Adapter Pattern

Tomás Felipe Melli

June 10, 2025

Índice

1 Intent

El **intent** es lo que define a un patrón. Entender esto nos va a permitir poder diferenciar uno de otro. En el caso del **Adapter** la intención es **convertir la interfaz de una clase en otra que el cliente espera. Permite que clases con interfaces incompatibles trabajen juntas.**

2 Motivation

Una aplicación puede necesitar usar clases existentes cuya interfaz no coincide con la requerida por el sistema. Por ejemplo, integrar una clase `TextView` dentro de un sistema gráfico que trabaja con la interfaz `Shape`. Un `TextShape` actúa como adaptador, permitiendo que `TextView` se utilice como si fuera una `Shape`.

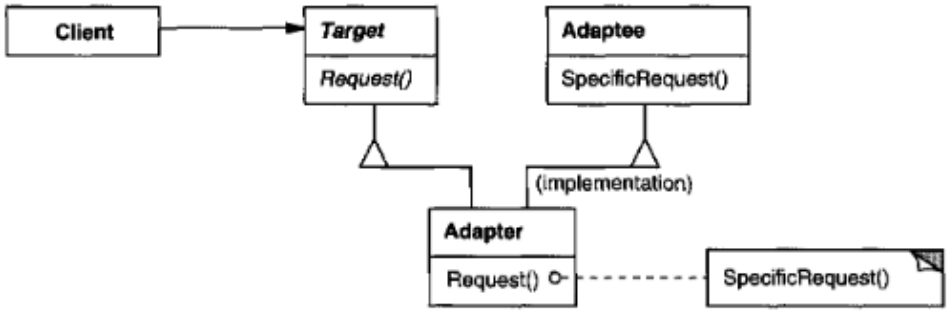
3 Applicability

Usar el patrón Adapter cuando:

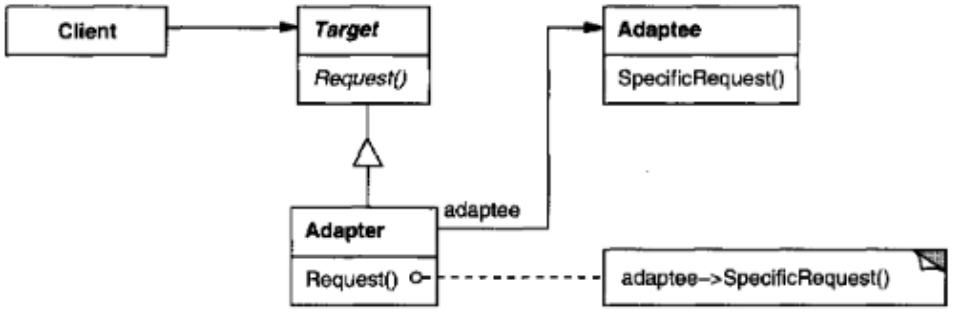
- Se quiere usar una clase existente pero su interfaz no coincide con la requerida.
- Se necesita reutilizar varias subclases existentes con interfaces incompatibles.

4 Structure

A class adapter uses multiple inheritance to adapt one interface to another:



An object adapter relies on object composition:



5 Participants

- **Target**: Interfaz esperada por el cliente (ej. `Shape`).
- **Client**: Utiliza objetos que siguen la interfaz Target (ej. `DrawingEditor`).
- **Adaptee**: Clase existente con interfaz incompatible (ej. `TextView`).
- **Adapter**: Convierte la interfaz del Adaptee en la del Target (ej. `TextShape`).

6 Example

Contexto del problema

El sistema tiene un editor gráfico que trabaja con figuras (`Shape`). Queremos reutilizar una clase existente llamada `TextView`, que muestra texto en pantalla, pero no implementa la interfaz requerida por `Shape`. No podemos modificar `TextView`.

Interfaz esperada: Shape

El editor espera que las figuras implementen la siguiente interfaz:

```
class Shape {
public:
    virtual void BoundingBox(Point& bottomLeft, Point& topRight) const = 0;
    virtual bool IsEmpty() const = 0;
    virtual Manipulator* CreateManipulator() const = 0;
    virtual ~Shape() = default;
};
```

`BoundingBox` devuelve el área ocupada. `IsEmpty` indica si hay contenido. `CreateManipulator` crea una herramienta para manipular la figura.

Clase existente: TextView

`TextView` tiene su propia interfaz, incompatible con `Shape`:

```
class TextView {
public:
    void GetOrigin(int& x, int& y) const;
    void GetExtent(int& width, int& height) const;
    bool IsEmpty() const;
};
```

Por ejemplo, no usa puntos ni retorna ‘BoundingBox‘ directamente.

Solución: Adapter TextShape

Creamos una clase `TextShape` que actúa como adaptador:

```
class TextShape : public Shape {
private:
    TextView* _text;

public:
    TextShape(TextView* t) : _text(t) {}

    void BoundingBox(Point& bottomLeft, Point& topRight) const override {
        int x, y, width, height;
        _text->GetOrigin(x, y);
        _text->GetExtent(width, height);
        bottomLeft = Point(x, y);
        topRight = Point(x + width, y + height);
    }

    bool IsEmpty() const override {
        return _text->IsEmpty();
    }

    Manipulator* CreateManipulator() const override {
        return new TextManipulator(this);
    }
};
```

Notas clave

- `TextShape` implementa la interfaz `Shape`.
- Usa composición: guarda internamente un puntero a `TextView`.
- Traduce llamadas desde la interfaz esperada hacia la clase adaptada.

Resultado

Ahora el editor puede trabajar con instancias de `TextShape` como si fueran figuras normales:

```
TextView* tv = new TextView();
Shape* shape = new TextShape(tv);

Point bl, tr;
shape->BoundingBox(bl, tr);
//   Funciona   como un Shape, aunque es un TextView adaptado!
```