FACULTAD DE INGENIERIA

Asignatura: Desarrollo de Software para Móviles (DSM01T)

CICLO ACADEMICO: 01-2021

Título:

"Tarea de Investigación #2"

Docente:

Ing. Alexander Alberto Sigüenza Campos

Presentado por:

Apellidos, Nombres	Carné
Pleitez Hércules, Kevin Eliu	Ph161929
Valle Serrano, Oswaldo	vs161940

Soyapango, 28 de abril de 2021

Arquitectura Clean

¿Qué es la arquitectura Clean?

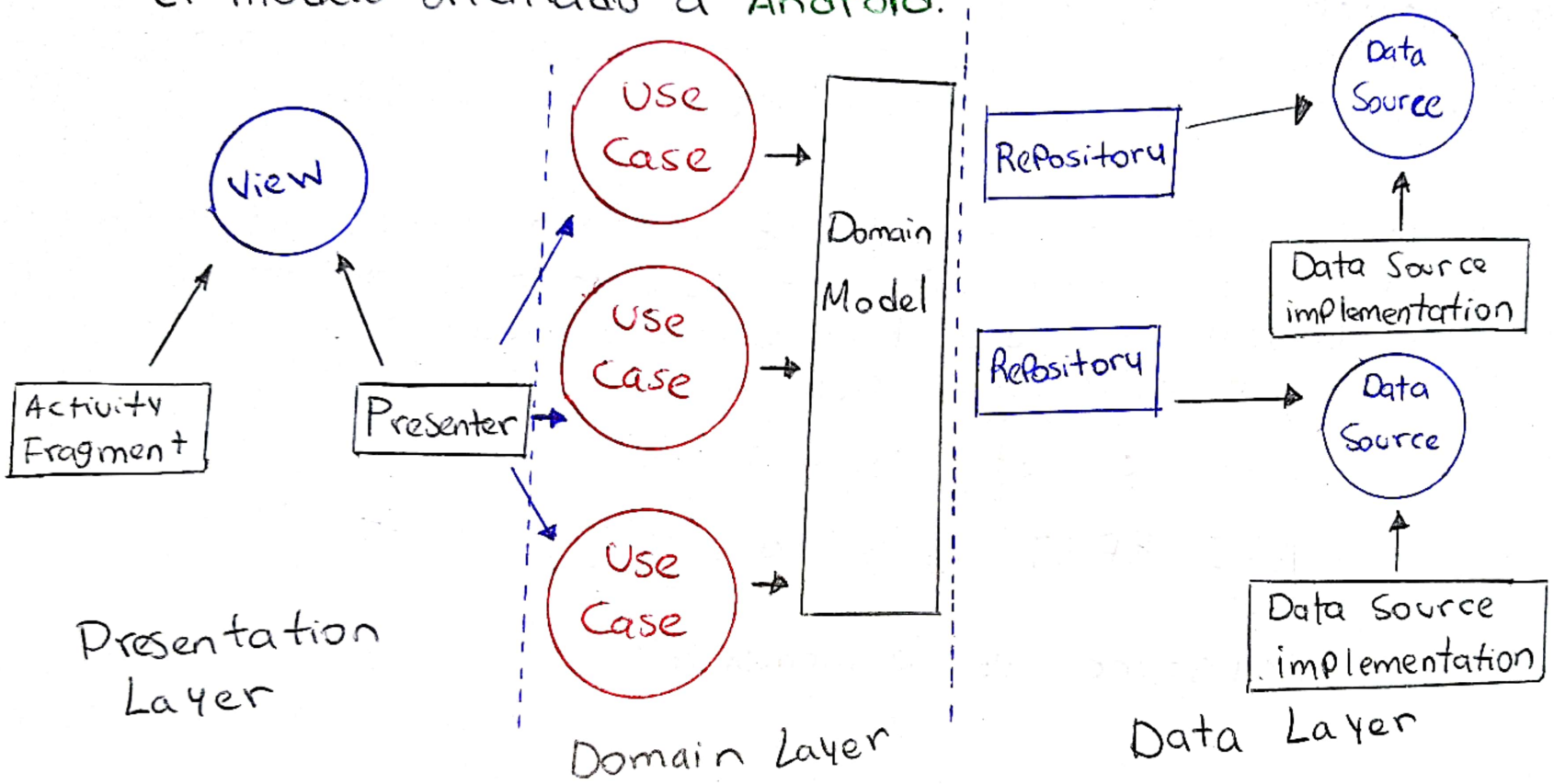
Es una Práctica de desarrollo de Software que permite organizar el desarrollo de código en un Proyecto; Separarlo y organizarlo en capas y cumpliendo con un Propósito cada una de estas que mientras más se extienda el Proyecto su complejidad de entendimiento no se vuelva una tarea difícil o a veces imposible de realizar

Esta Práctica tiene su origen en un artículo escrito por Robert C. Martin. Apodado por muchos como 'Uncle Bob', titulado 'The Clean Archi-tecture'. Este ingeniero de software de origen estado unidense también es autor de otro libro que lleva

Por título Clean Code.

La arquitectura Clean su impacto reside en organizar un Proyecto de manera interna, como organizar Pantallas o bibliotecas Por carpeta Porejemplo, de esta forma se facilita la compresión del Proyecto si bien cuando es desarrollado por un sólo usuario comprende que sucede Pero por nuevos usuarios que deseen Participar no sabrían por donde comenzar.

La arquitectura Clean tiene un modelo general, aquí veremos el modelo orientado a Android.



Presentation Layer: Es la capa en donde ocurre todo lo relacionado a cómo funcionan las vistas normalmente activities y Fragmentos los cuales contienen lógica pero únicamente enfocada en cómo funciona la vistas, es decir, lo que se muestra al usuario

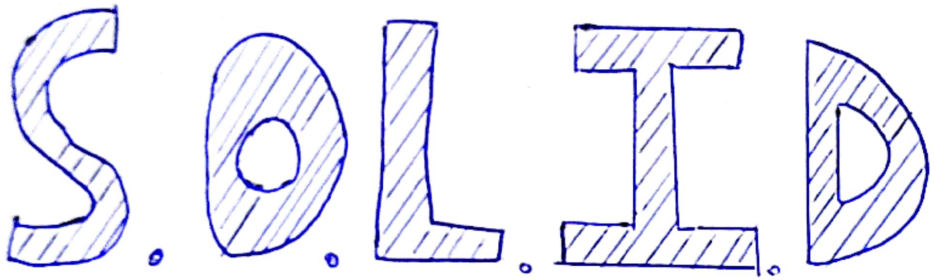
Domain Layer: Toda la lógica ocurre en esta capa aquí solo lo enfocado a que tiene que hacer la aplicación. Se encuentran entidades y casos de uso regularmente es solo código.

Data Layer: Es la capa en donde se obtienen todos los datos que necesita nuestra aplicación para funcionar y los datos pueden ser proveídos por una base de datos local o de la red o incluso de la memoria. Ejemplos de estos son:

- SQLite: datos locales
- API REST: de la red
- Keystore: de la memoria

Principio SOLID

SOLID es uno de los acrónimos más famosos en el mundo de la Programación. Introducido por Robert C. Martin, además de crear la estructura de trabajo Clean. También diseñó SOLID Pensando en 5 principios que debe cumplir el código Para ser factible.



- Single responsability Principle (Principio de responsabilidad única) Este principio establece que una clase, componente o microservicio debe ser responsable de una cosa. Una clase "empleado" solo debe tener tareas referentes a este y no encargarse de realizar ajenas como regresar el valor de una función de calculo de "Oficinas", por ejemplo.

- Open / Closed Principle (Principio de abierto / cerrado)

Establece que las entidades software (clases, módulos y funciones) deberian estar abiertos para su extensión, pero cerrados para su modificación, esto hace referencia a la herencia de clases Cuando un Padre hereda a un hijo sus funciones dentro de la clase, el hijo Puede llamar esas funciones y utilizarlas para extender sus Propias funciones creadas por el, sin embargo no Puede modificarlas.

- Liskov substitution Priniple (Principio de sustitución de Liskov)

Declara que una subclase debe ser sustituible por su superclase, y si al hacer esto el Programa falla, estamos violando este Principio

-> Cumpliendo con este Principio se confirmará que nuestro Programa tiene una jerarquía de clases facil de entender y un código reutilizable.

- Interface segegation Principle (Principio de segregación de la interfaz)

Este Principio establece que bs clientes no deberían verse forzados a depender de interfaces que no usan.

Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz cuya funcionalidad este cliente no usa, pero que otros clientes sí usan, este cliente estará siendo afectado por bs cambios que fuercan otros clientes en dicha interfaz Una forma de evitar es declarar diversas interfaces que se adapten a las necesidades del cliente.

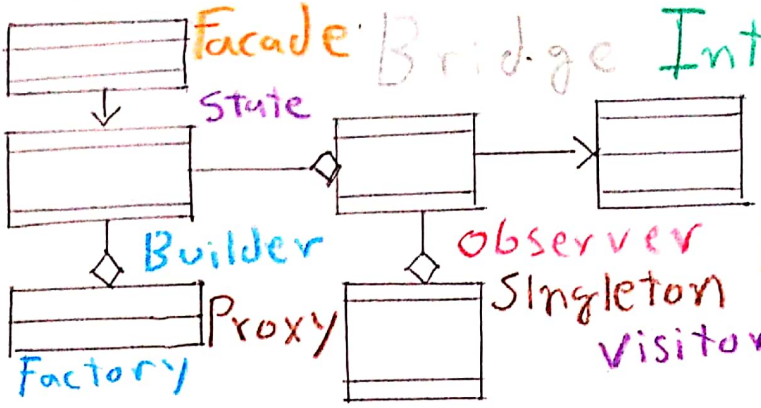
- Dependency inversion Principle (Principio de inversión de dependencia)

Establece que las dependencias, no en las concreciones. Es decir:

- Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
- Las abstracciones no deberían depender de detalles, debe ser lo contrario los detalles deberían depender de abstracciones.

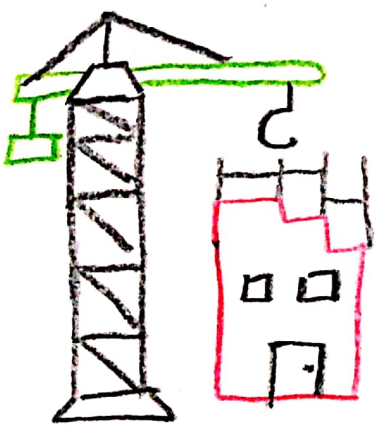
En algún momento nuestro Programa o aplicación llegará a estar formado por muchos módulos. Cuando esto pase es cuando debemos usar inyección de dependencias, lo que nos permitirá controlar las funcionalidades desde un sitio concreto en vez de estar esparcidas por todo el programa.

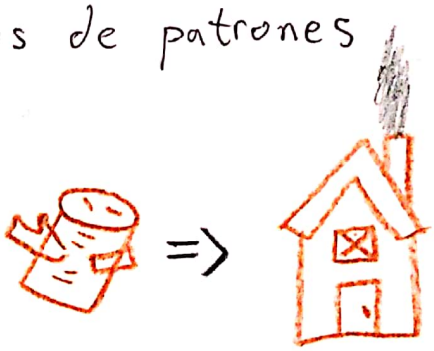
Patrones de Diseño



El diseño basado en patrones es un modelado que muestra diferentes formas de resolver un único problema, problemas recurrentes de diseño de software, que pueden ser reutilizados para otros mas, y que a su vez puede comportarse de distinta forma conforme pasa el tiempo y las relaciones entre cada etapa se ven afectados en el sentido que cambian por cómo están relacionados. Los caminos como resultado de dichos patrones pueden ser más cortos o más largos, a su vez más optimos o menos.

Tipos de patrones

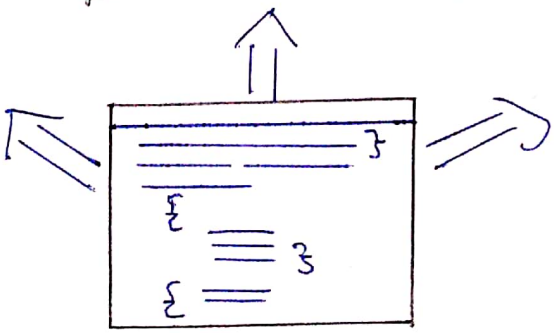




Creacionales

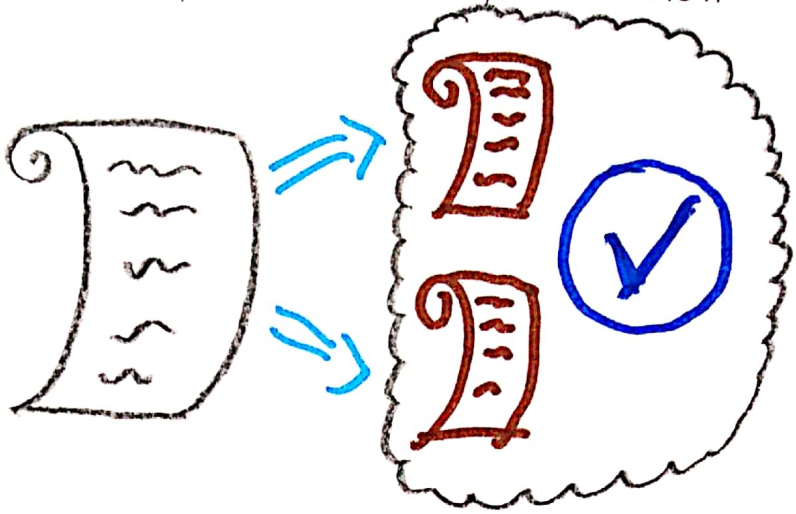
Estructurales

Comportamiento



Creacionales: Son los que facilitan la tarea de creación de nuevos objetos, de tal forma que el proceso de creación pueda ser desacoplado de la implementación del resto del sistema.

Los Patrones Creacionales facilitan la tarea de crear nuevos objetos encapsulando el proceso



Los más conocidos son:

Abstract Factory

Nos provee una interfaz que delega la creación de un conjunto de objetos relacionados sin necesidad de especificar las implementaciones concretas

Builder

Separa la creación de un objeto complejo de su estructura. Para crear representaciones diferentes

Prototype

Permite copiar objetos existentes sin hacer que su código dependa de sus clases.

Singleton

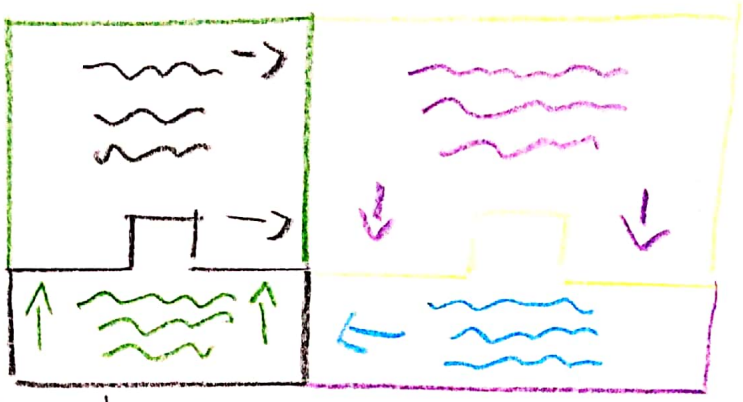
Este patrón de diseño restringe la creación de instancias de una clase a un único objeto.

Factory Method

Proporciona una interfaz para crear objetos en una superclase.

Estructurales: Son patrones que facilitan la modelación del software especificando como las clases se relacionan con otras.

Los patrones estructurales especifican la forma en que unas clases se relacionan con otras.



tipos de patron:

Adapter

Permite a dos clases con diferentes interfaces trabajar entre ellas

Bridge

Desacopla una abstracción de su implementación, para que las dos puedan evolucionar de forma independiente

Composite

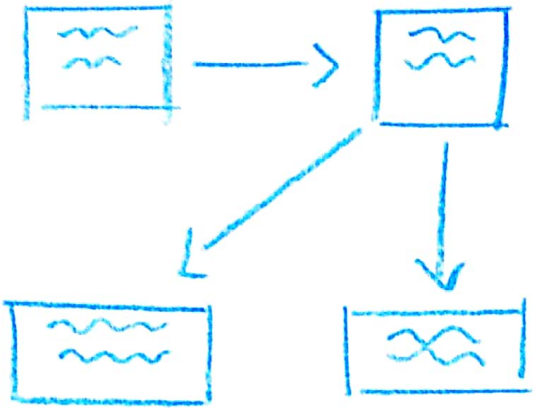
Facilita la creación de estructuras de objetos en árbol, donde todos los elementos emplean una interfaz

Decorator

Permite añadir funcionalidad extra a un objeto, de forma dinámica o estática.

Comportamiento: Se usan para gestionar algoritmos, relaciones y responsabilidades entre objetos. Se utilizan para detectar la presencia de patrones de comunicación ya presentes

Los patrones de comportamiento gestionan algoritmos, relaciones y responsabilidades entre objetos.



tipos de patrones:

Command:

Son objetos que encapsulan una acción y los parámetros que necesitan para ejecutarse

Iterator:

Se utiliza para poder movernos por los elementos de un conjunto de forma secuencial sin necesidad de exponer su implementación específica.

Interpreter

Define una representación para una gramática así como el mecanismo para evaluarla.

Mediator.

Objeto que encapsula como otro conjunto de objetos interactuan y se comunican entre si.

Ejemplo sencillo

Para este ejemplo haremos uso de un patrón de diseño creacional Singleton.

Singleton nos ayuda a que nuestros usuarios usen una misma instancia de un objeto, para utilizar la información o proceso del objeto, con ello ahorramos memoria porque evitamos que se cree un objeto por cada usuario y ese objeto es exactamente lo mismo para cada usuario, con ello ahorramos proceso por solamente usar uno.

Ejemplo de una conexión a una Base de Datos.

```java
package com.pruebaSingleton;

import com.pruebaSingleton.model.Conexion;

public class App {

	public static void main(String[] args) {
		//La construccion no se permite porque Conexion es una clase private, para ahorrar memoria al hacer las conexiones
		//Conexion c = new Conexion();
		Conexion c = Conexion.getInstancia();
		c.conectar();//Proceso para iniciar la conexion
		c.desconectar();//Proceso para cerrar la conexion

		boolean rpta = c instanceof Conexion;
		System.out.println(rpta);
		//Imprime el valor del proceso que se esta realizando, si es True la conexion se cierra.
	}


}
```

```java
package com.pruebaSingleton.model;

public class Conexion {


	private static Conexion instancia;
	//private static Conexion instancia = new Conexion();
	//Con el private evitamos que se cree una instancia por cada usuario.


	private Conexion() {

	}

	//Creacion de GetInstancia para uso de cualquier usuario

	public static Conexion getInstancia() {
		if(instancia == null) {
			instancia = new Conexion();
		}
		return instancia;
	}

	//Método de prueba de conexion
	public void conectar() {
		System.out.println("Me conecté a la BD");
	}

	//Método de prueba de desconexion
	public void desconectar() {
		System.out.println("Me desconecté de la BD");
	}

}
```