DISEÑO DE SISTEMAS

ALUMNA: Gamarra, Jael.

DOCENTES: Pioli, Pablo; Ferreyra, Juan Pablo.

FECHA: 24-06.

AÑO: 2022.

CARRERA: INGENIERÍA EN SISTEMAS DE INFORMACIÓN.

UTN

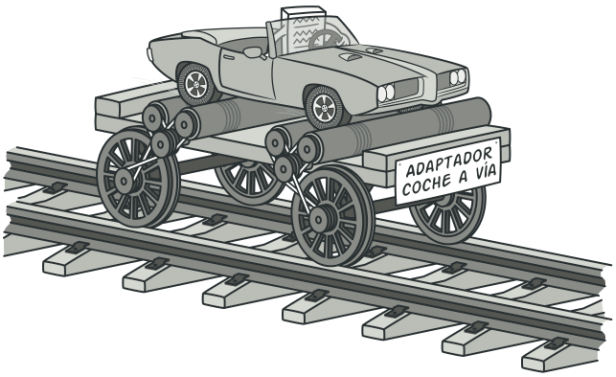
FACULTAD REGIONAL SAN FRANCISCO

ÍNDICE

ADAPTER

¿Qué es? ¿Para qué y cuándo se utiliza?

Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles, de tal modo que una clase que no pueda utilizar la primera haga uso de ella a través de la segunda. Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes.



También es conocido como Wrapper (Envoltorio).

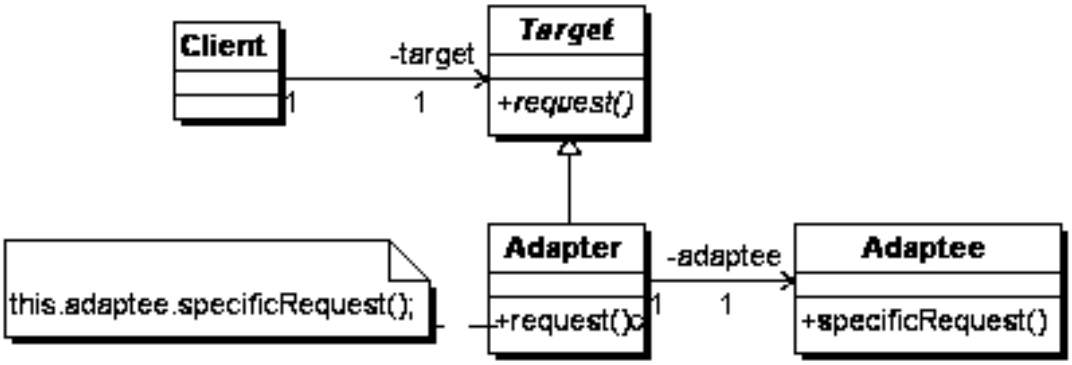
Es recomendable utilizar el patrón adaptador cuando:

- Se desea usar una clase existente, y su interfaz no sea igual a la necesitada.
- Cuando se desea crear una clase reutilizable que coopere con clases no relacionadas. Es decir, que las clases no tienen necesariamente interfaces compatibles.
- *(Solamente en el caso de un adaptador de objetos)* es necesario usar varias subclases existentes, pero no resulta práctico adaptar su interfaz heredando de cada una de ellas. Un adaptador de objeto puede adaptar la interfaz de su clase padre.

El mundo real está lleno de adaptadores. Por ejemplo, consideremos un adaptador USB a Ethernet. Necesitamos esto cuando tenemos una interfaz Ethernet en un extremo y USB en el otro. Ya que son incompatibles entre sí, usamos un adaptador que convierte uno a otro.

¿Cuáles son las características de su estructura?

Un adaptador de clases usa la herencia múltiple para adaptar una interfaz a otra. De manera gráfica la podemos representar de la siguiente manera:



Un adaptador de objetos se basa en la composición de objetos.

Participantes:

- **Target/Objetivo (Forma):** La cual se encarga de definir la interfaz específica del dominio que *Client* usa.
- **Client (Editor de dibujo):** Colabora con la conformación de objetos para la interfaz *Target*.
- **Adaptee (VistaTexto):** Define una interfaz existente que necesita adaptarse.
- **Adapter (FormaTexto):** Adapta la interfaz de *Adaptee* a la interfaz *Target*.

Client llama a las operaciones sobre una instancia *Adapter*. De hecho, el adaptador llama a las operaciones de *Adaptee* que llevan a cabo el pedido.

¿Cuáles son sus ventajas?

Entre sus ventajas encontramos la posibilidad que brinda de hacer que dos interfaces incompatibles, sean compatibles. Puede servir para encapsular clases que no controlamos, y que pueden cambiar.

Un adaptador de clases

- Adapta una clase Adaptable a Objetivo, pero se refiere únicamente a una clase Adaptable concreta. Por tanto, un adaptador de clases no nos servirá cuando lo que queremos es adaptar una clase y todas sus subclases.
- Permite que Adaptador redefina parte del comportamiento de Adaptable, por ser Adaptador una subclase de Adaptable.
- Introduce un solo objeto, y no se necesita ningún puntero de indirección adicional para obtener el objeto adaptado.

Por su parte, un adaptador de objetos

- Permite que un mismo adaptador funcione con mucho adaptables, es decir, con el Adaptable en sí y todas sus subclases, en caso de que las tenga. El adaptador también puede añadir funcionalidad a todos los Adaptables a la vez.
- Hace que sea más difícil redefinir el comportamiento de Adaptable. Se necesitará crear una subclase de Adaptable y hacer que el Adaptador se refiere a la subclase en vez de a la clase Adaptable en sí.

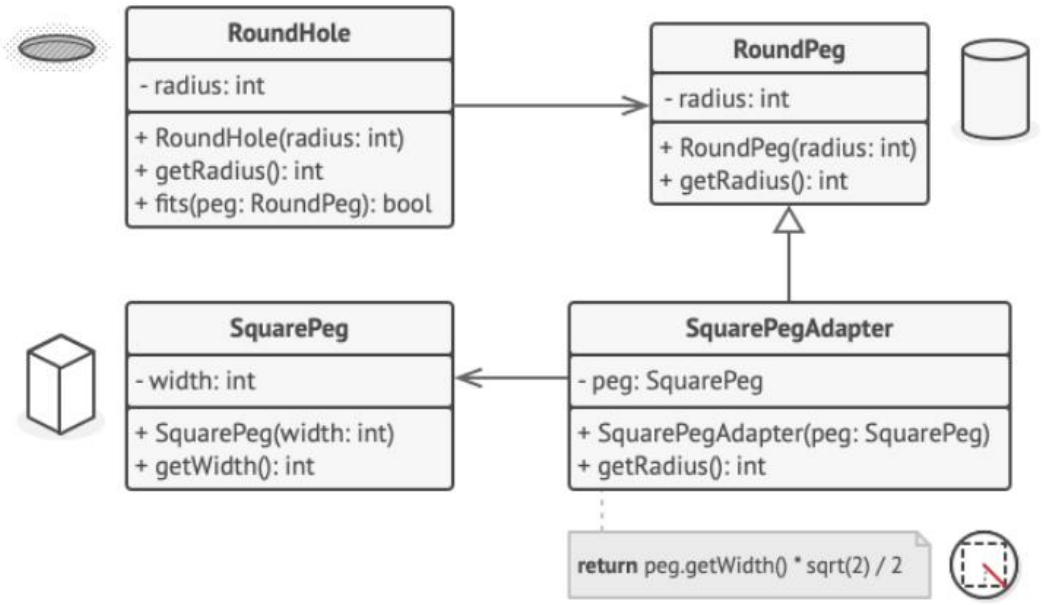
¿Y desventajas?

Como muchos patrones, añade complejidad al diseño. Hay quien dice que este patrón es un parche, utilizado en malos diseños.

La complejidad general del código aumenta, ya que debes introducir un grupo de nuevas interfaces y clases. En ocasiones resulta más sencillo cambiar la clase de servicio de modo que coincida con el resto del código.

Ejemplo-Pseudocódigo

1) Este ejemplo del patrón Adapter se basa en el clásico conflicto entre piezas cuadradas y agujeros redondos.



Adaptando piezas cuadradas a agujeros redondos.

El patrón Adapter finge ser una pieza redonda con un radio igual a la mitad del diámetro del cuadrado (en otras palabras, el radio del círculo más pequeño en el que quepa la pieza cuadrada).

```
// Digamos que tienes dos clases con interfaces compatibles:
// RoundHole (HoyoRedondo) y RoundPeg (PiezaRedonda).
class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Devuelve el radio del agujero.

    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()

class RoundPeg is
    constructor RoundPeg(radius) { ... }

    method getRadius() is
        // Devuelve el radio de la pieza.


// Pero hay una clase incompatible: SquarePeg (PiezaCuadrada).
class SquarePeg is
    constructor SquarePeg(width) { ... }
```

```
    method getWidth() is
        // Devuelve la anchura de la pieza cuadrada.


// Una clase adaptadora te permite encajar piezas cuadradas en
// hoyos redondos. Extiende la clase RoundPeg para permitir a
// los objetos adaptadores actuar como piezas redondas.
class SquarePegAdapter extends RoundPeg is
    // En realidad, el adaptador contiene una instancia de la
    // clase SquarePeg.
    private field peg: SquarePeg

    constructor SquarePegAdapter(peg: SquarePeg) is
        this.peg = peg

    method getRadius() is
        // El adaptador simula que es una pieza redonda con un
        // radio que pueda albergar la pieza cuadrada que el
        // adaptador envuelve.
        return peg.getWidth() * Math.sqrt(2) / 2


// En algún punto del código cliente.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // verdadero

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // esto no compila (tipos incompatibles)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // verdadero
hole.fits(large_sqpeg_adapter) // falso
```

2) Otro ejemplo:

Suponga que tiene una clase Bird con los métodos fly() y makeSound(). Y también una clase ToyDuck con el método squeak8(). Supongamos que le faltan objetos ToyDuck y le gustaría usar objetos Bird en su lugar. Los pájaros tienen una funcionalidad similar, pero implementan una interfaz diferente por lo que no podemos usarlos directamente. Así que

usaremos el patrón adaptador. Aquí nuestro cliente sería ToyDuck y el adaptado sería Bird.

```
interface Bird
{
    // birds implement Bird interface that allows
    // them to fly and make sounds adaptee interface
    public void fly();
    public void makeSound();
}

class Sparrow implements Bird
{
    // a concrete implementation of bird
    public void fly()
    {
        System.out.println("Flying");
    }
    public void makeSound()
    {
        System.out.println("Chirp Chirp");
    }
}

interface ToyDuck
{
    // target interface
    // toyducks dont fly they just make
    // squeaking sound
    public void squeak();
}

class PlasticToyDuck implements ToyDuck
{
    public void squeak()
    {
        System.out.println("Squeak");
    }
}

class BirdAdapter implements ToyDuck
{
    // You need to implement the interface your
    // client expects to use.
    Bird bird;
```

```
    public BirdAdapter(Bird bird)
    {
        // we need reference to the object we
        // are adapting
        this.bird = bird;
    }

    public void squeak()
    {
        // translate the methods appropriately
        bird.makeSound();
    }
}

class Main
{
    public static void main(String args[])
    {
        Sparrow sparrow = new Sparrow();
        ToyDuck toyDuck = new PlasticToyDuck();

        // Wrap a bird in a birdAdapter so that it
        // behaves like toy duck
        ToyDuck birdAdapter = new BirdAdapter(sparrow);

        System.out.println("Sparrow...");
        sparrow.fly();
        sparrow.makeSound();

        System.out.println("ToyDuck...");
        toyDuck.squeak();

        // toy duck behaving like a bird
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
    }
}
```

BIBLIOGRAFÍA

https://www.genbeta.com/desarrollo/patrones-de-diseno-adapter#:~:text=Ventajas%3A%20hace%20que%20dos%20interfaces,parche%2C%20utilizado%20en%20malos%20dise%C3%B1os.

https://es.wikipedia.org/wiki/Adaptador_(patr%C3%B3n_de_dise%C3%B1o).

https://www.geeksforgeeks.org/adapter-pattern/.

https://refactoring.guru/es/design-patterns/adapter.