Patrón de Diseño Adapter

Introducción

El **patrón de diseño Adapter** es un patrón estructural que permite a dos interfaces incompatibles trabajar juntas. En lugar de modificar las clases existentes, se crea un adaptador que actúa como un intermediario, convirtiendo la interfaz de una clase en una que la otra clase pueda entender.

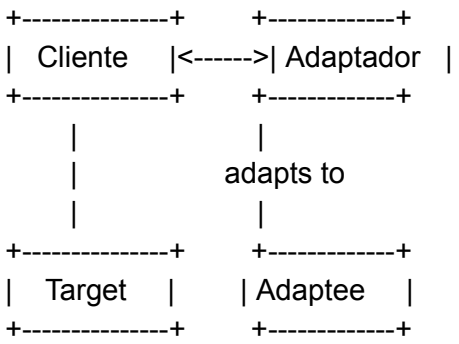
Propósito

Permitir que dos clases o interfaces incompatibles puedan trabajar juntas. Este patrón convierte la interfaz de una clase en otra interfaz que el cliente espera.

Motivación

Imagina que tienes un sistema que trabaja con una API externa, pero los métodos de esta API no coinciden con los métodos que utiliza tu sistema. El Adapter permite crear una "capa de adaptación" que hace que ambos sistemas puedan interactuar sin conflictos.

Diagrama de clases del Adapter



Cliente: Clase que utiliza la interfaz `Target`.
Target: Define la interfaz que el `Cliente` necesita.
Adaptador: Implementa `Target` y traduce las solicitudes del `Cliente` a una forma que `Adaptee` pueda manejar.
Adaptee: Clase con una interfaz incompatible que se desea adaptar.

Participantes

- **Target**: Define la interfaz esperada por el cliente.
- **Adaptador (Adapter)**: Traduce la interfaz de `Adaptee` a la interfaz de `Target`.
- **Adaptee**: Clase existente que tiene la funcionalidad requerida, pero una interfaz incompatible.

- **Cliente**: Interactúa con el sistema a través de `Target`.

Ventajas y Desventajas

- **Ventajas**:
  - Facilita la reutilización de código.
  - Permite trabajar con interfaces incompatibles sin modificar su código fuente.
  - Mejora la flexibilidad y la capacidad de extensión del sistema.
- **Desventajas**:
  - Puede aumentar la complejidad del sistema.
  - Puede llevar a una sobrecarga en el sistema si se usa en exceso.

Aplicaciones del Adapter

- Cuando necesitas utilizar una clase existente y su interfaz no coincide con lo que el cliente espera.
- Para integrar componentes de terceros en un sistema sin modificar su código fuente.

Ejemplo de Integración de una Pasarela de Pago con el Patrón Adapter

Conectar un sistema de pagos que ya tienes en funcionamiento con una nueva pasarela de pago externa. Supongamos que tienes una plataforma de e-commerce con un sistema que procesa pagos y este sistema espera que las pasarelas de pago implementen ciertos métodos. Sin embargo, una nueva pasarela de pagos que quieres integrar tiene una interfaz diferente a la que tu sistema espera.

Escenario:

Tienes una clase `ProcesadorDePagos` en tu plataforma de e-commerce que espera una interfaz de pasarela de pago llamada `PasarelaPago`. La interfaz de `PasarelaPago` tiene un método llamado `procesar_pago(cantidad)`. Sin embargo, la nueva pasarela, llamada `NuevaPasarelaPago`, tiene un método diferente, `realizar_transaccion(monto)` en lugar de `procesar_pago(cantidad)`.

Para que `ProcesadorDePagos` pueda usar `NuevaPasarelaPago` sin modificar su código, puedes implementar el patrón Adapter.

Implementación en Python

Aquí tienes una implementación simple en Python:

```python
# Interfaz de la pasarela de pago esperada
class PasarelaPago:
    def procesar_pago(self, cantidad):
        raise NotImplementedError("Este método debe ser implementado")

# Nueva pasarela de pago incompatible que queremos adaptar
class NuevaPasarelaPago:
    def realizar_transaccion(self, monto):
        return f"Transacción realizada por {monto} con NuevaPasarelaPago"

# Adaptador que adapta NuevaPasarelaPago a la interfaz PasarelaPago
class PasarelaAdapter(PasarelaPago):
    def __init__(self, nueva_pasarela):
        self.nueva_pasarela = nueva_pasarela

    def procesar_pago(self, cantidad):
        # Adaptamos el método al que espera el cliente
        return self.nueva_pasarela.realizar_transaccion(cantidad)

# Cliente que espera una interfaz PasarelaPago
class ProcesadorDePagos:
    def __init__(self, pasarela: PasarelaPago):
        self.pasarela = pasarela

    def ejecutar_pago(self, cantidad):
        print(self.pasarela.procesar_pago(cantidad))

# Uso del adaptador
nueva_pasarela = NuevaPasarelaPago()
adaptador = PasarelaAdapter(nueva_pasarela)
procesador = ProcesadorDePagos(adaptador)

# El ProcesadorDePagos ejecuta el pago usando la interfaz adaptada
procesador.ejecutar_pago(100)

```

Explicación

1. NuevaPasarelaPago solo tiene el método realizar_transaccion, que no es compatible con el sistema.
2. PasarelaAdapter actúa como intermediario, implementando el método procesar_pago que espera el cliente y adaptándolo para que llame internamente a realizar_transaccion.
3. ProcesadorDePagos sigue utilizando su método ejecutar_pago, sin saber que en realidad está usando NuevaPasarelaPago a través del adaptador.

Salida esperada

Al ejecutar procesador.ejecutar_pago(100), obtendrás:

```
PS C:\Users\daria> & C:/Users/daria/AppData/Local/Programs/Python/Python312/python.exe c:/Users/daria/OneDrive/Documentos/Adapter.py
Transacción realizada por 100 con NuevaPasarelaPago
PS C:\Users\daria>
```

Conclusión

El patrón Adapter es una solución ideal para resolver problemas de incompatibilidad entre interfaces, ya que permite integrar clases de manera flexible, reutilizando código existente sin necesidad de alterarlo.