicarse con el presentador. Únicamente deben implementar la interfaz de la vista, que servirá de puente de comunicación entre el presentador y las Activity o Fragments, de forma que a través de los métodos implementados representen por pantalla los datos.



Presentador: Es la capa que actúa como intermediaria entre el modelo y la vista. Se encarga de enviar las acciones de la vista hacia el modelo de tal forma que, cuando el modelo procese la petición y devuelva los datos, el presentador los devolverá asimismo a la vista. El presentador no necesita conocer la vista, ya que se comunica con ella a través de una interfaz.

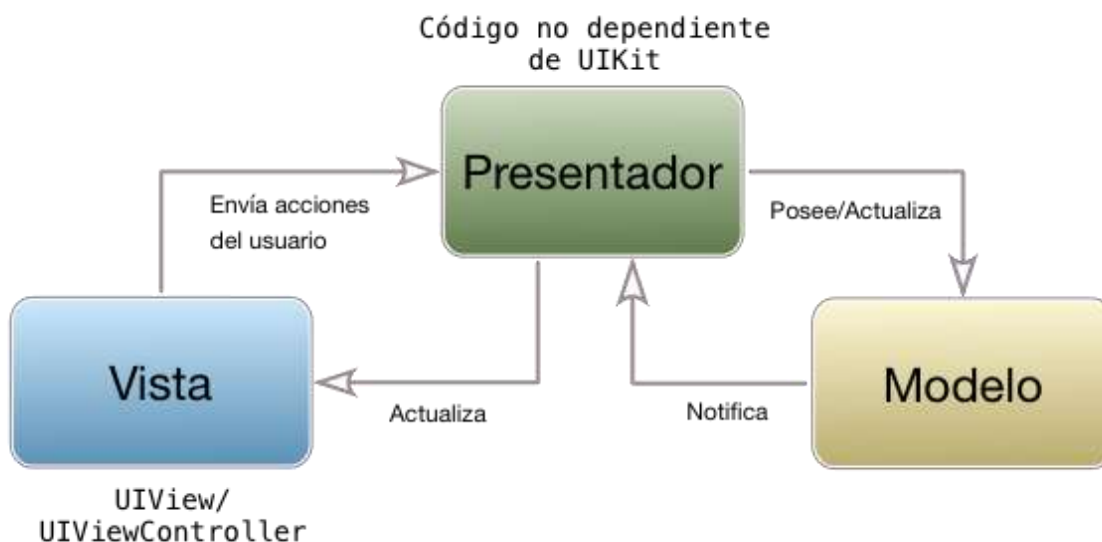


Ilustración 2 Representación MVP (Modelo-Vista-Presentador)

Comparando MVC y MVP

MVC	MVP
El controlador puede tener múltiples vistas en las que representar los datos de forma distinta, por lo que es el controlador el que decide qué vista utilizar	Cada vista tiene asignada su presentador, responsable de gestionar su lógica de representación.
modelo de MVC es un modelo sin apenas lógica, representa el estado del dominio	Contiene lógica de negocio del sistema.
Contiene la lógica de negocio del sistema y sirve de intermediario entre el modelo y la vista	Únicamente se encarga de la lógica representación de la vista, así como de hacer de intermediario entre el modelo y la vista

Patrón Observer se basa en dos objetos con responsabilidades bien definidas en los cuales tenemos los 2 siguientes:



Observables: Son objetos con un estado concreto, capaces de informar a los subscriptores suscritos al observable y que desean ser notificar sobre cambios de estado de estos



Observadores: Son objetos que se subscriben a los objetos observables y que solicitan ser notificados cuando el estado de estos observables cambie.

Link Repositorio aplicación ejemplo

<https://github.com/AdonayMejia/EjemploMVP>