TALENTO DIGITAL
INTELIGENCIA HUMANA

# Talento Digital para Chile:

## **MÓDULO** 1
## PROGRAMACIÓN BÁSICA EN JAVA

UN PROYECTO DE:

Ministerio de Hacienda · Ministerio del Trabajo y Previsión Social · CORFO · sence · SOFOFA · OTIC SOFOFA Capital Humano · CPC · ACTI · BID · FCH Fundación Chile · kodea

DESARROLLADO POR:

# MÓDULO 1 - PROGRAMACIÓN BÁSICA EN JAVA

## 1.4 PRINCIPIOS BÁSICOS DE DISEÑO ORIENTADO A OBJETOS

**Semana 4 - Día 19**

### Objetivo de la jornada

- Comprender y aplicar el principio de abierto cerrado para mejorar el código de un problema según requerimientos establecidos.
- Entender el principio de sustitución de Liskov para mejorar el código de problemas planteados según indicaciones entregadas.
- Conocer el principio de Segregación de Interfaces para luego utilizarlos para mejorar el código de problemas planteados.

### Principio de Abierto-Cerrado

Este principio hace referencia a que una entidad de Software debería estar abierta a extensión pero cerrada a modificación, esto significa que se debe extender el comportamiento de las clases sin necesidad de modificar su código. Esto ayuda a seguir añadiendo funcionalidades sin afectar el código existente. Nuevas funcionalidades implicarán añadir nuevas clases y métodos, pero en general no debería suponer modificar lo que ya ha sido escrito.

Una de las formas más sencillas para detectarlo es darse cuenta de qué clases se modifican a menudo. Si cada vez que hay un nuevo requisito o una modificación de los existentes, las mismas clases se ven afectadas, podemos empezar a entender que estamos violando este principio.

Un ejemplo de aplicación del principio Abierto/Cerrado en código Java sería lo siguiente:

Se cuenta con una clase (Vehiculo) con un método que se encarga de imprimir un vehículo por pantalla. Por supuesto, cada vehículo tiene su propia forma de ser pintado:

```
public class Vehiculo{
    public TipoVehiculo getTipo() {
    }
}
```

Básicamente es una clase que especifica su tipo mediante un enumerado. Podemos tener por ejemplo un enum con un par de tipos:

```
public enum TipoVehiculo {
    AUTO,
    MOTO
}
```

El método de la clase que se encarga de pintar los vehículos es el siguiente:

```
public void Imprmir(Vehiculo veh) {
    switch (veh.getTipo()) {
        case AUTO:
            ImprimirAuto(veh);
            break;
        case MOTO:
            Imprimir(veh);
            break;
    }
}
```

Mientras no se necesiten dibujar más tipos de vehículos ni se vea que este switch se repite en varias partes del código, no se debe modificar. Incluso el hecho de que cambie la forma de dibujar un auto o una moto estaría encapsulado en sus propios métodos y no afectaría al resto del código. Sin embargo, se puede llegar al punto en el que se necesite dibujar uno o más tipos de vehículos, lo que implicaría crear un nuevo enumerado, un nuevo case y un nuevo método para implementar el dibujado del vehículo. En este caso sería buena idea aplicar el principio Abierto/Cerrado.

Una solución a esto mediante herencia y polimorfismo, es sustituir ese enumerado por clases reales, y que cada clase sepa cómo pintarse:

```
public abstract class Vehiculo {
    public abstract void Imprimir();
}
public class Auto extends Vehiculo {
     @Override public void Imprimir() {
    }
 }
 public class Moto extends Vehiculo {
      @Override public void Imprimir() {
       }
 }
```

El método anterior quedaría reducido a esto:

```
public void Imprimir(Vehiculo veh) {
    veh.Imprimir();
}
```

Siguiendo estas modificaciones, añadir nuevos vehículos es tan sencillo como crear la clase correspondiente que extienda de Vehiculo:

```
public class Camion extends Vehiculo {
  @Override public void Imprimir() {
   }
}
```

Este principio es casi imposible de llevar a cabo en un 100% y puede hacer que sea ilegible e incluso más difícil de mantener. Este principio, al igual que el resto de las reglas SOLID se deben aplicar cuando corresponda y sin obsesionarse con cumplirlas en cada punto del desarrollo. En la mayoría de los casos es más sencillo limitarse a usarlas cuando surjan necesidades reales.

**Principio de Sustitución de Liskov**

El principio de sustitución de Liskov menciona que si en alguna parte del código se está 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 obliga a asegurarse de que cuando se extiende una clase no se está alterando el comportamiento de la clase padre.

En lenguaje más formal: si S es un subtipo de T, entonces los objetos de tipo T en un programa de computadora pueden ser sustituidos por objetos de tipo S (es decir, los objetos de tipo S pueden sustituir objetos de tipo T), sin alterar ninguna de las propiedades deseables de ese programa (la corrección, la tarea que realiza, etc.).

Algunos beneficios del principio de Sustitución de Liskov son los siguientes:

- El código es más reutilizable.
- Facilidad de entendimiento de las jerarquías de clase.
- Validar que las abstracciones están correctas.

Es muy común basar los diseños en las propiedades de las clases del objeto que representan el mundo real, lo que puede llevar a cometer errores, sin embargo, si se basan en las conductas de los objetos, es posible evitar estos errores.

**Principio de Segregación de Interfaces**

El principio de segregación de interfaces viene a decir que ninguna clase debería depender de métodos que no usa. Por lo tanto, cuando se crean interfaces que definan comportamientos, es importante asegurarse 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.

Las interfaces ayudan a desacoplar módulos entre sí. Esto debido a que si se tiene una interfaz que explica el comportamiento que el módulo espera para comunicarse con otros módulos, siempre se podrá crear una clase que lo implemente de modo que cumpla las condiciones. El módulo que describe la interfaz no tiene que saber nada sobre el código y, sin embargo, se podrá trabajar con él sin problemas.

La problemática surge cuando esas interfaces intentan definir más cosas de las debidas, por ejemplo que las clases hijas no usen muchos de esos métodos, y habrá que entregarles una implementación. Es muy habitual lanzar una excepción o simplemente no hacer nada al respecto.

Si se lanza una excepción, es más que probable que el módulo que define esa interfaz use el método en algún momento, y esto hará fallar el programa. El resto de implementaciones que se puedan dar generan efectos secundarios no esperados, y a los que sólo se podrá responder conociendo el código fuente del módulo en cuestión.

Si al implementar una interfaz se logra apreciar que uno o varios de los métodos no tienen sentido y pareciera hacer falta dejarlos vacíos o lanzar excepciones, es muy probable que se esté violando este principio. Si la interfaz forma parte del código, se debe dividir en varias interfaces que definan comportamientos más específicos.

El principio de segregación de interfaces ayuda a no obligar a ninguna clase a implementar métodos que no utiliza. Esto evitará problemas que puedan llevar a errores inesperados y a dependencias no deseadas.