Programación orientada a objetos Individual - POO - Unidad 3
Actividad: Taller sobre Principios SOLID en Java - Abierto/Cerrado
Tutor: JOHN CARLOS ARRIETA ARRIETA DOCENTE

Taller: Principio de Abierto/Cerrado (OCP - Open/Closed Principle)

Objetivo del Taller:

En este taller, los estudiantes aprenderán el **Principio de Abierto/Cerrado (OCP)**, uno de los principios fundamentales del diseño orientado a objetos en SOLID. Este principio establece que las clases, módulos o funciones deben estar **abiertas para la extensión, pero cerradas para la modificación**. El taller aborda definiciones formales y cotidianas, sus beneficios, desventajas, conceptos previos, ejemplos correctos e incorrectos, y ejercicios propuestos para afianzar el conocimiento.

---

1. Definición Formal del OCP:

El **Principio de Abierto/Cerrado (Open/Closed Principle)** establece que **una clase debe estar abierta para la extensión, pero cerrada para la modificación**. Esto significa que se debe poder agregar nueva funcionalidad a las clases sin necesidad de modificar el código existente.

- **Definición Técnica:** "Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para la extensión, pero cerradas para la modificación."

Imagina que construyes una casa (aplicación poo) con una estructura sólida. Si más adelante quieres hacer mejoras (métodos) o agregar una habitación (clase o interface), en lugar de destruir una pared estructural (modificación el código de las clases), simplemente construyes la nueva habitación (clase o interface) anexa (extensión o herencia). De igual forma, en el diseño de software, debes poder agregar nuevas funcionalidades sin alterar el código que ya funciona (herencia y polimorfismo).

---

2. Beneficios del OCP:

- **Facilita el Mantenimiento:** Permite agregar nuevas características al sistema sin modificar el código existente, lo que minimiza el riesgo de introducir nuevos errores.
- **Permite la Evolución del Código:** Los requisitos de software suelen cambiar con el tiempo, y este principio facilita el crecimiento del sistema de manera segura.
- **Mayor Reusabilidad:** Las clases extensibles mediante la herencia o el uso de interfaces son más reutilizables en diferentes contextos.

---

3. Desventajas del OCP:

- **Complejidad Inicial:** Aplicar el OCP puede requerir una planificación y diseño inicial más complejo, utilizando patrones de diseño como la herencia, la composición o las interfaces.
- **Aumento de Clases:** La aplicación del OCP puede llevar a la creación de múltiples clases, lo que puede resultar difícil de gestionar si no se tiene una estructura clara.

---

4. Conceptos Relacionados que se Deben Dominar:

- **Herencia:** Capacidad de una clase para heredar las características de otra clase.
- **Polimorfismo:** Capacidad de diferentes clases para implementar métodos con la misma firma de diferentes maneras.
- **Abstracción:** Definición de una clase base o interfaz que define un comportamiento común sin especificar detalles de implementación.

---

5. Ejemplos de Aplicación del OCP:

Ejemplo 1: Sistema de Pago con Extensión para Nuevos Métodos de Pago

- **Situación Inicial:** Una clase `ProcesadorPago` solo maneja pagos con tarjeta de crédito. Se necesita agregar pagos con PayPal.
- **Aplicación del OCP:** Crear una clase base `Pago` y clases específicas para cada método de pago.

```
// Clase base Pago
public abstract class Pago {
    public abstract void procesarPago(double monto);
}
```

```
// Clase TarjetaCredito
```

```java
public class TarjetaCredito extends Pago {
    @Override
    public void procesarPago(double monto) {
        System.out.println("Procesando pago con tarjeta de crédito: $" + monto);
    }
}
```

```java
// Clase PayPal
public class PayPal extends Pago {
    @Override
    public void procesarPago(double monto) {
        System.out.println("Procesando pago con PayPal: $" + monto);
    }
}
```

```java
// Clase ProcesadorPago
public class ProcesadorPago {
    public void realizarPago(Pago metodoPago, double monto) {
        metodoPago.procesarPago(monto);
    }
}
```

Explicación: Si en el futuro se necesita agregar otro método de pago (por ejemplo, `PagoBitcoin`), se puede extender sin modificar el código de `ProcesadorPago`.

Ejemplo 2: Notificaciones por Diferentes Canales

- **Situación Inicial:** Una clase `Notificacion` solo envía correos electrónicos y se requiere agregar notificaciones por SMS.
- **Aplicación del OCP:** Crear una clase base `Notificacion` e implementar diferentes clases para cada canal de notificación.

```java
// Clase base Notificacion
public abstract class Notificacion {
    public abstract void enviar(String mensaje);
}
```

```java
// Clase NotificacionEmail
public class NotificacionEmail extends Notificacion {
    @Override
    public void enviar(String mensaje) {
        System.out.println("Enviando email: " + mensaje);
    }
}
```

```java
// Clase NotificacionSMS
public class NotificacionSMS extends Notificacion {
    @Override
    public void enviar(String mensaje) {
        System.out.println("Enviando SMS: " + mensaje);
    }
}
```

```java
// Clase GestorNotificacion
public class GestorNotificacion {
    public void notificar(Notificacion notificacion, String mensaje) {
        notificacion.enviar(mensaje);
    }
}
```

Explicación: Si se necesita agregar otro canal de notificación (por ejemplo, `NotificacionPush`), se puede hacer sin cambiar el código de `GestorNotificacion`.

---

6. Ejemplos de Violación del OCP:

Ejemplo 1: Clase `ProcesadorPago` que Modifica Código para Nuevos Métodos de Pago

- **Violación:** Cada vez que se agrega un nuevo método de pago, se modifica la clase `ProcesadorPago`.

```
public class ProcesadorPago {
    public void realizarPago(String metodo, double monto) {
        if (metodo.equals("TarjetaCredito")) {
            System.out.println("Procesando pago con tarjeta de crédito: $" + monto);
        } else if (metodo.equals("PayPal")) {
            System.out.println("Procesando pago con PayPal: $" + monto);
        }
        // Problema: Si se agrega otro método, hay que modificar este código.
    }
}
```

Ejemplo 2: Clase `Notificacion` que Modifica Código para Nuevos Canales

- **Violación:** La clase `Notificacion` maneja directamente los diferentes canales, lo cual viola el OCP.

```
public class Notificacion {
    public void enviar(String canal, String mensaje) {
        if (canal.equals("Email")) {
            System.out.println("Enviando email: " + mensaje);
        } else if (canal.equals("SMS")) {
            System.out.println("Enviando SMS: " + mensaje);
        }
        // Problema: Si se agrega otro canal, hay que modificar este código.
    }
}
```

---

7. Ejercicios Propuestos

Ejercicio 1: Sistema de Descuento para Tienda

1. Define una clase `Descuento` que solo maneje descuentos de porcentaje.
2. Extiende el sistema para agregar descuentos fijos sin modificar la clase base `Descuento`.

Ejercicio 2: Gestión de Documentos con Diferentes Formatos

1. Define una clase `Documento` que solo maneje la exportación a PDF.
2. Extiende el sistema para permitir exportar a otros formatos como Word o Excel sin modificar el código existente.

Ejercicio 3: Sistema de Envío de Mensajes

1. Define una clase `Mensajero` que solo maneje el envío de correos electrónicos.
2. Refactoriza el sistema para permitir el envío de mensajes por diferentes canales (por ejemplo, SMS o notificaciones push) sin modificar la clase base.

Conclusión del Taller:

Este taller ha proporcionado una comprensión profunda del **Principio de Abierto/Cerrado (OCP)** y cómo aplicarlo para crear sistemas que puedan ser extendidos sin necesidad de modificar el código existente. Al aplicar el OCP, los estudiantes aprenderán a estructurar sus sistemas de una manera más flexible y menos propensa a errores.