**Concepto principal del OCP (Open/Closed Principle)**

El principio de Open/Closed (Abierto/Cerrado) es uno de los cinco fundamentos de SOLID, orientado a promover un diseño de software más flexible, escalable y mantenible. Este principio establece que "una entidad de software (como una clase, módulo o función) debe estar abierta para su extensión, pero cerrada para su modificación."

**🟢 Significado de “abierto para extensión”**

Una clase está abierta para extensión cuando permite agregar nuevas funcionalidades o comportamientos sin modificar su estructura interna. Esto puede lograrse utilizando mecanismos como:

* Herencia (subclases que agregan nuevas funcionalidades).
* Interfaces (permiten múltiples implementaciones sin alterar la interfaz original).
* Composición (combinar comportamientos en tiempo de ejecución).

**🔴 Significado de “cerrado para modificación”**

Una entidad está cerrada para modificación cuando su código fuente permanece inalterado tras ser probado y utilizado. Esto es crucial para:

* Evitar la introducción de errores en funcionalidades ya existentes.
* Reducir el riesgo de regresiones.
* Proteger la lógica de negocio ya validada.

**Origen y relevancia dentro de SOLID**

El principio fue introducido por Bertrand Meyer en 1988 en su libro *Object-Oriented Software Construction*. Más adelante, Robert C. Martin (Uncle Bob) reinterpretó el principio en el contexto de diseño de software moderno, enfocándose en el uso de abstracciones para lograr esta extensibilidad sin modificar código existente.

**Importancia dentro de SOLID:**

* Promueve diseños desacoplados.
* Facilita la incorporación de nuevas funcionalidades.
* Mejora la mantenibilidad a largo plazo.

**Problemas que ayuda a evitar**

* Evita romper funcionalidades existentes, ya que el código probado no se toca.
* Reduce la duplicación de código, al extenderse mediante clases nuevas en vez de replicar comportamientos.
* Minimiza el acoplamiento, lo cual permite que distintos componentes evolucionen de forma independiente.

**Analogías para comprender el Principio Open/Closed**

El principio Open/Closed puede parecer abstracto al principio, pero se vuelve más claro cuando lo relacionamos con objetos o sistemas del mundo real que también están cerrados para modificación, pero abiertos a extensión. A continuación, se presentan varias analogías que ilustran cómo funciona este principio en la vida cotidiana:

**Puerto USB**

* **Analogía**: Puedes conectar nuevos dispositivos (mouse, teclado, disco externo) sin modificar el puerto USB.
* **Relación con OCP**: El puerto USB representa una interfaz cerrada para modificación, pero abierta a nuevas extensiones mediante dispositivos compatibles.

**Plugins en aplicaciones (ej. WordPress, Photoshop)**

* **Analogía**: Se pueden añadir nuevas funciones (temas, efectos, herramientas) sin tocar el código base de la aplicación.
* **Relación con OCP**: El núcleo del software permanece intacto, mientras que las funcionalidades se amplían mediante plugins.

**Cajones modulares (muebles)**

* **Analogía**: Puedes agregar nuevos cajones a un mueble sin reconstruirlo por completo.
* **Relación con OCP**: El diseño base no se modifica, pero está abierto a crecer con nuevos componentes.

**Aplicaciones modulares**

* **Analogía**: Puedes instalar nuevas funciones como calendario, tareas, música, etc., sin reescribir la app original.
* **Relación con OCP**: El sistema se adapta con módulos nuevos sin modificar lo ya existente.

**Enchufe universal**

* **Analogía**: Acepta múltiples tipos de conectores sin necesidad de cambiar el enchufe.
* **Relación con OCP**: Representa una estructura única que permite adaptación sin ser alterada.

**Ejemplos con código (Java)**

**Ejemplo sin aplicar el principio Open/Closed**

🔹 Clases involucradas:

CalculadoraDescuentos: contiene un método calcularDescuento() que usa if-else o switch para decidir qué tipo de descuento aplicar.

Problema: cada vez que se agrega un nuevo tipo de cliente, hay que modificar esta clase.

### **Problemas:**

* Cada vez que agregas un nuevo tipo de cliente modificas esta clase.
* Acoplamiento alto y difícil mantenimiento.
* No cumple con el Principio de **cerrado para modificación.**

**Ejemplo aplicando el principio Open/Closed correctamente**

🔹 Clases involucradas:

PoliticaDescuento (interfaz): define un método calcularDescuento(double monto).

DescuentoEstudiante, DescuentoTerceraEdad, DescuentoEmpleado: cada una implementa la interfaz y aplica su propia lógica.

CalculadoraDescuentos: delega el cálculo a una implementación concreta de PoliticaDescuento.

Resultado: podemos añadir nuevos tipos de descuento sin modificar las clases existentes.

### **Ventajas:**

* Puedes agregar nuevas políticas (como DescuentoVip) sin tocar CalculadoraDescuentos.
* Cumple con OCP: **cerrado para modificación, abierto para extensión**.
* Bajo acoplamiento, alta escalabilidad.

APLICANDO UML

Ejemplo gráfico de una mala implementación (clases modificadas constantemente).

Diagrama que respete el OCP (clases extendidas sin modificar las existentes).

Explicación de las diferencias visuales y estructurales.

**Explicación:**

* Toda la lógica de decisión está en CalculadoraDescuentos.
* La clase se modifica constantemente para agregar nuevos tipos de clientes.
* Está fuertemente acoplada y no respeta SRP (tiene múltiples responsabilidades).

**Explicación:**

* PoliticaDescuento actúa como una abstracción, permitiendo nuevas extensiones sin modificar CalculadoraDescuentos.
* Se agregan nuevas clases como DescuentoEmpleado sin tocar el código ya existente.
* Cada clase tiene una única responsabilidad y está desacoplada.
* Cumple con los principios OCP y SRP.

**Opciones de implementación**

El principio Open/Closed es clave para construir software robusto, mantenible y extensible, especialmente en entornos en constante evolución. Su implementación eficaz se logra mediante abstracción, interfaces y patrones de diseño como Strategy, Template Method y Decorator, que permiten añadir nuevas funcionalidades sin romper lo ya construido. Sin embargo, debe aplicarse con criterio, evitando el exceso de abstracciones prematuras y priorizando la simplicidad cuando el contexto lo requiere.

**Herencia**

Mediante el uso de clases base con métodos definidos de forma abstracta, las clases derivadas pueden extender esa clase base y proporcionar su propia implementación de los métodos. Lo que permite agregar nuevas funcionalidades o comportamientos sin modificar el código original.

**Composición y Abstracción**

Se definen interfaces que declaran métodos sin implementar su comportamiento. Nuevas funcionalidades se agregan cuando se crean nuevas clases que implementan esta interfaz con su propia versión. Las clases son más “independientes” y el programa es más flexible.

**Patrones de Diseño OCP**

**Strategy Pattern (Behavioral Pattern)**

Esta estrategia define una familia de algoritmos, encapsula cada una y las hace intercambiables. El algoritmo varía independientemente de los clientes que lo utilizan (Gamma, 1994). Este patrón de comportamiento está construido sobre tres pilares básicos:

* Interfaz de estrategia: Declara una interfaz común que implementarán las clases interesadas.
* Estrategia concreta: Encapsula una implementación específica del comportamiento definido por una interfaz común, y permite que el efiner delegue dinámicamente su lógica.
* Contexto (Composición):  Se encuentra configurado por la estrategia concreta. Mantiene una referencia con la interfaz de estrategia y puede efiner una interfaz que deje a la estrategia acceder a sus datos.

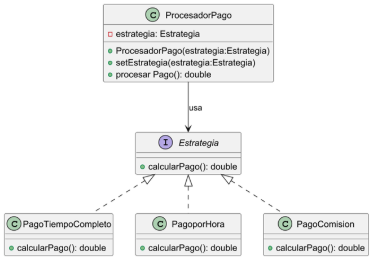


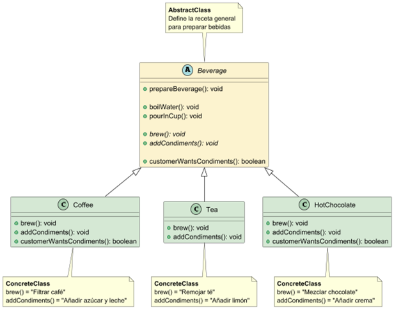
Ilustración 1: Ejemplo Strategy Pattern

**Template method (Behavioral Pattern)**

El patrón Template Method permite definir el esqueleto de un algoritmo en una clase abstracta, delegando ciertos pasos en subclases que pueden redefinirlos según sus necesidades. Esto facilita la extensión del comportamiento sin modificar la estructura original del algoritmo, cumpliendo así con el OCP.

Este patrón es ideal cuando se necesita garantizar la coherencia en el flujo de trabajo entre distintas implementaciones, permitiendo variaciones solo en pasos concretos:

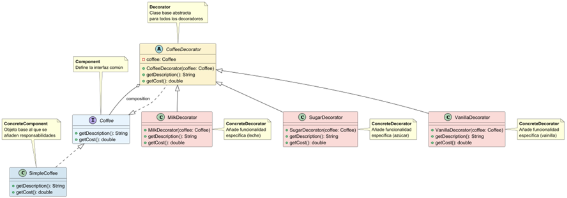
* Clase abstracta: Define el método de plantilla que implementa el esqueleto del algoritmo. Declara operaciones primitivas abstractas que deben ser implementadas por las subclases.
* Clase concreta: Implementa las operaciones primitivas necesarias para ejecutar los pasos específicos del algoritmo. Puede sobrescribir los métodos “hook” si desea modificar un comportamiento puntual sin alterar la estructura general.



**Decorator Pattern (Structural Pattern)**

Patrón de diseño que te permite extender el comportamiento de un objeto dinámicamente en tiempo de ejecución sin modificar la estructura interna (Añadir nuevas características o funciones). Es un enfoque más flexible (que usar herencia, por ejemplo) ya que encierra los componentes en otro objeto que agrega el borde.

* Componente: Define la interfaz común que tanto los objetos originales como los decorados deben implementar.
* Componente Concreto: Es el objeto base al que se le pueden añadir nuevas responsabilidades.
* Decorador: Mantiene una referencia al componente y define una interfaz compatible con la del componente.
* Decorador Concreto: Son los decoradores específicos que añaden funcionalidades concretas al componente.



**Nota:** Refactorizar hacia OCP se justifica cuando emergen patrones reales de cambio y extensión en el código. Como: la aparición repetida de modificaciones similares en el mismo módulo, necesidad frecuente de añadir nuevas variantes de comportamiento, o cuando el código comienza a exhibir múltiples responsabilidades.

## **Ventajas del Principio Open/Closed (OCP)**

### 🔹 1. Escalabilidad

Permite que el sistema crezca con nuevas funcionalidades sin alterar el código ya existente.

*Ejemplo*:  
En una aplicación de mensajería, si deseas añadir un nuevo canal (como WhatsApp), puedes crear una nueva clase CanalWhatsApp que implemente la interfaz CanalDeEnvio sin modificar los canales existentes (CanalEmail, CanalSMS).

*Beneficio clave*: Ahorra tiempo de desarrollo y reduce el riesgo de errores en módulos ya estables.

### 🔹 2. Mantenibilidad

El código existente permanece intacto. Esto reduce el riesgo de introducir nuevos bugs al cambiar partes ya probadas.

*Ejemplo*:  
Un sistema de cálculo de impuestos que ya tiene varias clases (ImpuestoIVA, ImpuestoICE) no necesita tocar su lógica base si se desea agregar un nuevo impuesto (ImpuestoVerde). Basta con extender.

*Beneficio clave*: Facilita pruebas unitarias, reduce regresiones y mejora la trazabilidad de cambios.

### 🔹 3. Flexibilidad

Al basarse en interfaces o clases abstractas, puedes reconfigurar o añadir funcionalidades con mayor libertad.

*Ejemplo*:  
Una tienda online puede añadir promociones dinámicas (2x1, 10% de descuento, cupones) como nuevas estrategias sin alterar la clase principal de cálculo de precio.

*Beneficio clave*: El sistema se adapta fácilmente a nuevos requerimientos del negocio.

## **Desventajas del Principio OCP**

### 🔸 1. Complejidad innecesaria

Aplicar OCP en todos los casos puede llevar a diseñar múltiples capas de abstracción sin beneficio real.

*Ejemplo*:  
En un programa simple de calculadora, crear una interfaz OperacionMatematica para sumar, restar, multiplicar y dividir podría ser una exageración, cuando un simple switch sería más directo.

*Problema*: Dificulta el mantenimiento para tareas simples y puede confundir a nuevos desarrolladores.

### 🔸 2. Sobrecarga de clases e interfaces

Una mala aplicación del OCP puede generar una **explosión de clases** por cada pequeña variación de comportamiento.

*Ejemplo*:  
Crear una clase nueva por cada tipo de producto en un carrito de compras (ProductoFisico, ProductoDigital, ProductoAlquilado, etc.) sin distinguir si realmente se comportan distinto.

*Problema*: Aumenta el tamaño del proyecto innecesariamente y complica la navegación del código.

## **Relación del OCP con otros principios SOLID**

### 1. SRP – Principio de Responsabilidad Única

* OCP se facilita cuando las clases tienen una única razón de cambio.
* Si una clase tiene múltiples responsabilidades, extenderla sin modificarla se vuelve difícil o incluso imposible.

*Ejemplo*:  
Una clase Factura que imprime, calcula impuestos y guarda en base de datos viola SRP y OCP. Dividirla en CalculadorImpuestos, ImpresoraFactura, RepositorioFactura permite aplicar OCP limpiamente.

### 2. LSP – Principio de Sustitución de Liskov

* OCP y LSP están íntimamente ligados. Si una subclase no puede reemplazar a su superclase sin errores, entonces no está bien diseñada y rompe el OCP.

*Ejemplo*:  
Si FacturaTarjeta lanza errores cuando se usa en lugar de Factura, entonces no respeta LSP ni OCP. Debe comportarse de manera coherente con su clase base.

### 3. DIP – Principio de Inversión de Dependencias

* OCP necesita que el código dependa de abstracciones y no de implementaciones concretas.
* DIP proporciona la base técnica para lograr OCP al permitir la inyección de dependencias y flexibilidad para cambiar comportamientos.

*Ejemplo*:  
Una clase ProcesadorDePago debería depender de una interfaz MetodoPago, no de PagoTarjeta o PagoEfectivo. Así puedes inyectar fácilmente nuevas extensiones sin modificar el código del procesador.

**🔗 Referencias**

* Trbl. (2024, 25 enero). *Principios S.O.L.I.D. – "Open-Closed"*. TRBL Services. <https://trbl-services.eu/blog-solid-open-closed>
* Maluenda, R. (2022, 12 mayo). *SOLID: los 5 principios que te ayudarán a desarrollar software de calidad*. Profile Software Services. <https://profile.es/blog/principios-solid-desarrollo-software-calidad/#2_Principio_de_AbiertoCerrado>
* Arsys. (s. f.). *Principios SOLID en programación orientada a objetos*. <https://www.arsys.es/blog/principios-solid-en-la-programacion-orientada-a-objetos>
* Maldonado, L. (2021). *Analogías para entender los principios SOLID*. https://medium.com/@luismaldonado
* Arsys. (s.f.). *Principios SOLID en programación orientada a objetos*. <https://www.arsys.es/blog/principios-solid-en-la-programacion-orientada-a-objetos>
* Cueva, J. (2023). *Open/Closed Principle – Ejemplos fáciles*. <https://jcdev.io/solid-open-closed-principle-ejemplos>
* Bilescu, A. (6 de Mayo de 2023). Codementor. Obtenido de Investing in Code Quality: The Decorator Pattern and Its Role in Implementing SOLID Principles: https://www.codementor.io/@adrianbilescu/investing-in-code-quality-the-decorator-pattern-and-its-role-in-implementing-solid-principles-24jb2i9ghf
* Darde, P. R. (18 de Noviembre de 2021). Medium. Obtenido de Learning the Open-Closed principle with the Strategy design pattern: https://medium.com/@pablodarde/learning-the-open-closed-principle-with-the-strategy-design-pattern-933dfa04d1e8
* Gamma, E. (1994). Design Patterns. Elements of Reusable Object-Oriented Software. Massachusetts: Addison-Wesley.
* Hossain, M. (3 de Febrero de 2025). Medium. Obtenido de Explained Design Patterns: Strategy and Template Method: https://medium.com/@mail2mhossain/explained-design-patterns-strategy-and-template-method-f57acfc0e2bf
* Öztürk, T. (6 de Enero de 2023). Medium. Obtenido de A Practical Guide to Implementing the Open-Closed Principle in Your Code: https://medium.com/@ozturktariik/a-practical-guide-to-implementing-the-open-closed-principle-in-your-code-181d58962ee7
* Patel, A. (30 de Junio de 2023). The Server Side. Obtenido de How to apply the open-closed principle in Java: https://www.theserverside.com/tip/How-to-apply-the-open-closed-principle-in-Java