Patrones estructurales

Adapter

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"
[Christopher Alexander].

Por: Brayan Estiven Aguirre Aristizábal – 20231020156
Santiago Zamudio Diaz – 20231020128
Daniel Josué Rozo Rodríguez – 20222020006

1. Introducción

Los patrones estructurales describen como los objetos y las clases pueden ser combinados para formar estructuras más grandes y complejas, es decir, tratan de conseguir que cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos. Lo fundamental son las relaciones de uso entre los objetos, y, éstas están determinadas por las interfaces que soportan los objetos. Los patrones estructurales estudian como se relacionan los objetos en tiempo de ejecución. Sirven para diseñar las interconexiones entre los objetos. En este caso presentaremos el patrón Adapter y sus principales características.

2. Adapter

Nombre del patrón:
Adapter

Clasificación del patrón:

Estructural

Intención:

Es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

También conocido por:

Simplified Interface, Adaptador, Wrapper

Motivación:

El patrón estructural adapter convierte la interfaz de una clase en otra distinta que es la que esperan los clientes, permitiendo que cooperen clases que de otra manera no podrían por tener interfaces incompatibles. Este patrón se centra en resolver incompatibilidades entre dos interfaces existentes. Se utiliza para hacer funcionar las cosas después de que estas han sido diseñadas.

Aplicabilidad:

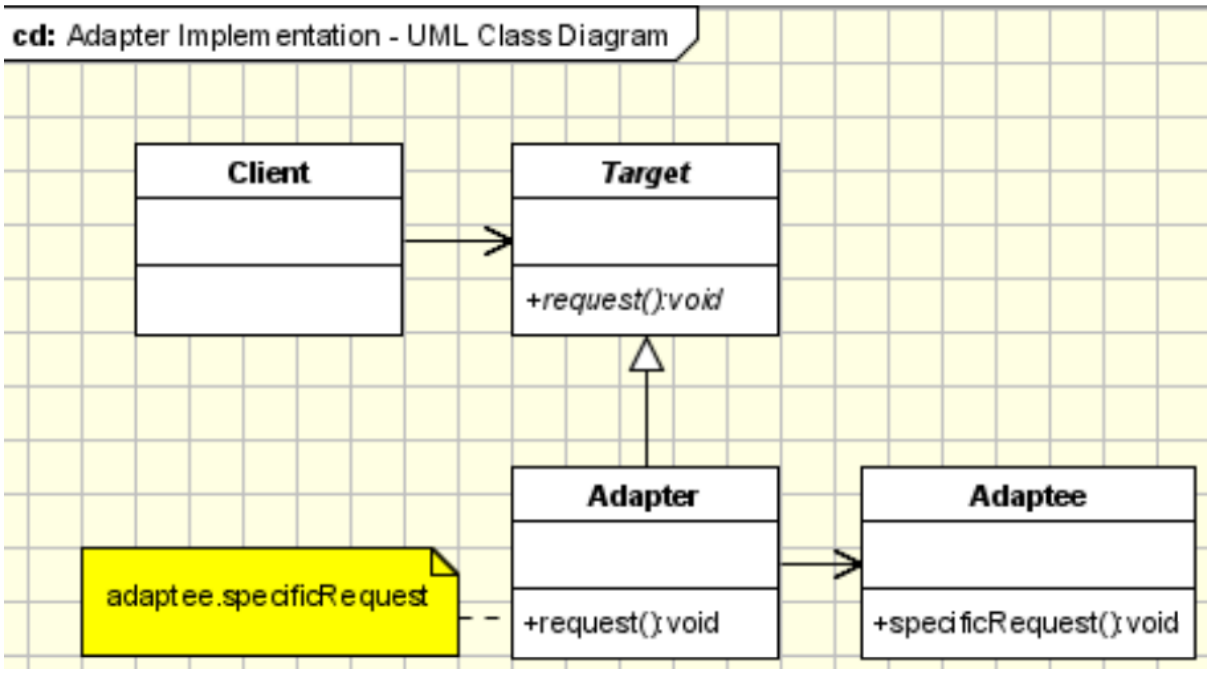
Este patrón puede ser usado cuando

• Se quiera utilizar una clase existente, y ésta no tenga el tipo de interfaz que necesitamos.

• Cuando se quiera crear una clase reutilizable que coopere con clases con las que no está relacionada, y que por tanto no tengan interfaces compatibles.

• Cuando se necesite usar varias subclases existentes, pero sin tener que adaptar su interfaz creando una nueva subclase de cada una.

Estructura:



Participantes:

- **Target:** define la interfaz específica del dominio (es la que usarán los clientes).
- **Client:** Colabora con objetos de tipo Target.
- **Adaptee:** La clase existente cuya interfaz necesita ser adaptada.
- **Adapter:** Adapta la interfaz de Adaptee a la de Target.

Colaboraciones:

- El cliente trabaja con objetos a través de la interfaz Target. El cliente no sabe si está trabajando con un objeto adaptador o un objeto que implementa la interfaz Target directamente.
- El adaptador contiene una instancia del Adaptee y traduce las llamadas de la interfaz Target a la interfaz Adaptee. La llamada del cliente al método Target se convierte en una llamada al método correspondiente en el Adaptee.
- El adaptador implementa la interfaz Target para que el cliente la use.

Consecuencias:

- Puedes separar la interfaz o el código de conversión de datos de la lógica de negocio primaria del programa.
- Puedes introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente.
- 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 de tu código.

Implementación:

- Lo podemos implementar tomando la clase adaptadora cuando queramos usar una clase existente, pero cuya interfaz no sea compatible con el resto del código.
- Utiliza el patrón cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase. Puedes extender cada subclase y colocar la funcionalidad que falta, dentro de las nuevas clases hijas. No obstante, deberás duplicar el código en todas estas nuevas clases.
- Otra forma de implementar de manera más fácil sería colocar la funcionalidad que falta dentro de una clase adaptadora. Después envolver objetos a los que les falten funciones, dentro de la clase adaptadora, obteniendo esas funciones necesarias de un modo dinámico.

Código ejemplo:

En este ejemplo se va a encajar piezas cuadradas en agujeros redondos, mostrando de manera sencilla el modo en que un Adapter puede hacer que objetos incompatibles trabajen juntos. “Peg” significa clavija o tornillo.

```java
package round;

public class RoundHole {
    private final double radius;

    public RoundHole(double radius){
        this.radius = radius;
    }
    public double getRadius(){
        return radius;
    }
    public boolean fits(RoundPeg peg){
        boolean result;
        result = (this.getRadius() >= peg.getRadius());
        return result;
    }
}
```

```java
package round;

public class RoundPeg {
    private double radius;

    public RoundPeg() {};

    public RoundPeg(double radius){
        this.radius = radius;
    }
    public double getRadius(){
        return radius;
    }
}
```

```java
package square;

public class SquarePeg {
    private final double width;

    public SquarePeg(double width){
        this.width = width;
    }
    public double getwidth(){
        return width;
    }
    public double getSquare(){
        double result;
        result = Math.pow( a: this.width,  b: 2);
        return result;
    }
}
```

```java
package adapters;
import round.RoundPeg;
import square.SquarePeg;

public class SquarePegAdapter extends RoundPeg{
    private final SquarePeg peg;

    public SquarePegAdapter(SquarePeg peg){
        this.peg = peg;
    }
    @Override
    public double getRadius(){
        double result;
        result = (Math.sqrt(Math.pow((peg.getwidth() / 2), b:2) * 2));
        return result;
    }
}
```

```java
/*
Modelos de programación I - Marcela Espinosa
Patrón Adapter
*/

import adapters.SquarePegAdapter;
import round.RoundHole;
import round.RoundPeg;
import square.SquarePeg;

public class Demo {
    public static void main(String[] args){
        RoundHole hole = new RoundHole( radius: 5);
        RoundPeg rpeg = new RoundPeg( radius: 5);
        if(hole.fits( peg: rpeg)){
            System.out.println( x: "La clavija redonda r5 encaja en el agujero redondo r5.");
        }

        SquarePeg smallSqPeg = new SquarePeg( width: 2);
        SquarePeg largeSqPeg = new SquarePeg( width: 20);

        SquarePegAdapter smallSqPegAdapter = new SquarePegAdapter( peg: smallSqPeg);
        SquarePegAdapter largeSqPegAdapter = new SquarePegAdapter( peg: largeSqPeg);
        if(hole.fits( peg: smallSqPegAdapter)){
            System.out.println( x: "La clavija cuadrada w2 encaja en el orificio redondo r5.");
        }
        if(!hole.fits( peg: largeSqPegAdapter)){
            System.out.println( x: "La clavija cuadrada w20 no encaja en el orificio redondo r5.");
        }
    }
}
```

Usos conocidos:

- Muy usado en microservicios, APIS, plataformas en la nube, sistemas de mensajería, colas, frameworks y bibliotecas de desarrollo web.

Patrones relacionados:

- Tiene procedimientos muy similares al patrón **Decorator**, así como una estructura muy similar a la del **Bridge**, ya que ambos patrones se basan en la composición, delegando el trabajo a otros objetos; lo que cambia es el tipo de problema que solucionan como tal.

3. Referencias Bibliográficas

[1] Refactoring Guru. (s.f). Patrónes estructurales: Adapter. Refactoring Guru. https://refactoring.guru/es/design-patterns/adapter

[2] Design Patterns. Elements of Reusable Object-Oriented Software - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - Addison Wesley (GoF- Gang of Four)