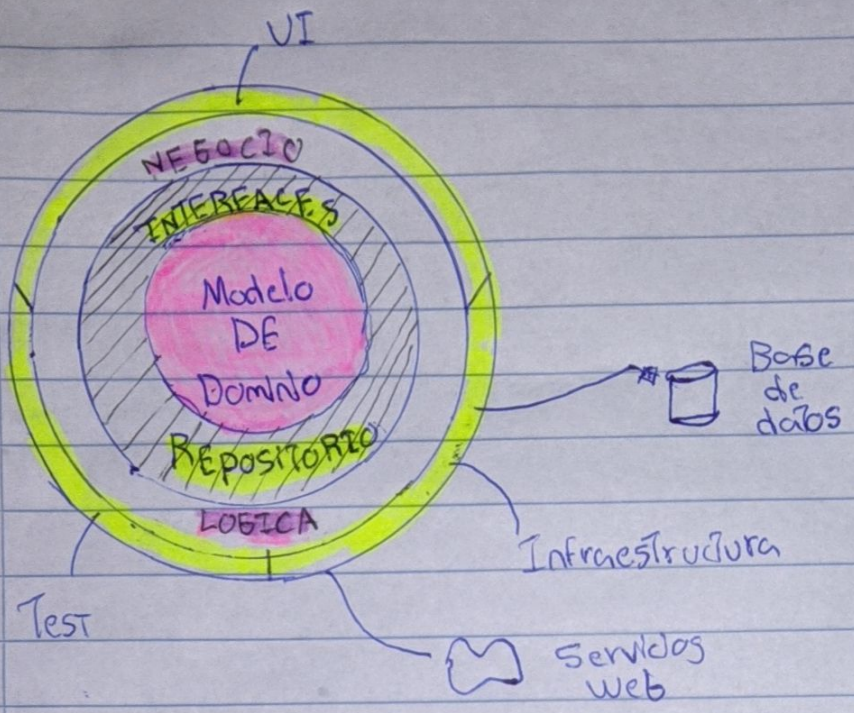
¿Qué es la arquitectura limpia?



Es aquella que pretende desarrollar estructuras modulares separadas que sean fáciles de leer, que muestren un código limpio y proporcione una buena Testeabilidad.

Según uncle Bob el código limpio debe regirse por los siguientes principios:

- **Independiente del framework**: Las librerías que se usen no deben condicionar el código sino que se tiene que acoplar con la estructura.

- **Testeables**: La lógica de la aplicación debe ser Testeable independientemente de la interfaz gráfica, modelo, base de datos o peticiones http.

- **Independientes de la interfaz gráfica**: Se debe buscar la manera de cambiar fácilmente la Interfaz grafica para ello se deben aplicar patrones de diseño de software como PVP, MVVM o MVC.

- Independientes de los orígenes de datos: Se debe contar con una flexibilidad a la hora de sustituir el origen de datos sin importar si se encuentra en una base de datos local, ficheros o peticiones a una api. para asegurar este apartado se utilizan patrones de diseño como el patrón repositorio.

- Independientes de factores externos: Las reglas de negocio no deben conocer nada ajeno a ellas.

Otro aspecto fundamental en el que se basa la arquitectura limpia es en la separación de responsabilidades dentro de las capas. En el interior se encuentra la lógica de negocio, luego el transporte o comunicación y en la parte más externa se encuentran las vistas, bases de datos e interfaces externas, etc.

A su vez existe un sentido para recorrer las capas, desde la capa más externa hasta el interior. Por lo tanto con esta separación la capa interior no debe saber nada de las capas exteriores. Esto significa que la lógica de negocio no debe utilizar ningún elemento de otra capa.

CAPÍTULO 3

PRINCIPIOS SOLID

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.

PRINCIPIO DE LA RESPONSABILIDAD UNICA (S)

El principio nos dice que una clase debe contener una única funcionalidad, caso contrario habrá que separar las clases en múltiples clases. De esta forma conseguimos que la clase sea más entendible y fácil de mantener.

PRINCIPIO DE SER ABIERTO Y CERRADO (O)

Este principio nos indica que nuestro código debe estar abierto a extensiones y cerrado a modificaciones, es decir, que un código ya escrito y finalizado no se toque, ya que podría afectar a su funcionamiento. Sin embargo, dado que nuestro sistema evolucionará con el tiempo, debe estar abierto a cambios, es decir, que podamos extender las clases por medio de clases abstractas, que nos permitan crear nuevos objetos que hereden de estas clases, sin alterar su funcionamiento.

Ejemplo:

Public abstract class figura { ... }

↓ Herencia

Public class Trinangulo extends figura { ... }

PRINCIPIO DE SUSTITUCION DE LISKOV (L)

El Principio nos dice que toda clase que extienda la funcionalidad de una clase base, debe hacerlo sin alterar su funcionamiento. De esta forma, cualquier subclase que herede de la clase padre podrá ser sustituida por otra subclase, sin afectar el comportamiento de la clase padre. Si una clase hija no implementa una propiedad o método de la clase padre, la hija no puede ser subclase de la clase base, puesto que estaría violando el principio de sustitución de Liskov.

Ejemplo

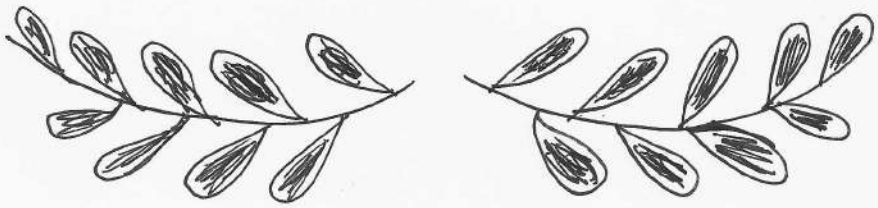
```
Public abstract class figura{...}

Public class Triangulo extend figura{...}  -> Subclase de figura

Public abstract class figura{...}

Public class circulo{..}  -> No puede ser subclase
```

PRINCIPIO DE SEGREGACION DE INTERFAZ (I)

El principio nos dice que no debemos tener métodos no implementados en nuestras clases extendidas, muchas veces se da el caso que tenemos métodos que nunca implementamos. Para evitar esto, nos recomienda dividir la interfaz en interfaces más pequeñas, así todos los métodos que implementemos tendrán su propósito.

PRINCIPIO DE INVERSION DE DEPENDENCIA (D)

Este principio establece que, en el dominio de nuestro sistema, debemos utilizar interfaces o abstracciones, elementos que cambien con poca frecuencia, de tal forma que sean las concreciones de menor las que dependan de elementos y no a la inversa.



CAPITULO 4:

PATRONES DE DISEÑO

Los patrones de diseño facilitan la solución de problemas comunes existentes en el desarrollo de software con la interacción entre interfaz de usuario, lógica de negocio y los datos.

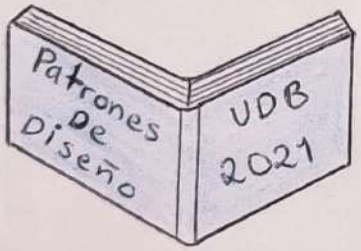
Tipos de patrones de diseño

Se clasifican en:

1. Creacionales: su objetivo es resolver los problemas de creación de instancias.

2. Estructurales: se ocupa de resolver problemas sobre la estructura de las clases.

3. Comportamiento: Ayudan a resolver problemas relacionados con el comportamiento de la aplicación. (interacción y responsabilidad entre objetos y clases de la aplicación).



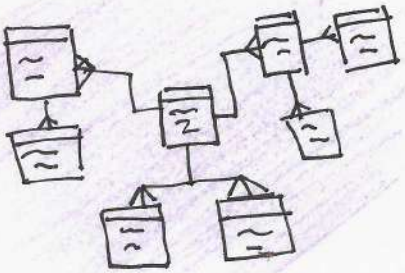
Dos patrones más utilizados
- MVC (Modelo Vista Controlador)
- MVP (Modelo Vista Presentador)

Ambos tratan de separar la presentación de la lógica de negocio y de los datos. Esto facilita el desarrollo de una aplicación y favorece su mantenimiento.

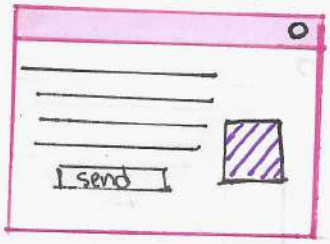
La idea principal es que cada una de las capas tenga su propia responsabilidad.

El patrón de diseño MVC es uno de los más conocidos por la comunidad de desarrolladores de software. Plantea el uso de 3 capas para separar la interfaz de usuario de los datos y la lógica del negocio. Estas capas son:

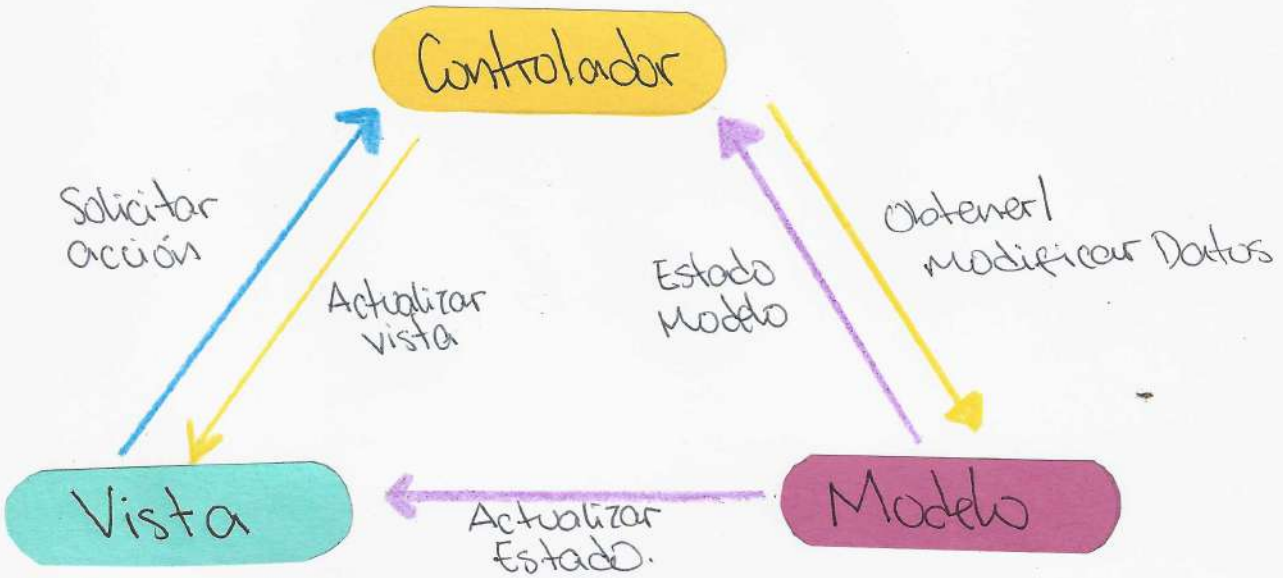
Modelo: Contiene el conjunto de clases que definen la estructura de datos con los que vamos a trabajar en el sistema. Su principal responsabilidad es el almacenamiento y persistencia de los datos de nuestra aplicación. El modelo es independiente de la representación de los datos en la vista.



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

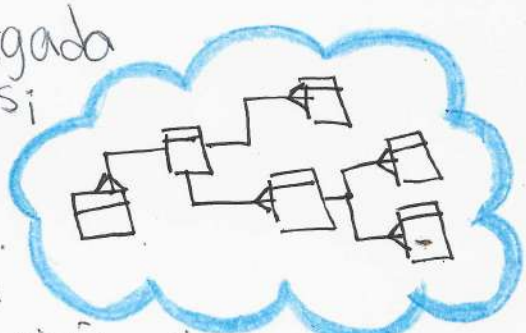


Controlador: Capa intermediaria entre la vista y el modelo. Es capaz de responder a eventos capturandolos por la interacción de usuarios en la interfaz, para posteriormente procesar la petición y solicitar datos o modificarlos en el modelo, retornando a la vista el modelo para representarlo en la interfaz.

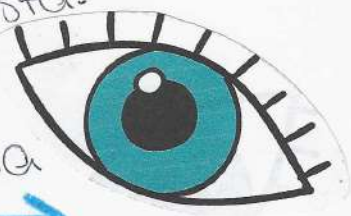


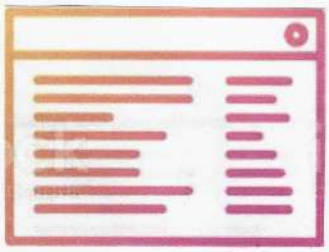
El patrón MVP deriva del MVC y nos permite separar aún más la vista de la lógica del negocio y de los datos. En este patrón, toda la lógica de la presentación de la interfaz reside en el Presentador, de forma que este da el formato necesario a los datos y los entrega a la vista para que esta simplemente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

Modelo: Es la capa encargada de gestionar los datos; su principal responsabilidad es la persistencia y almacenamiento de datos. En esta capa se encuentra la lógica del negocio de la 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 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





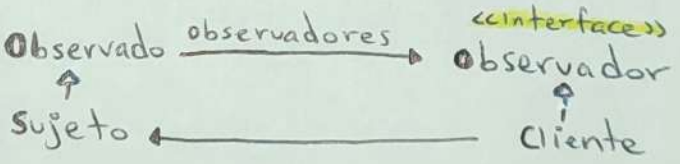
Para comunicarse con el presentador. Únicamente deben implementar la interfaz con la vista, que servirá de puente de comunicación entre el presentador y las Activities de forma que a través de los métodos implementados representen por pantalla los datos.

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 usando una interfaz.

File Edit View Run Terminal Help X

Patrones de diseño •

```
<h2> Patrón Observer </h2>
<p> El patrón Observer se basa en dos objetos
    con una responsabilidad bien definida: </p>
<ol>
    <li> Observables </li>
    <p> Son objetos con un estado concreto,
        capaces de informar a los suscriptores
        suscritos al observable y que desean ser
        notificados sobre cambios de estado
        de estos objetos.
    </p> </br>
    <li> Observadores </li>
    <p> son objetos que se suscriben a los objetos
        observables y que solicitan ser notificados
        cuando el estado de los observables cambie.
    </p>
</ol>
</br>
```




```
<h2> Patrón Singleton </h2>
<p> Es un patrón de diseño creacional que
    nos permite asegurarnos de que una
    clase tenga una única instancia, a
    la vez que proporciona un punto de
    acceso global a dicha instancia
</p>
```