Principios S.O.L.I.D

SOLID resume buenas prácticas en el uso de las distintas técnicas que provee el paradigma, prácticas fundamentales para el buen diseño que propician la mantenibilidad, extensibilidad, adaptabilidad y escalabilidad del código, y sobre todo la salud de los programadores que tendrán que leer y/o modificar nuestro código.

SOLID, por sus siglas en inglés, significa:

S-Single Responsability:
Este principio trata de destinar cada clase a una finalidad sencilla y concreta. En muchas ocasiones estamos tentados a poner un método reutilizable que no tienen nada que ver con la clase simplemente porque lo utiliza. En ese momento pensamos "Ya que estamos aquí, para que voy a crear una clase para realizar esto. Directamente lo pongo aquí".
El problema surge cuando tenemos la necesidad de utilizar ese mismo método desde otra clase. Si no se refactoriza en ese momento y se crea una clase destinada para la finalidad del método, nos encontraremos con que a largo plazo con que las clases realizan tareas que no deberían ser de su responsabilidad.
Con la anterior mentalidad nos encontraremos, por ejemplo, con un algoritmo de formateo de números en una clase destinada a leer de la base de datos porque fue el primer sitio donde se empezó a utilizar. Esto conlleva a tener métodos difíciles de detectar y encontrar, de manera que el código hay que tenerlo memorizado en la cabeza.

Ej:

Un CLIENTE desea saber si tiene dinero en el banco para realizar el pago de sus deudas.

La siguiente clase es la solución inicial al problema:

```
public class Cliente {
	public boolean PagarDeudas()	{
		Boolean Deuda;
		Deuda = false;
		if (TieneDineroEnCuenta())		{
			Deuda = true;
		}
		return Deuda;
	}

	//Funcion que valida si tiene dinero en la cuenta
	public boolean TieneDineroEnCuenta()	{
		return true;
	}
}
```

Aunque este código funciona, tiene como problema que se han combinado 2 responsabilidades y esto va a ser más complejo de mantener/escalar. El verificar si tiene el dinero suficiente en el banco es otra responsabilidad, no le pertenece directamente al usuario porque posiblemente queramos usar esa lógica en otro lugar y vamos a tener que copiar/pegar código.
Teniendo en cuenta el principio de responsabilidad única se puede realizar una refactorización del código y dividirlo en dos clases como se ve a continuación:

```
public class Cliente {

	Cuenta cuenta = new Cuenta();

	public boolean pagardeudas()
	{
		Boolean Deuda;
		Deuda = false;

		if (cuenta.tieneDineroEnCuenta())
		{
			Deuda = true;
		}
		return Deuda;
	}

}

public class Cuenta
{
	public boolean tieneDineroEnCuenta()
	{
		return true;
	}
}
```

O-Open Closed:
Principio que habla de crear clases extensibles sin necesidad de entrar al código fuente a modificarlo. Es decir, el diseño debe ser abierto para poderse extender pero cerrado para poderse modificar. Aunque dicho parece fácil, lo complicado es predecir por donde se debe extender y que no tengamos que modificarlo. Para conseguir este principio hay que tener muy claro cómo va a funcionar la aplicación, por donde se puede extender y cómo van a interactuar las clases.

El uso más común de extensión es mediante la herencia y la implementación de métodos. Existe otra alternativa que consiste en utilizar métodos que acepten una interfaz de manera que podemos ejecutar cualquier clase que implemente esa interfaz. En todos los casos, el comportamiento de la clase cambia sin que hayamos tenido que tocar código interno.

Ej:

Ver los permisos que puede tener un cliente dentro del sistema, uno de ellos es "Puede realizar un pago" basado en el Rol.
Una posible solución al ejemplo sería crear las clases Admin, Operador y Pagos, como se observa a continuación.

```
public class Admin {
            public Integer Id;
            public String Name;
}

public class Operador {
            public Integer Id;
            public String Name;
}

public class Pagos {

        public boolean pagooperador(Operador user)
   {
      // Lógica de negocio
      return true;
   }

   public boolean pagoadmin(Admin user)
   {
      // Lógica de negocio
      return true;
   }

}
```

El problema con esta solución radica en futuros cambios, es decir, el día de mañana se solicita que se cree un nuevo rol, por ejemplo “vendedor". La solución actual sería crear un método más pagovendedor. Pero el problema radica en que la clase tiene que implementar más código.

Una solución alterna sería crear una clase persona de la cual heredan cada uno de los usuarios que se vayan solicitando, además de tener solo un método de pagos donde se reciba un objeto tipo persona. Esto garantiza que la clase de pagos no será necesario cambiarla ante nuevos requerimientos pero si se puede utilizar o extender para cualquier otro tipo de usuario que se desee crear.

Las clases de la solución después de aplicar la refactorización son las siguientes:

```
public class Persona {
        public Integer Id;
        public String Name;
}

public class Admin extends Persona {

}

public class Operador extends Persona{

}

public class Vendedor extends Persona{

}

public class Pagos {

        public boolean pago(Persona user)
   {
      // Lógica de negocio
      return true;
   }

}
```

L-Liskov Substitution:

El principio de sustitución de Liskov nos dice 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. Esto nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento de la padre.

Ej:

Se tiene una clase base abstracta que será un Robot y este puede realizar una cantidad X de punto de daño al momento de atacar, pero necesita ser heredada par ser usada.

```
abstract class Robot {

	public abstract Integer atacar();

}
```

Asimismo, la tecnología ha avanzado y llegaron 2 modelos nuevos que van a heredar las características del Robot. La diferencia es que estos tienen más poder de ataque, por lo tanto se debe sobrescribir sus métodos.

```
public class Robot2018 extends Robot{

	@Override
	public Integer atacar() {
		return 150;
	}	}
public class Robot2020 extends Robot{

	@Override
	public Integer atacar() {
		return 300;
	}

}
```

Si se organiza una batalla de robots se tendría un código como se muestra a continuación:

```
public class Pelea {

	Robot2018 robot1 = new Robot2018();
	Robot robot2 = new Robot2018();



	Robot2020 robot3 = new Robot2020();
	Robot robot4 = new Robot2020();

	public void xxx() {
		robot1.atacar();//Daña 150
		robot2.atacar();//Daña 150
```

```
            robot3.atacar();//Daña 300
            robot4.atacar();//Daña 300
        }

}
```

Se puede observar como los objetos robot1 y robot2 son instanciados de la misma manera a pesar de ser objetos diferentes, además tanto robot1 como robot2 pueden atacar. La aplicación de la sustitución de Liskov se ve reflejada en el robot2 puesto que reemplaza un objeto Robot2018 sin afectar el código.

I-Interface Segregation:
Cuando se definen interfaces estas deben ser específicas a una finalidad concreta. Por ello, si tenemos que definir una serie de métodos que debe utilizar una clase a través de interfaces, es preferible tener muchas interfaces que definen pocos métodos que tener una interface con muchos métodos.

Ej:

Se tiene la siguiente interfaz que se utiliza para determinar los tiempos que se pueden demorar comiendo y trabajando.

```
Interface Worker() {
        int work();
        int eat();
}
```

La cual es implementada en las siguientes clases:

```
Class ManWorker implements Worker {
        int work(){ return 30000;}
        int eat(){return 8000;}
}
Class RobotWorker implements Worker {
        int work(){ return 30000;}
        int eat(){//No aplica para un robot}
}
```

El problema es que todos implementan la misma interfaz y realmente todos los trabajadores no pueden realizar todas las actividades. Lo mejor que se puede hacer es dividir la interfaz, en mini interfaces de esta manera se puede asignar comportamiento a quien queramos.

Al aplicar la refactorización del código se tiene la siguiente solución:

```
Interface Workable() {
        int work();
}

Interface Feedable() {
        int eat();
}

Class ManWorker implements Workable, Feedable {
        int work(){ return 30000;}
        int eat(){return 8000;}
}

Class RobotWorker implements Worker {
        int work(){ return 30000;}
}
```

Se nota que la clase RobotWorker no se ve obligado a implementar métodos que no necesita.

D-Dependency Inversion:
El objetivo de este principio es poder 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. Todos estos aspectos se especificarán mediante interfaces, y el núcleo no tendrá que conocer cuál es la implementación real para funcionar.

Ej:

Se tiene una cesta de la compra(ShoppingBasket) que lo que hace es almacenar la información y llamar al método de pago para que ejecute la operación. El código sería algo así:

```
public class ShoppingBasket {

        public void buy(Shopping shopping) {

                SqlDatabase db = new SqlDatabase(); db.save(shopping);
                CreditCard creditCard = new CreditCard();       creditCard.pay(shopping);
        }
```

```
public class SqlDatabase {

	public void save(Shopping shopping){
		// Saves data in SQL database
	}

}

public class CreditCard {

	public void pay(Shopping shopping){
		// Performs payment using a credit card
	 }

}
```

Con esta implementación se está violando el principio de inversión de dependencias. Una clase de más alto nivel, como es la cesta de la compra, está dependiendo de otras de alto nivel, como cuál es el mecanismo para almacenar la información o para realizar el método de pago. Se encarga de crear instancias de esos objetos y después utilizarlas.
Además si se quiere añadir métodos de pago, o enviar la información a un servidor en vez de guardarla en una base de datos local, no hay forma de hacerlo sin desmontar toda la lógica.
El primer paso para solucionar el problema es dejar de depender de concreciones, como las clases CreditCard y SqlDatabasese. Una solución sería crear interfaces que definen el comportamiento que debe dar una clase para poder funcionar como mecanismo de persistencia o como método de pago. Después de la refactorización el resultado es el siguiente:

```
public interface Persistence {
   void save(Shopping shopping);
}

public class SqlDatabase implements Persistence {

	@Override
	public void save(Shopping shopping){
	// Saves data in SQL database
	}

}

public interface PaymentMethod {
```

```
	void pay(Shopping shopping);

}

public class CreditCard implements PaymentMethod {

	@Override
	public void pay(Shopping shopping){
		// Performs payment using a credit card
	}

}
```

Ahora ya no depende de la implementación particular que se decide, sin embargo aún se debe seguir instanciando en ShoppingBasket. Por tal motivo el siguiente paso es invertir las dependencias. En este caso se hará mediante el constructor pero se puede realizar de otras maneras como utilizando métodos setter. Después de realizar la refactorización se tiene el siguiente resultado:

```
public class ShoppingBasket {

	private final Persistence persistence;
	private final PaymentMethod paymentMethod;

	public ShoppingBasket(Persistence persistence, PaymentMethod paymentMethod) {
		this.persistence = persistence;
		this.paymentMethod = paymentMethod;

	}

	public void buy(Shopping shopping) {
		persistence.save(shopping);
		paymentMethod.pay(shopping);
	}

}
```

Si se quisiera pagar con Paypal y almacenar los cambios en el servidor, nos bastaría con crear unas clases que implementen sus respectivas interfaces.

```
public class Server implements Persistence {

   @Override
```

```
    public void save(Shopping shopping) {
        // Saves data in a server
    }
}

public class Paypal implements PaymentMethod {

    @Override
    public void pay(Shopping shopping) {
        // Performs payment using Paypal account
    }
}
```

Nótese que no hubo necesidad de cambiar la lógica del dominio.