# PRINCIPIOS DE DISEÑO

# PRINCIPIO ≠ PATRÓN

---

**¿En qué favorece cumplir los principios de diseño?**

Reutilización del código

Mantenimiento

Escalabilidad

**¿Cómo lo solucionamos?**

Creando otra clase que asuma esa responsabilidad

```
class Coche {
    String marca;

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

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

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

# SOLID

## SINGLE RESPONSIBILITY

Una clase, componente o microservicio debe ser responsable de una sola cosa.

Una clase, componente o microservicio solo debería tener una razón para cambiar

¿Qué ocurre en esta clase coche?

```
class Coche {
    String marca;

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

    String getMarcaCoche(){ return marca; }

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

## ¿Cómo lo solucionamos?

Haciendo que coche sea una clase abstracta y crear una subclase por marca

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

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

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

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

# S O L I D

## O PEN-CLOSED

Establece que las entidades software (clases, módulos y funciones) deberían estar abiertas para su extensión, pero cerradas para su modificación.

```
class Coche {
    String marca;

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

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

¿Qué ocurre en este método?

```
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);
    }
}
```

¿Cómo lo solucionamos?

Declarando un método abstracto en la clase Coche e implementándolo en cada subclase posteriormente

```
abstract class Coche {

    // ...
    abstract int numAsientos();
}
```

```
class Renault extends Coche {

    // ...
    @Override
    int numAsientos() {
        return 5;
    }
}
```

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

# S O L I D

## LISKOV SUBSTITUTION

Declara que una superclase debe ser sustituible por su subclase, y si al hacer esto, el programa falla, estaremos violando este principio

¿Qué ocurre en este método?

```
Coche[] arrayCoches = {
        new Renault(),
        new Audi(),
        new Mercedes(),
        new Ford()
};
```

```
public static void imprimirNumAsientos(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        if(coche instanceof Renault)
            System.out.println(numAsientosRenault(coche));
        if(coche instanceof Audi)
            System.out.println(numAsientosAudi(coche));
        if(coche instanceof Mercedes)
            System.out.println(numAsientosMercedes(coche));
        if(coche instanceof Ford)
            System.out.println(numAsientosFord(coche));
    }
}
```

## ¿Cómo lo solucionamos?

Segregando la interfaz IAve en varias. De esta manera cada tipo de ave solo implementará las que necesite. (Caso: Loro, pingüino, etc..)

```
interface IAve {
    void comer();
}
interface IAveVoladora {
    void volar();
}

interface IAveNadadora {
    void nadar();
}
```

```
class Pinguino implements IAve, IAveNadadora{

    @Override
    public void nadar() {
        //...
    }

    @Override
    public void comer() {
        //...
    }
}
```

# S O L I D

## INTERFAZ SEGREGATION

Una clase no puede verse forzada a depender de interfaces que no usa.

Se quiere añadir pingüino y se necesita el método nadar
¿Qué ocurre con el método nadar en la clase Loro?

```
interface IAve {
    void volar();
    void comer();
}
```

→

```
interface IAve {
    void volar();
    void comer();
    void nadar();
}
```

```
class Loro implements IAve{

    @Override
    public void volar() {
        //...
    }

    @Override
    public void comer() {
        //..
    }
}
```

## ¿Cómo lo solucionamos?

Creando una interfaz que se llame conexión.
La clase AccesoDatos declarará una interfaz Conexión en vez de una base de datos específica.

```
interface Conexion {
    Dato getDatos();
    void setDatos();
}

class AccesoADatos {

    private Conexion conexion;

    public AccesoADatos(Conexion conexion){
        this.conexion = conexion;
    }

    Dato getDatos(){
        conexion.getDatos();
    }
}
```

```
class DatabaseService implements Conexion {

    @Override
    public Dato getDatos() { //... }

    @Override
    public void setDatos() { //... }
}

class APIService implements Conexion{

    @Override
    public Dato getDatos() { //... }

    @Override
    public void setDatos() { //... }
}
```

# S O L I D

## D EPENDENCY INVERSION

Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Las abstracciones no deberían depender de detalles

¿Qué ocurre si mañana cambiamos de BBDD?

```
class DatabaseService{
    //...
    void getDatos(){ //... }
}
```

```
class AccesoADatos {

    private DatabaseService databaseService;

    public AccesoADatos(DatabaseService databaseService){
        this.databaseService = databaseService;
    }

    Dato getDatos(){
        databaseService.getDatos();
        //...
    }
}
```

# OTROS PRINCIPIOS DE DISEÑO

## DRY
(Don’t repeat yourself)


## LAW OF DEMETER
(No hables con extraños)