El Principio de Inversión de Dependencias

El Principio de Inversión de Dependencias, también conocido como Dependency Inversion Principle o DIP es el último de los cinco Principios SOLID de la programación orientada a objetos.

La definición original: Los módulos de alto nivel no deberían depender de los módulos de bajo nivel.

Un módulo es una clase o bien un paquete (conjunto de clases). Alto/bajo nivel no es fácil de explicar. Normalmente algo de alto nivel se refiere a aquellas clases que para nosotros son especialmente importantes y queremos escribirlas de forma más genérica para que sean reusables y fáciles de mantener, por ejemplo, la clase que contiene la lógica del juego de siete y media o tres en raya la consideramos de alto nivel y la clase que gestiona la entrada salida de bajo nivel.

Ahora, podemos concretar un poco más la definición original de la siguiente forma:

- Ambos módulos deberían depender de abstracciones.
- Las abstracciones no deberían depender de los detalles. Los detalles deben depender de abstracciones.

Sigue siendo todo muy genérico así que usaremos un ejemplo para concretar lo que quiere decir lo anterior.

Ejemplo

Pensemos en los típicos mandos a distancia que encienden y apagan inalámbricamente dispositivos**.** Por tanto, queremos objetos Mando que se usan como conmutadores para encender y apagar otros dispositivos. Inicialmente los mandos sólo se utilizan para encender/apagar objetos lámparas y se hizo el diseño abajo descrito en uml. Los objetos Lampara reciben mensajes encender() y apagar() de objetos Mando. Así, a un mando se le indica una lámpara y es capaz de encenderla y apagarla. Hacemos el siguiente diseño en el que un Mando usa una Lampara:

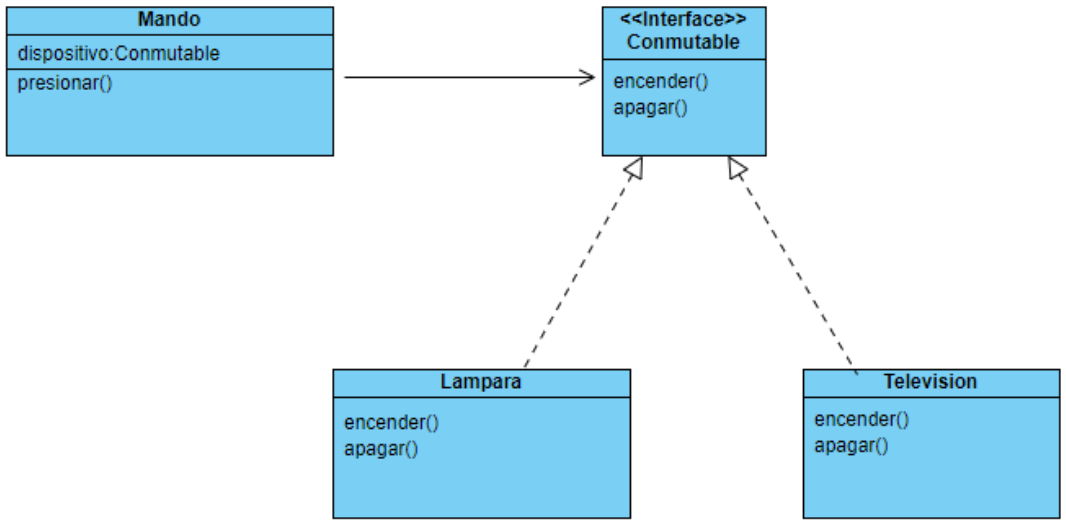
Problemas del diseño:

- Mando depende directamente de Lampara: Si Lampara cambia, Mando cambiará también.

- Mando no es reusable: No podrías usar presionar() para encender una Lavadora o una tv por ejemplo.

Si en nuestra app resulta que Mando “es importante” querremos hacerlo de más alto nivel para poder rehusarlo con cualquier aparato electrico, no solo lamparas, entonces considaramos importantes los problemas anteriores, y por tanto al considerar Mando una clase de *alto nivel* muy importante, el diseño anterior es una violación al principio de Inversión de Dependencias

Solución
La solución pasa por crear una capa intermedia para evitar la conexión directa entre Mando y Lampara para lo que definiremos una interfaz abstracta asociada a Mando e implementada por cualquier clase como Lampara, Televisión, Lavadora, ….



De este modo:
- Mando depende únicamente de abstracciones, puede ser reusado en varias clases que implementen Conmutable. Por ejemplo, Televisión, Lavadora, Ventilador …
- Cambios en Lampara no afectarán a Mando ya que Mando solo usará Conmutable

Observa pues que:
- En primer diseño: Mando dependía de lampara, una clase muy concreta y por tanto de “bajo nivel”
- En segundo diseño: aplicando inversión de dependencias eliminamos dependencias de objetos de bajo nivel. Lampara(bajo nivel) pasa a depender de Conmutable y Mando también pasa a depender de Conmutable.

¡Las dependencias han sido invertidas! lo más importante no depende de lo menos importante.

Nota: ojo, en POO hay mucha terminología. No confundir Inversión de Dependencias Inyección de Dependencias. Inyección de dependencias es un patrón. Veremos lo que es un patrón más adelante.

Ejercicio: Implementa en java el diseño anterior, de forma que funcione la siguente App

```
public class App {
    public static void main(String[] args) {
       Lampara l= new Lampara();
       Television t= new Television();
       Mando m1= new Mando(l);
       m1.presionar();
       m1.presionar();
       m1.presionar();
       //otro mando con otro dispositivo sin problemas
       Mando m2= new Mando(t);
       m2.presionar();
       m2.presionar();
    }
}
```

que genera la siguiente salida

```
Lampara encendida
Lampara apagada
Lampara encendida
Tele encendida
Tele apagada
```

Iremos introduciendo poco a poco los patrones de diseño. Este principio se recoge en muchos patrones, como por ejemplo en Observer y Strategy