**1.3. Liskov Substitution Principle:**

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.

Este principio viene a desmentir la idea preconcebida de que las clases son una forma directa de modelar la realidad, y que hay que tener cuidado con esa modelización.

La primera en hablar de él fue [Bárbara Liskov](https://en.wikipedia.org/wiki/Barbara_Liskov) (de ahí el nombre), una reconocida ingeniera de software americana.

### ¿Cómo detectar que estamos violando el principio de sustitución de Liskov?

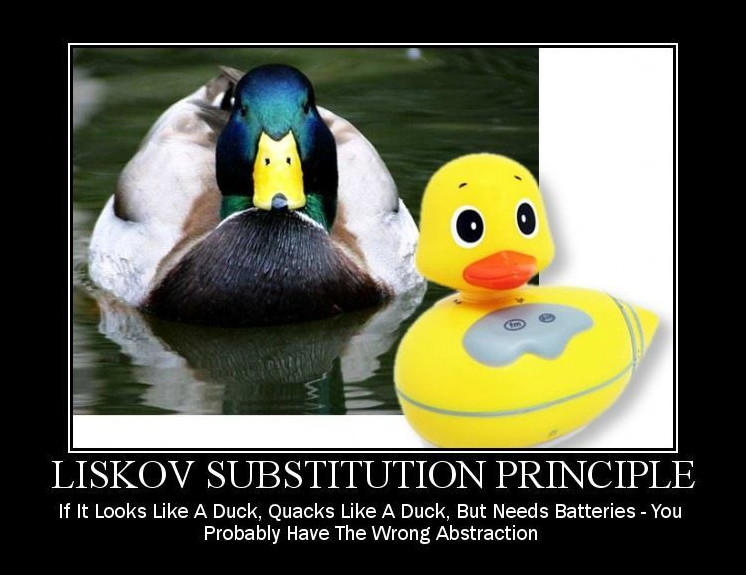
Seguro que te has encontrado con esta situación muchas veces: creas una clase que extiende de otra, pero de repente uno de los métodos te sobra, y no sabes que hacer con él.

Las opciones más rápidas son bien dejarlo vacío, bien lanzar una excepción cuando se use, asegurándote de que nadie llama incorrectamente a un método que no se puede utilizar.

**Si un método sobrescrito no hace nada o lanza una excepción, es muy probable que estés violando el principio de sustitución de Liskov**.

Si tu código estaba usando un método que para algunas concreciones ahora lanza una excepción, ¿cómo puedes estar seguro de que todo sigue funcionando?

Imagen de [Derick Balley](https://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/)



Otra herramienta que te avisará fácilmente son los tests. Si los tests de la clase padre no funcionan para la hija, también estarás violando este principio.

Veremos un ejemplo con el primer caso.

**- Ejemplo:** En la vida real tenemos claro que un elefante es un animal. Imaginemos que tenemos la clase Animal que representa un animal, y les damos a los animales la propiedad de andar y saltar:

open **class** Animal {

open **fun** walk() { ... }

open **fun** jump() { ... }

}

Y tenemos una parte del código donde recibimos un animal, y necesitamos que el animal salte:

**fun** jumpHole(a: Animal){

a.walk()

a.jump()

a.walk()

}

Ahora nos creamos un elefante. Pero claro, un elefante no puede saltar, así que decidimos lanzar una excepción para asegurarnos de detectarlos si esto ocurre:

**class** Elephant : Animal() {

override **fun** jump() =

**throw** Exception("Los elefantes no pueden saltar")

}

Ahora en todos los sitios donde estemos usando jumpHole(), si el animal es un elefante, tendremos una excepción.

Mal asunto, ¿no?

### ¿Cómo lo solucionamos?

Aquí lo que tenemos que entender es que la abstracción que hemos decidido hacer no es correcta.

Hay animales que no saltan, así que estamos dando por ciertos casos que se pueden volver en nuestra contra.

Por tanto, las clases tienen que representar esos posibles estados inequívocamente, y las funciones usar las abstracciones que necesiten.

¿Qué podríamos hacer en este caso? Plantear un tipo de animal ligero que sí que puede saltar, mientras que damos por hecho que los animales en general no pueden hacerlo:

open **class** Animal {

open **fun** walk() { }

}

open **class** LightweightAnimal : Animal() {

open **fun** jump() { }

}

Esto nos permite definir animales que sí pueden saltar y otros que no. Por ejemplo un perro y un elefante:

**class** Dog: LightweightAnimal()

**class** Elephant: Animal()

Y la función de jumpHole() solo admitiría animales que pueden saltar:

**fun** jumpHole(a: LightweightAnimal){

a.walk()

a.jump()

a.walk()

}

Elegir las abstracciones correctas muchas veces no es fácil, pero tenemos que intentar limitar al máximo cuál es su alcance para no pedir más de lo que se necesita ni menos.

Esta es la solución que obtendríamos mediante herencia aplicando el Principio de Liskov, pero también se podría haber solucionado mediante composición.

La herencia nos puede generar una jerarquía de clases muy compleja si hay muchos tipos de animales, así que en función del problema hay que plantearse cuál merece la pena usar.

Esta segunda opción es la que veremos con el [Principio de segregación de interfaces](https://devexperto.com/principio-de-segregacion-de-interfaces/)

**- Conclusión:**

El principio de Liskov **nos ayuda a utilizar la herencia de forma correcta**, y a tener mucho más cuidado a la hora de extender clases.

En la práctica nos ahorrará muchos errores derivados de nuestro afán por modelar lo que vemos en la vida real en clases siguiendo la misma lógica.

No siempre hay una modelización exacta, por lo que este principio nos ayudará a descubrir la mejor forma de hacerlo.