Principio de segregación de interfaces

Recuerda que una forma, entre otras, de detectar fallos de diseño en nuestras clases es comprobar si cumplen los 5 principios SOLID :

- S – Single Responsibility Principle (SRP)
- O – Open/Closed Principle (OCP)
- L – Liskov Substitution Principle (LSP)
- **I – Interface Segregation Principle (ISP)**
- D – Dependency Inversion Principle (DIP)

Uno de los principios es *El principio de segregación de interfaces*

Este principio nos recuerda que **ninguna clase debería depender de métodos que no usa**. Por tanto, cuando creemos interfaces que definan comportamientos, es importante estar seguros de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. **En caso contrario, es mejor tener varias interfaces más pequeñas**.

Se viola este principio cuando escribimos interfaces que intentan definir más cosas de las debidas, lo que se denominan *fat interfaces* y probablemente ocurrirá que **las clases hijas acabarán por no usar muchos de esos métodos**, pero a pesar de todo habrá que darles una implementación que serán del tipo lanzar una excepción o simplemente no hacer nada.

¿Cómo detectar que estamos violando el Principio de segregación de interfaces?

Si al implementar una interfaz se observa que uno o varios de los métodos no tienen sentido y tenemos que dejarlos vacíos o lanzar excepciones, es muy probable que estemos violando este principio.
La solución: Dividir la interface original en varias interfaces que definan comportamientos más específicos.

Ejemplo:
Escribimos en un interface el comportamiento de los vehículos. Considero que en principio los vehículos deben de poder moverse arrancando y parando y además algunos como un avión puede volar.

```
interface Vehiculo{
   void start();
   void stop();
   void fly();
}
```

Ahora quiero modelar un Coche, que es un vehículo y por tanto quiero que implemente el comportamiento del vehículo

```
interface Vehiculo{
   void start();
   void stop();
   void fly();
}

class Coche implements Vehiculo{
   String modelo;
   double precio;

   @Override
   public void start(){System.out.println("arrancando coche ...");}
   @Override
   public void stop(){System.out.println("arrancando coche ...");}
   @Override
   public void fly(){
      //?????????
   }
}
```

¡¡los coches no vuelan!!
Tuve que crear un obligatoriamente un método fly() vacío

Sí aplicamos el principio de segregación de interfaces mejoramos el diseño simplemente dividiendo el interface anterior en dos

```
interface Movable{
   void start();
   void stop();

}
interface Flyable{
   void fly();
}

class Coche implements Movable{
   String modelo;
   double precio;

   @Override
   public void start(){System.out.println("arrancando coche ...");}
   @Override
   public void stop(){System.out.println("parando  coche ...");}

}

class Avion implements Movable,Flyable{
```

```
    String modelo;
    double precio;

    @Override
    public void start(){System.out.println("arrancando avión ...");}
    @Override
    public void stop(){System.out.println("parando  avión ...");}
    @Override
    public void fly() {
        System.out.println("avión volando  ...");

    }

}
```

Ejercicio:

En el siguiente ejemplo 3 reproductores implementan MediaPlayer. Todos los reproductores son reproductores de audio y video excepto WinampMediaPlayer que sólo reproduce audio pero que se ve obligada a una sobrescritura poco natural del método playVideo(). Mejora el diseño segregando el interface MediaPlayer.

```
interface MediaPlayer {
    public void playAudio();
    public void playVideo();
}
class DivMediaPlayer implements MediaPlayer {
    @Override
    public void playAudio() {
        System.out.println(" Playing audio ..........");
    }

    @Override
    public void playVideo() {
        System.out.println(" Playing video ..........");
    }
}
class VlcMediaPlayer implements MediaPlayer {
    @Override
    public void playAudio() {
        System.out.println(" Playing audio ..........");

    }

    @Override
    public void playVideo() {
        System.out.println(" Playing video ..........");

    }
}
class WinampMediaPlayer implements MediaPlayer {

    // Play video is not supported in Winamp player
    public void playVideo() {
      System.out.println(" lo siento, no sé reproducir vídeo.");

    }

    @Override
    public void playAudio() {
```

```
        System.out.println("Playing audio .....");

    }
}
```