Diseño de software para Android

Docente:

Alexander Alberto Siguenza Campos

Integrantes:

- Andrea Elizabeth Blanco Suárez BS200382

- Jason Alexander Fuentes Reyes FR200028

- Byron Roberto Sánchez Carrillo SC170935

- Diego Alberto Sandoval Rivera SR200029

Fecha de Entrega:

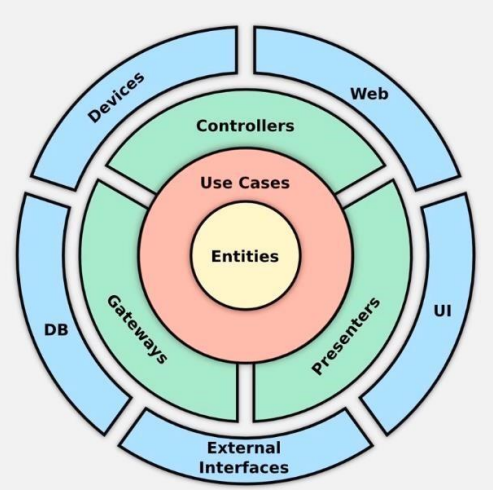
09/04/2022

CAPITULO 2

Arquitectura clean

Esta arquitectura se basa en una serie de principios los cuales poseen un objetivo y finalidad principal, la cual se base en mantener toda la lógica de una aplicación separada, y de esta manera poder obtener una lógica más sostenible, escalable y fácil de comprender.

Para la arquitectura clean una app se debe dividir en distintas responsabilidades y cada una debe llevar su propia función, cada una de estas esta gráficamente representada por distintas capaz, siendo las inferiores las que no poseen conocimiento sobre su capa superior y las superiores si poseen conocimientos de las capas interiores.



En la siguiente grafica podemos visualizar las capas existentes, retomando lo anterior, podemos hacer énfasis en la capa “casos de uso” la cual debe ser estrictamente independiente de las demás capas externas, es decir, no debe o no tiene que conocer la estructura, software, etc… implementados las capas superiores a ella.

La capa de entidades y casos de uso forman el dominio de la app, es decir son los elementos o capas que le dan vida a la aplicación.

Entidades : forman parte de la lógica y datos de negocio de nuestra aplicación y pueden ser reutilizables.

Casos de uso: Estos forman parte de la lógica respectiva al flujo de la aplicación, los cuales se acoplan a la lógica existente de las entidades.

Adaptadores: Transforman toda la información que poseen nuestras vistas a información que pueda ser interpretada o requerida para los casos de uso.

Frameworks: En esta se aloja todas las bases de datos, Interfaz del Usuario, Plataformas de uso externo.

Los sistemas construidos con esta arquitectura deben tener las siguientes características y ventajas:

- El desarrollo o dominio de la aplicación no debe depender del framewoks que se utiliza.
- Los datos deben ser testebles fácilmente, sin hacer uso de la interfaz, base de datos, etc…

- Se deben realizar cambios de la interfaz del usuario fácilmente, sin complicaciones.
- Las bases de datos locales, relacionales, peticiones, etc… pueden ser sustituidas por otras o actualizadas de manera sencilla. o L a capa de dominio de la aplicación no debe tener dependencia de ninguna otra.

La arquitectura clean es recomendada para aplicaciones que han sido programadas para un periodo medio-extenso o extenso, ya que su utilización favorece mucho a poder realizar cambios que vayan surgiendo o necesitando con el paso del tiempo.

CAPITULO 3

Patrones de diseñó

Los patrones de diseño de una estructura en el desarrollo de software son muy útiles para la solución de problemas que pueden surgir durante el desarrollo de un proyecto. Estos patrones de diseño tienen como propósito resolver los problemas vinculados con la interfaz de usuario(UI), la lógica de negocios y los datos que se manejan.
Los patrones de diseño dentro del desarrollo de apps de Android son importantes dentro de la organización de un proyecto, entre los más populares está el patrón MVC (Modelo Vista Controlador) y el MVP (Modelo Vista Presentador).

Para entenderlos es necesario saber qué son. Por una parte el MVC además de ser uno de los más conocidos dentro de la comunidad de desarrolladores para android, acá existen 3 capas, modelo, vista y controlador, aquí el componente principal es el controlador pues es la intermediaria entre la vista y el modelo, por ende es la que da el feedback ante eventos, es decir, captura la interacción del usuario en la interfaz para procesarla y solicitar datos o cambiarlos dentro de lo que viene siendo el modelo.
El MVP por su parte tiene su característica que aleja más la vista de la lógica de negocio y por ende de los datos. Acá pues la diferencia radica en que existe un Presentador, es decir, Modelo, Vista y Presentador. Aquí también este tiene la función de servir como intermediario ante la vista y el modelo pero en este caso el presentador se comunica con la vista por medio de una interfaz.

.Diferencias:

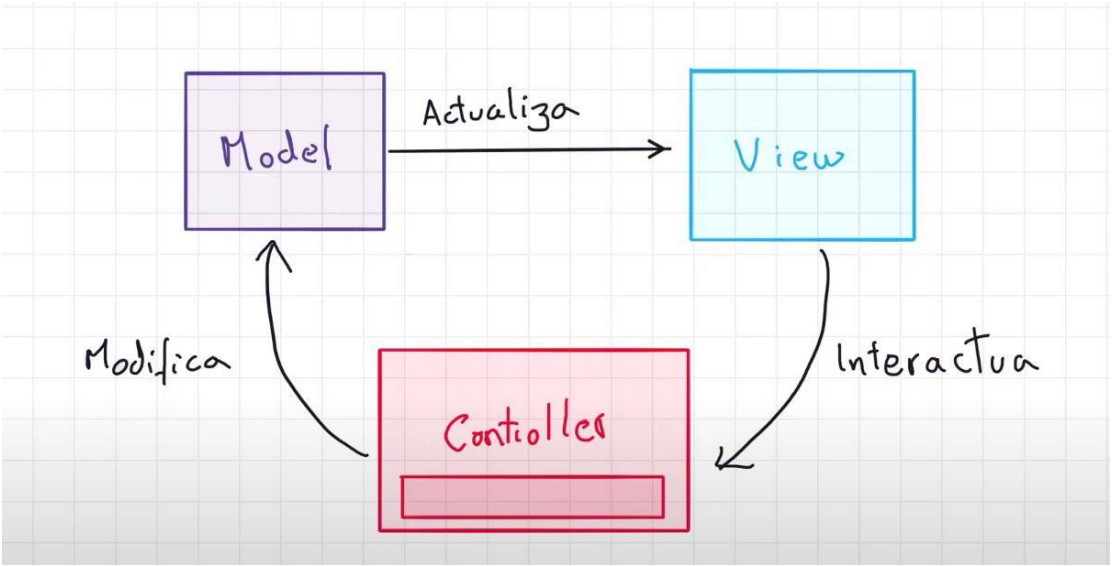
MVC	MVP
● Múltiples vistas en el controlador ● Controlador decide qué vista utilizar ● Lógica débil ● El controlador contiene la lógica de negocio	● Cada vista tiene su presentador ● Contiene una lógica más concreta El ● rol del presentador, contrario al controlador es que, maneja la lógica únicamente de la representación de la vista asignada ● El presentador se comunica con el modelo a través del patrón Observer

Cabe aclarar que el patrón observer es el que consiste de dos participantes, el sujeto y los observadores, también llamado patrón de comportamiento, es el que observa y procesa los algoritmos, las relaciones y los roles entre objetos.

Entonces, para dar un veredicto. El modelo y controlador del MVC viene siendo el modelo del MVP, por lo que la vista del MVC sería el equivalente del presentador y la vista del MVP.

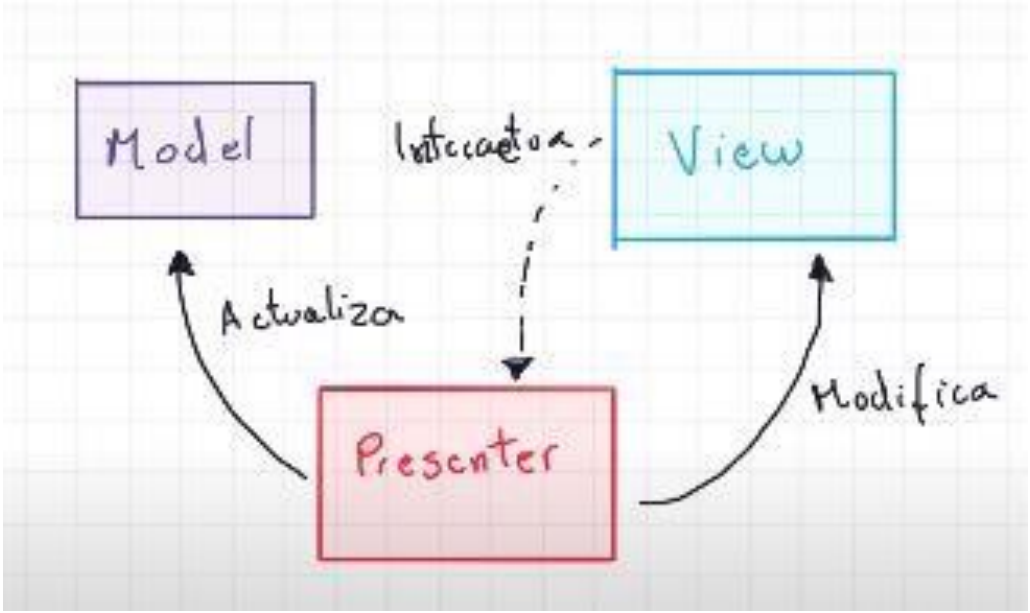
Para que quede más claro se muestran las siguientes ilustraciones.

MVC



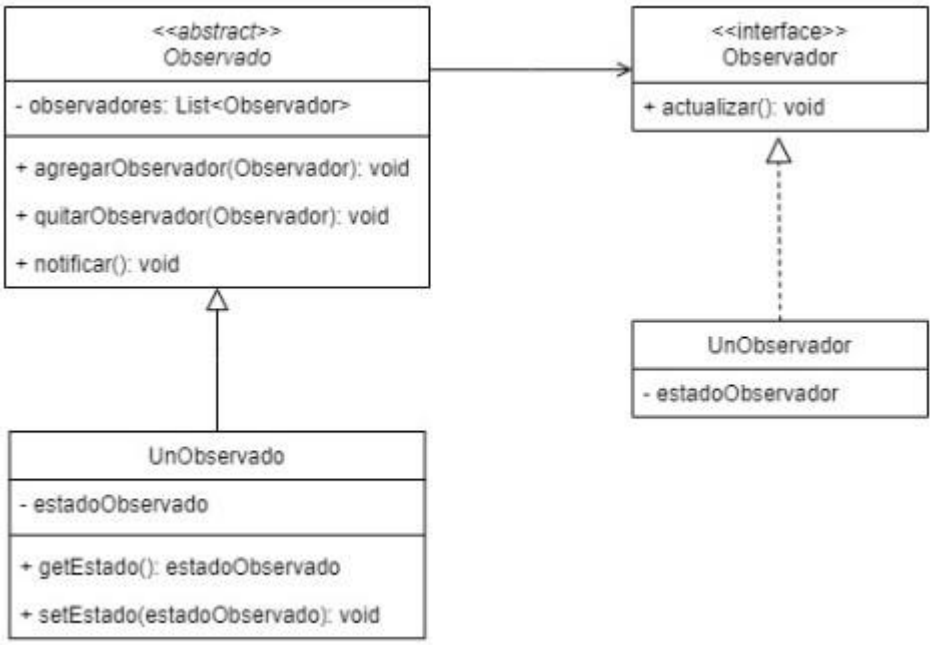
Jesús Angulo, JS.(S.F). Arquitecturas Móviles. visual-foxpro-programmer.com. Recuperado de: https://visualfoxpro-programmer.com/talk-about-advantages

MVP



Jesús Angulo, JS.(S.F). Arquitecturas Móviles. visual-foxpro-programmer.com. Recuperado de: https://visualfoxpro-programmer.com/talk-about-advantages

PATRON OBSERVER



Ezequiel Ortega Mateo.E.O.(2020). Patrón de comportamiento - Observer. SOMOSPNT. Recuperado de: https://somospnt.com/blog/155-patron-de-comportamiento-observer

CAPITULO 4

Principios Solid

Son una serie de normas que sirven para facilitarnos a los desarrolladores la labor de crear programas legibles y mantenibles es uno de los acrónimos más famosos del mundo de la programación creado por Robert C. Martin en el año 2000. Los 5 principios Solid son

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)

Principio de responsabilidad única (S):

Todas las clases deben de ser responsables de una sola función (el término “decoupled” en inglés), lo que provoca que tengas que elaborar varias clases para varias funciones, en el momento de utilizar una clase que realice más de una función estaría incumpliendo.

“Una clase debe tener solo una razón para cambiar” – Uncle Bob

Se necesita analizar cada clase para poder llegar a separar cada función lo que no es trabajo fácil. Gracias a este principio nuestras clases tendrán un bajo acoplamiento y a la hora de realizar cambios menos clases se verán afectadas.

Principio de ser abierto y cerrado (O):

El código debe estar listo para que esta pueda tener extensiones pero que no se pueda modificar lo anteriormente realizado. Fue nombrado por primera vez por Bertrand Mayer, un programador francés, en su libro Object Oriented Software Construction en 1988.

Este principio se logra por medio de la utilización de clases abstractas que nos permitan crear objetos que hereden de estas clases lo que evitaría que se altere su funcionamiento

Ejemplo:

Public class Persona {

String nombre;

String apellidos;

Public String getPersona(){

String s;

S=nombre+" "+apellidos;

Return s;

}

}

Principio de sustitución de Liskov (L):

Toda clase que extienda funciones de una clase padre debe de implementar toda la funcionalidad de la clase base sin alterar el funcionamiento, La primera en hablar de él fue Bárbara Liskov una reconocida ingeniera de software americana.

Esta nos dice que si se usa una clase extendida tenemos que poder utilizar cualquiera de sus clases hijas, en el caso de que alguna subclase de alguna clase padre no implemente alguna de las propiedades o métodos se estaría incumpliendo este principio

Principio de segregación de interfaz (I):

A la hora de heredar clases debemos de evitar que existan métodos de interfaces que no sean implementados en el caso de que no sea necesaria para la clase. Este viene a decir que ninguna clase debe depender de métodos que no utiliza, esto surge cuando interfaces intentan definir mas cosas de las debidas.

Eso significa que si al implementar una interfaz algunos métodos no están siendo utilizados y te hace falta dejarlos vacíos o lanzar excepciones eso significa que estas violando este principio

Principio de inversión de dependencia (D):

Gracias a este principio podemos hacer que el código no dependa de los detalles de implementación, debido a que la dependencia entre clases es un problema a la hora de introducir cambios al sistema. A la hora de instanciar estas clases debes conocer sus funciones para que no dependa de detalles.

Cualquier instanciación de clases complejas o módulos es una

Violación de este principio a la hora de realización de tests esto puede ser un problema por que no se realizara de forma rápida lo que provocaría un problema. Una de las mejores técnicas para lidiar con las colaboraciones entre clases, produciendo un código reutilizable, sobrio y preparado para cambiar sin producir efectos bola de nieve

Fuentes de referencia:

- *Patrón de comportamiento - Observer*. (s. f.). somospnt. https://somospnt.com/blog/155-patron-de-comportamiento-observer
- *Android: del diseño de la arquitectura al despliegue profesional*. (s. f.). Alfaomega, Marcombo.
- Doe, J. (2022, 10 abril). *Hable sobre las ventajas y desventajas de MVC, MVP y MVVM*. Modo Android. https://visual-foxproprogrammer.com/talk-about-advantages
- *Clean architecture para Android con Kotlin: una visión pragmática*. (2018, octubre 1). DevExperto, por Antonio Leiva. https://devexperto.com/clean-architecture-android/