Principios S.O.L.I.D.

Single Responsibility (Unica Responsabilidad)

Mantener la cohesión, cada clase, método debería tener una sola razón de existir (o de cambio). Se debe reunir las cosas que cambian por las mismas razones, y separar aquellas que cambian por razones diferentes. En el momento en el que una clase adquiere más responsabilidad pasa a estar acoplada.

Mal: En esta imagen se puede ver como esta clase tiene dos razones para cambiar. La clase Coche permite tanto el acceso a las propiedades de la clase como a realizar operaciones sobre la DB, por lo que la clase ya tiene más de una responsabilidad.

```
class Coche {
    String marca;

    Coche(String marca){ this.marca = marca; }

    String getMarcaCoche(){ return marca; }

    void guardarCocheDB(Coche coche){ ... }
}
```

Bien

```
class Coche {
    String marca;

    Coche(String marca){ this.marca = marca; }

    String getMarcaCoche(){ return marca; }
}

class CocheDB{
    void guardarCocheDB(Coche coche){ ... }
    void eliminarCocheDB(Coche coche){ ... }
}
```

Open – Close (Abierto – Cerrado)

En otras palabras: las clases que usas deberían estar abiertas para poder extenderse y cerradas para modificarse.Es decir, que se debe poder agregar comportamiento si tener que modificar código existente.

Teniendo:

```java
class Coche {
    String marca;

    Coche(String marca){ this.marca = marca; }

    String getMarcaCoche(){ return marca; }
}
```

```java
public static void main(String[] args) {
    Coche[] arrayCoches = {
            new Coche("Renault"),
            new Coche("Audi")
    };
    imprimirPrecioMedioCoche(arrayCoches);
}

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        if(coche.marca.equals("Renault")) System.out.println(18000);
        if(coche.marca.equals("Audi")) System.out.println(25000);
    }
}
```

En este caso no se cumpliria el principio Open-Closed por el hecho de que si quisieramos añadir una nueva marca, por ejemplo agregando la marca "Mercedes" quedaria de la siguiente forma:

```java
public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        if(coche.marca.equals("Renault")) System.out.println(18000);
        if(coche.marca.equals("Audi")) System.out.println(25000);
        if(coche.marca.equals("Mercedes")) System.out.println(27000);
    }
}
```

Una posible solución podría ser lo siguiente:

```
abstract class Coche {
    // ...
    abstract int precioMedioCoche();
}

class Renault extends Coche {
    @Override
    int precioMedioCoche() { return 18000; }
}

class Audi extends Coche {
    @Override
    int precioMedioCoche() { return 25000; }
}

class Mercedes extends Coche {
    @Override
    int precioMedioCoche() { return 27000; }
}
```

```
public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        System.out.println(coche.precioMedioCoche());
    }
}
```

Liskov Sustitution (Sustitución)

Significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.

Interface Segregation (Segregación de la interfaz)

Hacer interfaces que sean específicas para un tipo de cliente. Es preferible contar con muchas interfaces que definan pocos métodos que tener una interface forzada a implementar muchos métodos a los que no dará uso.. Es decir, los clientes no deberían ser forzados a usar interfaces que no usan.

Dependency Inversion (inversión de dependencia)

Los módulos de alto nivel no deberían depender de módulos de bajo nivel, ambos deberían depender de abstracciones. Las abstracciones no deberían depender de los detalles, si no al revés. El objetivo de este es alcanzar un bajo acoplamiento.Es decir, es mejor depender de abstracciones, como una interfaz, o una clase abstracta, en vez de clases concretas.