Patrón de diseño

Un patrón de diseño representa una respuesta universal y reutilizable a problemas habituales en el diseño de software. Estos modelos ofrecen un esquema claramente establecido para estructurar el código, lo que simplifica su mantenimiento, reutilización y escalabilidad.
Los patrones de diseño no son un código concreto, sino orientaciones para solucionar problemas de arquitectura de software a través de buenas prácticas. Se emplean en la codificación orientada a objetos (POO) y facilitan la creación de sistemas más eficaces, versátiles y sólidos.

Tipos

Patrones Creacionales:

Estos patrones se centran en la mejor manera de crear objetos, asegurando que el proceso sea eficiente y flexible. Evitan la creación directa de instancias mediante el uso de constructores, promoviendo en su lugar la encapsulación y delegación de la lógica de creación.Ejemplos: Singleton, Factory Method, Builder, Prototype, Abstract Factory.

Patrones Estructurales:

Estos patrones definen la composición de clases y objetos para formar estructuras más complejas y eficientes. Ayudan a organizar mejor el código y facilitan la integración entre diferentes componentes del sistema. Ejemplos: Adapter, Composite, Decorator, Facade, Proxy, Bridge.

Patrones de Comportamiento:

Se enfocan en la interacción y comunicación entre objetos, facilitando la asignación de responsabilidades y la cooperación entre diferentes partes del código. Ejemplos: Observer, Strategy, Command, State, Mediator, Chain of Responsibility.

Ejemplos

Patrón Adapter: El patrón Adapter permite que clases con interfaces incompatibles trabajen juntas. Actúa como un puente entre dos interfaces que no son compatibles directamente, permitiendo que una clase adapte su interfaz a la esperada por otra clase.

```
// Interfaz esperada por el cliente
public interface ITarget
{
    void Request();
}
// Clase existente con una interfaz incompatible
public class Adaptee
{
    public void SpecificRequest()
    {
        Console.WriteLine("Solicitud específica.");
    }
}
// Adaptador que adapta la interfaz de Adaptee a la esperada por ITarget
public class Adapter : ITarget
{
    private Adaptee _adaptee;
    public Adapter(Adaptee adaptee)
    {
        _adaptee = adaptee;
    }
    public void Request()
    {
        _adaptee.SpecificRequest();
    }
}
```

Aplicaciones del Adapter:

- Integración de sistemas con interfaces diferentes.
- Conexión de clases heredadas con nuevas interfaces.
- Implementación de adaptadores para dispositivos con distintos estándares.

Patrón Decorator

Utilidad:

El patrón Decorator permite agregar responsabilidades a un objeto de manera dinámica, sin afectar a otros objetos de la misma clase. Es útil cuando necesitas añadir funcionalidades a un objeto sin modificar su estructura original, permitiendo extender sus capacidades de forma flexible.

```
// Componente base
public interface IComponent
{
    void Operation();
}
// Implementación concreta del componente
public class ConcreteComponent : IComponent
{
    public void Operation()
    {
        Console.WriteLine("Operación del componente base.");
    }
}
// Decorador abstracto
public abstract class Decorator : IComponent
{
    protected IComponent _component;

    public Decorator(IComponent component)
    {
        _component = component;
    }

    public virtual void Operation()
    {
        _component.Operation();
    }
}
// Decorador concreto que agrega funcionalidad
public class ConcreteDecoratorA : Decorator
{
    public ConcreteDecoratorA(IComponent component) : base(component) {}

    public override void Operation()
    {
        base.Operation();
        Console.WriteLine("Operación agregada por ConcreteDecoratorA.");
    }
}
```

Aplicaciones del Decorator:

- Añadir funcionalidades a objetos en tiempo de ejecución.
- Extender clases sin necesidad de herencia.

- Decorar objetos gráficos o visuales en aplicaciones.

Patrón Strategy

Utilidad:

El patrón Strategy permite cambiar el comportamiento de un objeto en tiempo de ejecución mediante la sustitución de su algoritmo. Este patrón es útil cuando se tiene una familia de algoritmos que pueden ser intercambiados dependiendo de la situación, sin modificar el cliente que los utiliza.

```
// Interfaz que define un comportamiento de algoritmo
public interface IStrategy
{
    void Execute();
}
// Algoritmos concretos que implementan la estrategia
public class ConcreteStrategyA : IStrategy
{
    public void Execute()
    {
        Console.WriteLine("Estrategia A ejecutada.");
    }
}
public class ConcreteStrategyB : IStrategy
{
    public void Execute()
    {
        Console.WriteLine("Estrategia B ejecutada.");
    }
}
// Contexto que usa una estrategia
public class Context
{
    private IStrategy _strategy;
```

```
    public Context(IStrategy strategy)
    {
        _strategy = strategy;
    }
    public void SetStrategy(IStrategy strategy)
    {
        _strategy = strategy;
    }
    public void ExecuteStrategy()
    {
        _strategy.Execute();
    }
}
```

Aplicaciones del Strategy:

- Elección dinámica de algoritmos, como en compresión de datos o algoritmos de búsqueda.
- Cambio de estrategias en videojuegos, como diferentes modos de ataque.
- Sistemas de pago que permiten cambiar de estrategia dependiendo del método (tarjeta, PayPal, etc.).

Estos tres patrones de diseño ofrecen soluciones elegantes y flexibles para problemas comunes en la programación.

- Adapter facilita la integración de sistemas con interfaces incompatibles.
- Decorator permite extender funcionalidades sin modificar el objeto original.
- Strategy permite intercambiar algoritmos o comportamientos de manera dinámica.