PATRONES ESTRUCTURALES (ADAPTADOR CLASE Y OBJETO)

AGENDA

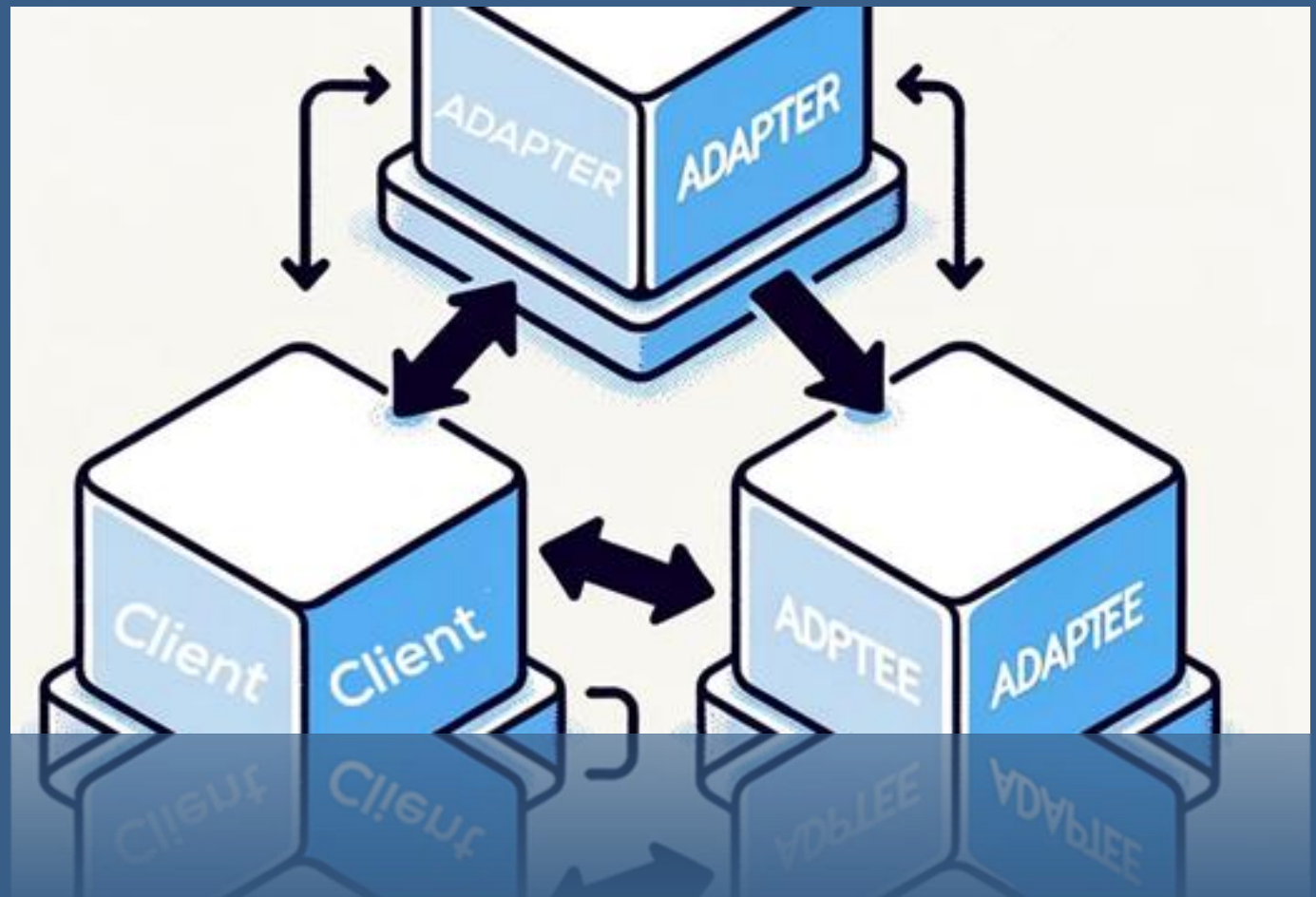
- Introducción
- Tipos de Adaptador
- Modelo/Diagramas de Clase
- Ejemplos

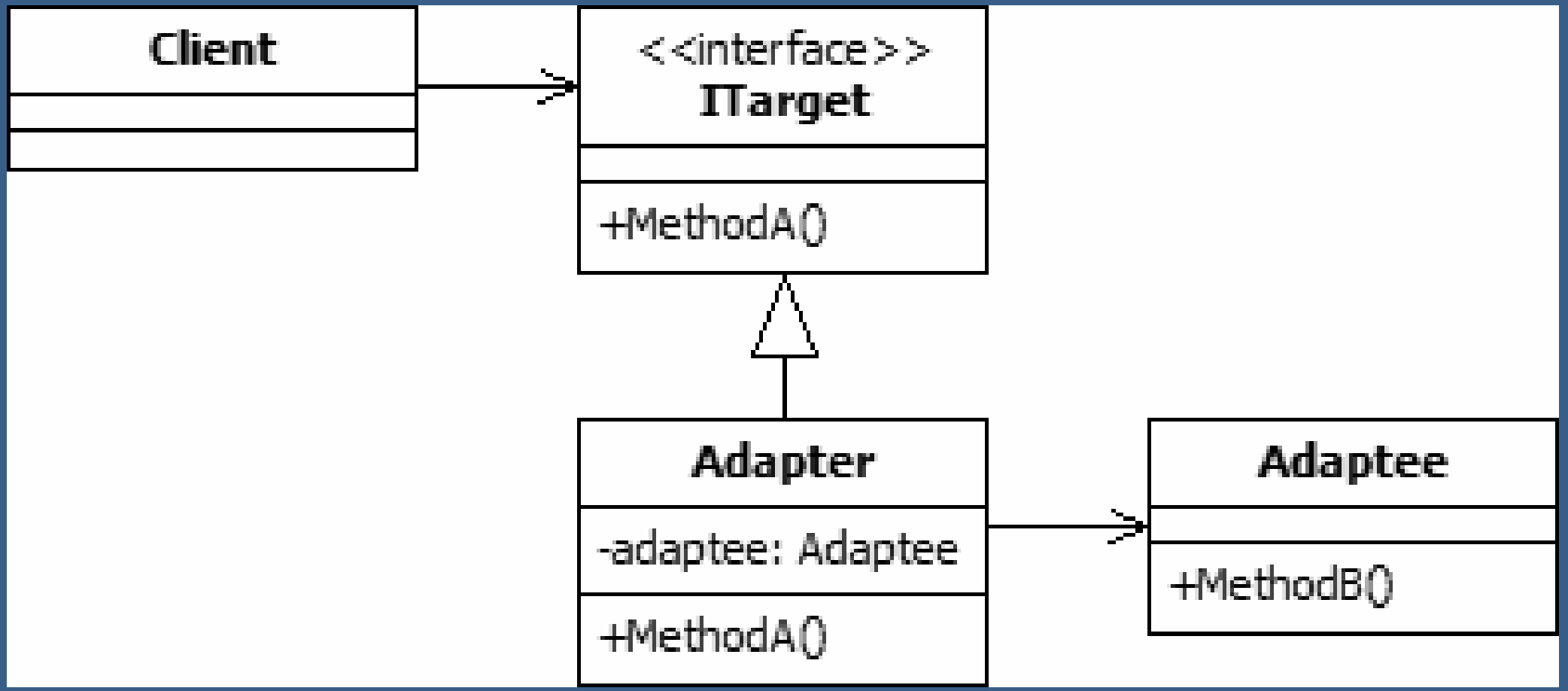
INTRODUCCIÓN AL ADAPTER PATRON

¿CUÁL ES EL PROPÓSITO DEL PATRÓN ADAPTADOR?

El “pattern adapter” o también llamado “wrapper” tiene el propósito de convertir una interfaz en otra interfaz esperada por el usuario/cliente

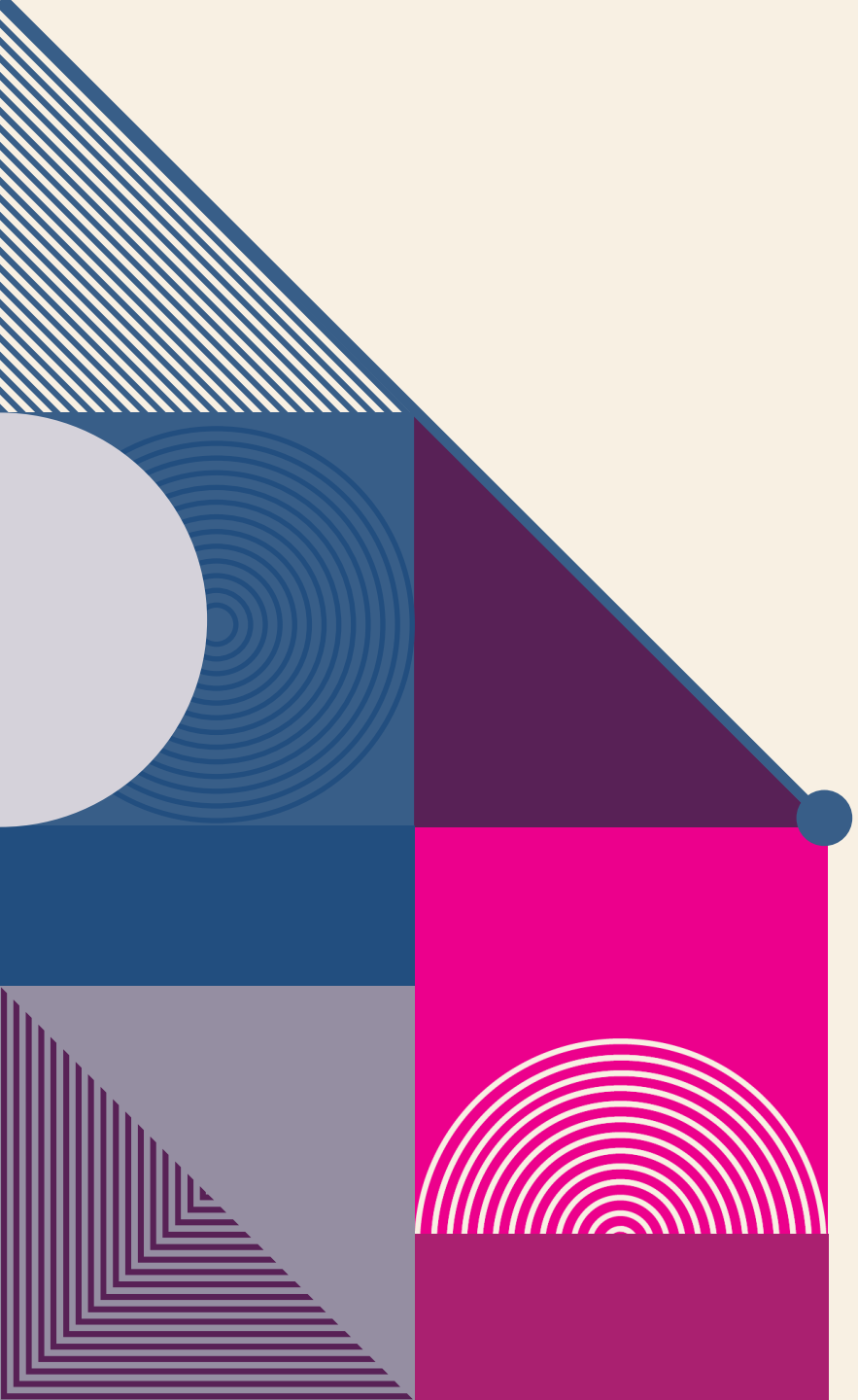
ADAPTER ENVUELVE LA FUNCIONALIDAD DE UNA CLASE Y LA TRADUCE A UNA INTERFAZ ESPERADA POR EL CLIENTE, SIMILAR A CÓMO UN ADAPTADOR DE CORRIENTE PERMITE CONECTAR DISPOSITIVOS CON ENCHUFES INCOMPATIBLES




Client
<<interface>>
ITarget
+MethodA()
Adapter
-adaptee: Adaptee
+MethodA()
Adaptee
+MethodB()

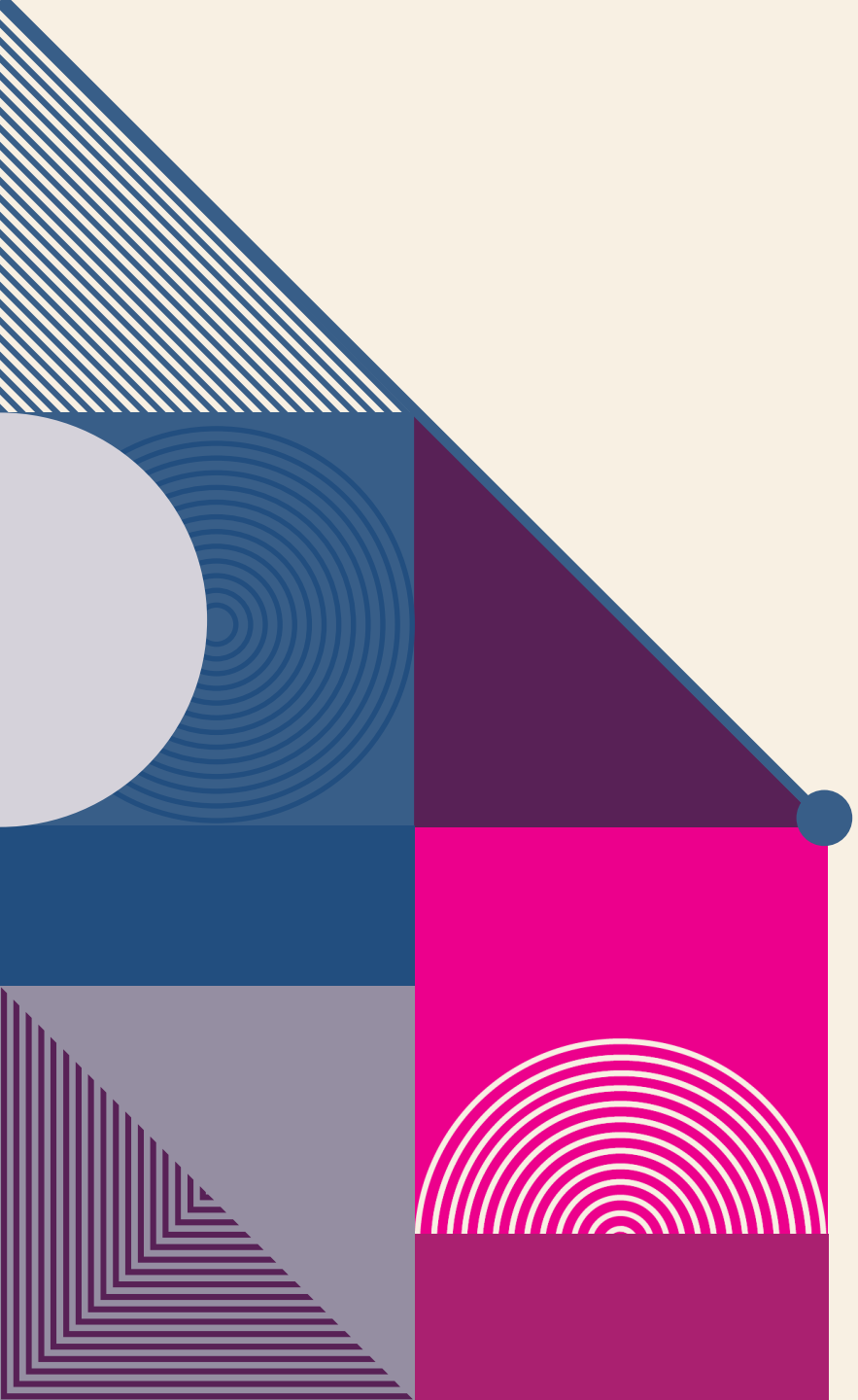
TIPOS DE ADAPTADOR

- Adaptador de clase
- Adaptador de objetos



ADAPTADOR DE CLASE

Usa herencia múltiple, es decir, el adaptador hereda del Target (interfaz esperada) y del Adaptee (clase existente). Es eficiente, pero depende de que el lenguaje permita herencia múltiple.



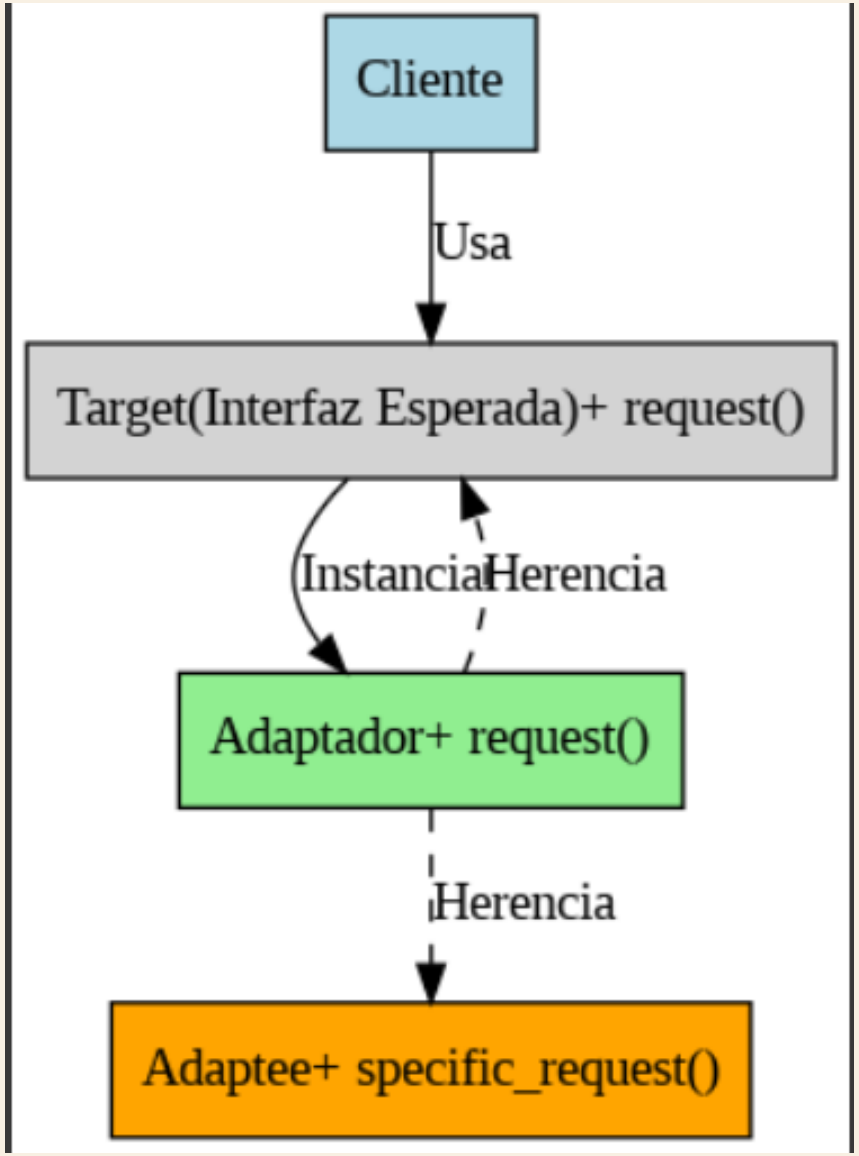
ADAPTADOR DE OBJETOS

Usa composición, lo que significa que el adaptador tiene una instancia del “Adaptee” dentro de él. Es más flexible, ya que puede cambiar el Adaptee en tiempo de ejecución y no depende de herencia múltiple.

DIAGRAMA DE CLASES

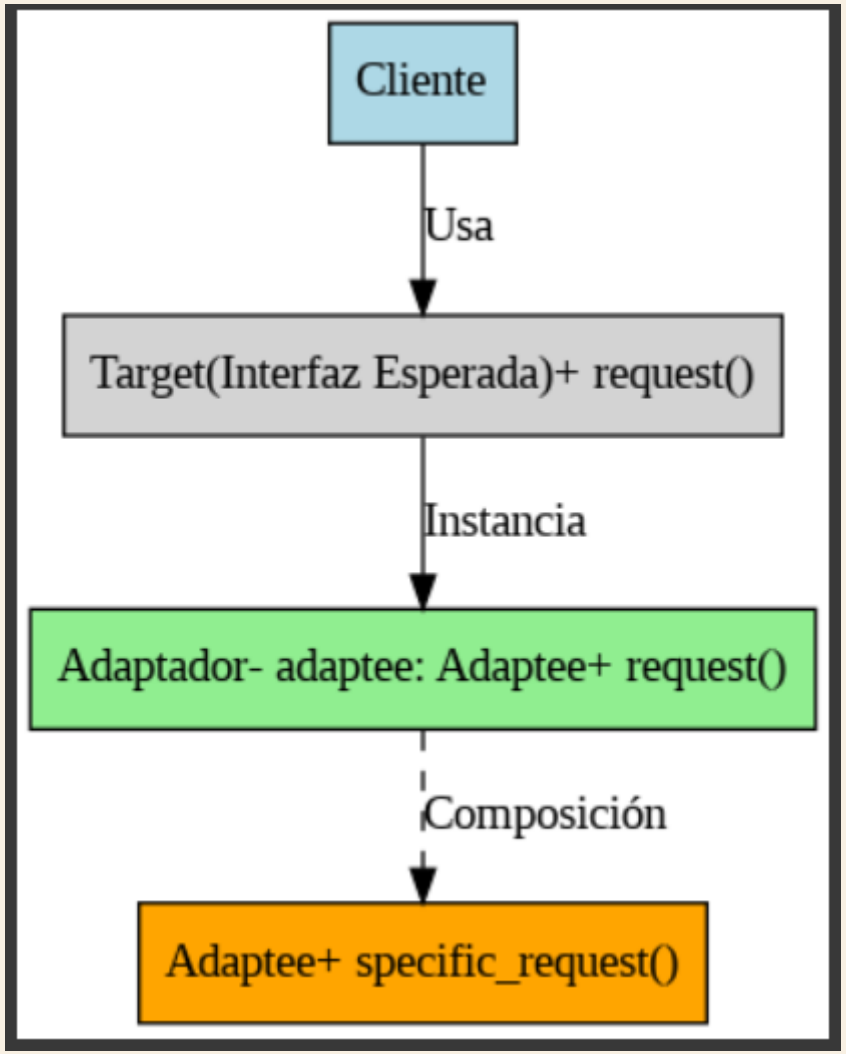
MODELO PARA EL ADAPTADOR DE CLASE

Se usa herencia para adaptar una clase existente (Adaptee) a una interfaz esperada (Target). El adaptador hereda tanto de Target como de Adaptee.



MODELO PARA EL ADAPTADOR DE OBJETO

Se usa composición, manteniendo una instancia de Adaptee dentro del adaptador, delegando las llamadas al método requerido
para cumplir con la interfaz Target.



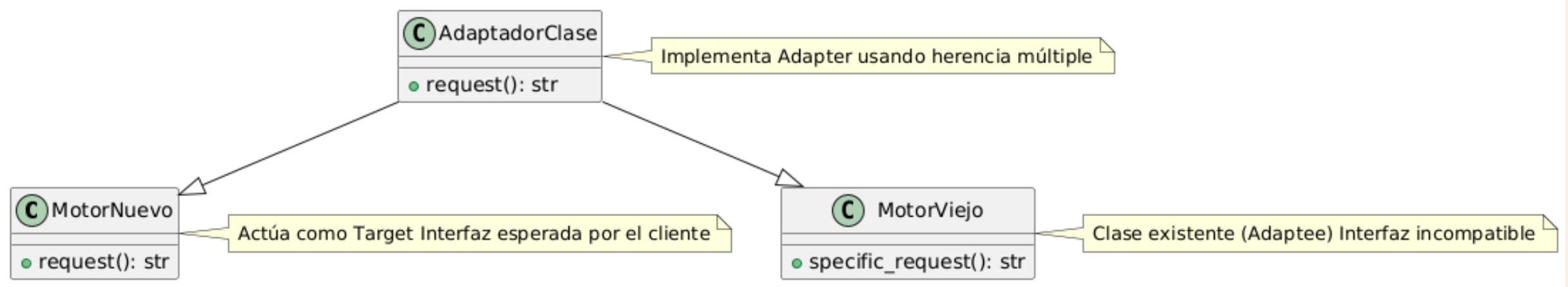
EJEMPLO DE CÓDIGO

```python
# La clase MotorNuevo actúa como la interfaz objetivo (Target).
# Define la interfaz que el cliente espera utilizar.
class MotorNuevo:
    def request(self):
        # Retorna una cadena que representa el funcionamiento estándar del motor nuevo.
        # No recibe parámetros.
        # No tiene restricciones.
        return "Motor Nuevo: Funcionamiento estándar"

# La clase MotorViejo representa una clase existente con una interfaz incompatible (Adaptee).
class MotorViejo:
    def specific_request(self):
        # Retorna una cadena invertida que representa un mensaje no legible directamente por el cliente.
        # No recibe parámetros.
        # No es compatible con la interfaz esperada por el cliente.
        return "etnega roloV"

# La clase AdaptadorClase implementa el patrón Adapter usando herencia múltiple.
# Hereda de MotorNuevo (Target) y MotorViejo (Adaptee) para traducir la interfaz incompatible.
class AdaptadorClase(MotorNuevo, MotorViejo):
    def request(self):
        # Adapta la respuesta del método specific_request del MotorViejo.
        # Llama al método original y transforma su salida invirtiendo la cadena.
        # Retorna una cadena legible para el cliente.
        resultado = self.specific_request()       # Obtiene la cadena invertida del Adaptee.
        resultado = resultado[::-1]               # Invierte la cadena para que tenga sentido.
        return f"Adaptador: (Traducido) {resultado}"

# Se instancia el adaptador de clase y se llama al método 'request' que el cliente espera.
cliente = AdaptadorClase()
print(cliente.request())  # Salida esperada: Adaptador: (Traducido) Volor agente
```


AdaptadorClase
request(): str
Implementa Adapter usando herencia múltiple
MotorNuevo
request(): str
Actúa como Target Interfaz esperada por el cliente
MotorViejo
specific_request(): str
Clase existente (Adaptee) Interfaz incompatible

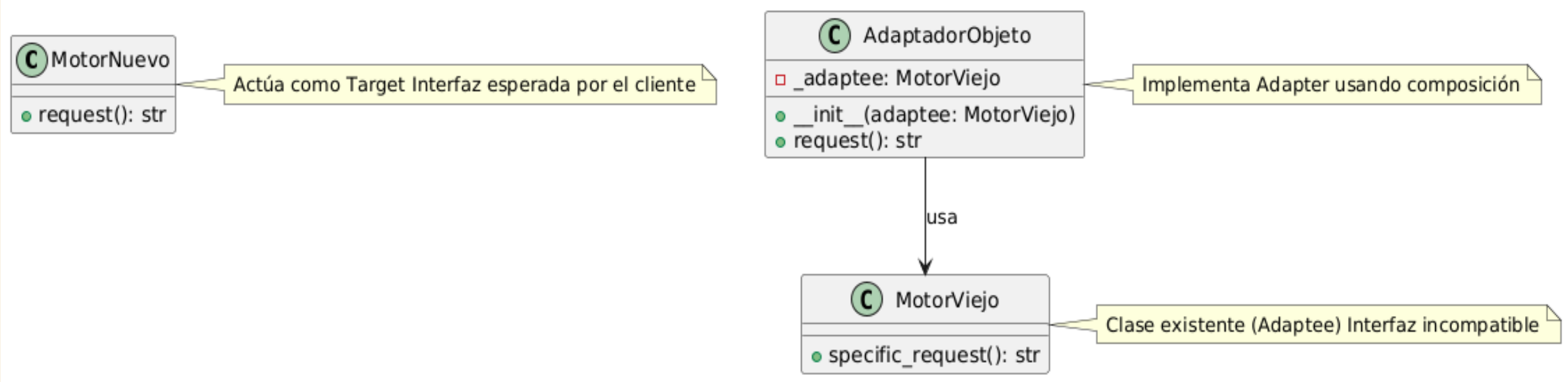
```python
# La clase MotorNuevo define la interfaz esperada por el cliente (Target).
class MotorNuevo:
    def request(self):
        # Retorna una cadena que representa el funcionamiento estándar del motor nuevo.
        return "Motor Nuevo: Funcionamiento estándar"

# La clase MotorViejo representa la clase existente con una interfaz no compatible (Adaptee).
class MotorViejo:
    def specific_request(self):
        # Retorna una cadena invertida que necesita ser adaptada para el cliente.
        return "etnega roloV"

# La clase AdaptadorObjeto implementa el patrón Adapter mediante composición.
# Recibe una instancia del Adaptee (MotorViejo) y traduce su interfaz a la esperada por el cliente.
class AdaptadorObjeto:
    def __init__(self, adaptee):
        # Recibe y almacena una instancia del Adaptee.
        # Se espera que el objeto pasado tenga el método specific_request.
        self._adaptee = adaptee

    def request(self):
        # Llama al método specific_request del Adaptee.
        # Invierte la cadena resultante para adaptarla al formato esperado.
        # Retorna una cadena legible por el cliente.
        resultado = self._adaptee.specific_request()  # Obtiene la cadena del adaptee.
        resultado = resultado[::-1]                   # La invierte para hacerla comprensible.
        return f"Adaptador: (Traducido) {resultado}"

# Se instancia el MotorViejo y luego se pasa al AdaptadorObjeto.
# El cliente llama a request, que ahora es compatible gracias al adaptador.
motor_viejo = MotorViejo()
adaptador = AdaptadorObjeto(motor_viejo)
print(adaptador.request())  # Salida esperada: Adaptador: (Traducido) Volor agente
```


MotorNuevo
request(): str
Actúa como Target Interfaz esperada por el cliente
AdaptadorObjeto
_adaptee: MotorViejo
__init__(adaptee: MotorViejo)
request(): str
Implementa Adapter usando composición
usa
MotorViejo
specific_request(): str
Clase existente (Adaptee) Interfaz incompatible

REFERENCIAS

- Adapter - Structural Pattern - Software-Pattern.org
- Adapter Design Pattern
- gomson/Design-Patterns: Design Patterns: Elements of Reusable Object-Oriented Software
- Design Patterns: Elements of Reusable Object-Oriented Software - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - Google Libros
- Design Patterns: Elements of Reusable Object Oriented Software

MUCHAS GRACIAS