

Tema 3

# Diseño de Software





## Principios SOLID



# Problemas

- ❖ Rigidez
- ❖ Fragilidad
- ❖ Inmovilidad
- ❖ Viscosidad
- ❖ Complejidad innecesaria
- ❖ Repetición innecesaria
- ❖ Opacidad



# Enfoque Agil/OOD

## ❖ Qué asumimos:

- La única constante en el software es el cambio en los requerimientos

## ❖ Qué hacemos:

- Detectar el problema siguiendo las metodologías ágiles
- Diagnosticar aplicando principios de diseño (OOD)
- Resolver mejorando el diseño, usando frecuentemente patrones como referencia



# Principios SOLID

- ❖ Responsabilidad única
- ❖ Abierto-Cerrado
- ❖ Substitución de Liskov
- ❖ Inversión de dependencia
- ❖ Segregación de Interfaz



# Patrones y Principios SOLID

- ❖ Los patrones de diseño (GoF) muestran, en alguna medida, uno o mas principios SOLID, aunque...
- ❖ no hay relación directa entre los patrones de diseño (GoF) y los principios SOLID. Estos últimos son atributos que indican si un diseño es mejor o peor.
- ❖ Estos principios definen lineamientos, no son reglas estrictas. Hay que comprender su motivación y aplicarlos con criterio.



# Responsabilidad única

*Una clase debe tener una  
única razón para ser cambiada.*

No es cierto que un elevado numero de clases pequeñas (o métodos pequeños) sea mas difícil de entender. Siempre todo ese código estará ahí.



# Responsabilidad única



## Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

<http://blogs.msdn.com/b/cdn devs/archive/2009/07/15/the-solid-principles-explained-with-motivational-posters.aspx>





```
class Coche {  
    String marca;  
  
    Coche(String marca){ this.marca = marca; }  
  
    String getMarcaCoche(){ return marca; }  
  
    void guardarCocheDB(Coche coche){ ... }  
}
```



```
class Coche {  
    String marca;  
  
    Coche(String marca){ this.marca = marca; }  
  
    String getMarcaCoche(){ return marca; }  
}  
  
class CocheDB{  
    void guardarCocheDB(Coche coche){ ... }  
    void eliminarCocheDB(Coche coche){ ... }  
}
```



# Abierto-Cerrado

*Las entidades de software (clases, módulos, funciones, etc) deben estar abiertas a extensión, pero cerradas a modificación.*

Acercarse a un ideal

Los cambios deben generar código **nuevo**,  
no modificar el código **viejo**.



# Abierto-Cerrado



## Open-Closed Principle

Open-chest surgery isn't needed when putting on a coat.

<http://blogs.msdn.com/b/cdn devs/archive/2009/07/15/the-solid-principles-explained-with-motivational-posters.aspx>



```
class Coche {  
    String marca;  
  
    Coche(String marca){ this.marca = marca; }  
  
    String getMarcaCoche(){ return marca; }  
}
```

Si quisiéramos iterar a través de una lista de coches e imprimir sus marcas por precio promedio

```
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);  
    }  
}
```



Esto no cumpliría el principio abierto/cerrado, ya que si decidimos añadir un nuevo coche de otra marca:

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

También tendríamos que modificar el método que hemos creado anteriormente:

```
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);  
    }  
}
```



```
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 main(String[] args) {

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

    imprimirPrecioMedioCoche(arrayCoches);
}

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



# Substitución de Liskov

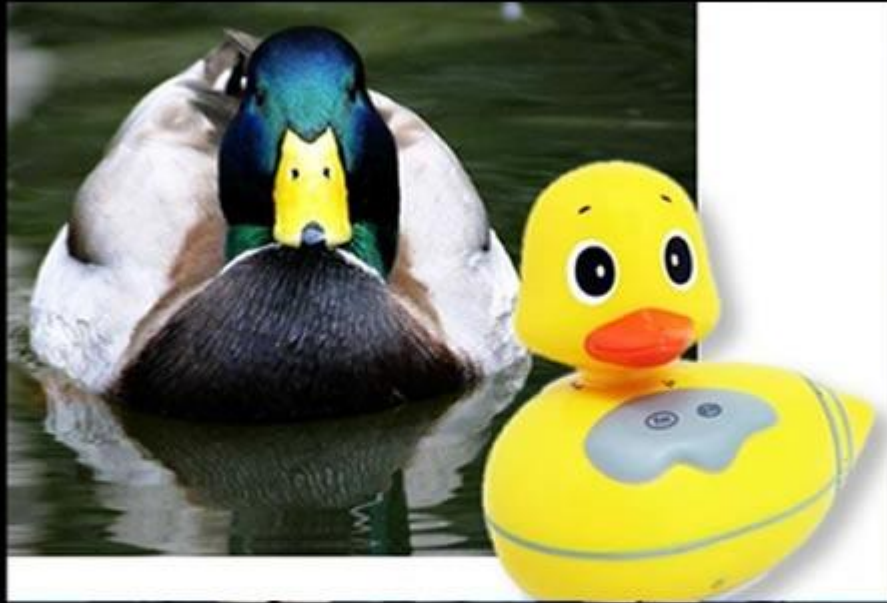
*Los subtipos deben ser sustituibles por sus tipos base.*

Es la base de poder del polimorfismo.

Cuidarse de `typeof()` y otros datos de tipo en runtime.



# Substitución de Liskov



## Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

<http://blogs.msdn.com/b/cdn devs/archive/2009/07/15/the-solid-principles-explained-with-motivational-posters.aspx>





- ❖ 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.



# Inversión de dependencia



## Dependency Inversion Principle

Would you solder a lamp directly  
to the electrical wiring in a wall?

<http://blogs.msdn.com/b/cdn devs/archive/2009/07/15/the-solid-principles-explained-with-motivational-posters.aspx>



# Inversión de dependencia

- a) Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.*
- b) Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.*

Es el principio general detrás del concepto de Layers o Capas.



# Segregación de Interfaz



## Interface Segregation Principle

You want me to plug this in *where*?

<http://blogs.msdn.com/b/cdn devs/archive/2009/07/15/the-solid-principles-explained-with-motivational-posters.aspx>



# Segregación de Interfaz

*Los clientes no deben ser forzados a depender de métodos que no usan.*

- ❖ Apunta a evitar las interfaces “gordas”.
- ❖ No importa la cantidad de métodos, sino que todos sus clientes las utilicen.
- ❖ Inadvertidamente podemos **acoplar clientes** que usan ciertos métodos con **otros clientes** que no los usan.



```
interface IAve {  
    void comer();  
}  
interface IAveVoladora {  
    void volar();  
}  
interface IAveNadadora {  
    void nadar();  
}  
class Loro implements IAve, IAveVoladora{  
    @Override  
    public void volar() {  
        //...  
    }  
    @Override  
    public void comer() {  
        //...  
    }  
}  
class Pinguino implements IAve, IAveNadadora{  
    @Override  
    public void nadar() {  
        //...  
    }  
    @Override  
    public void comer() {  
        //...  
    }  
}
```