TRABAJO PRÁCTICO
PATRONES

DISEÑO DE SISTEMAS DE INFORMACIÓN

Alumna: Karlen, Lucía

Docentes: Ferreyra, Juan Pablo

Pioli, Pablo

Año: 2023

ÍNDICE

PATRONES DE DISEÑO

Los patrones de diseño (design patterns) son soluciones habituales a problemas comunes en el diseño de software. Cada patrón es como un plano que se puede personalizar para resolver un problema de diseño particular de tu código.

Los patrones de diseño varían en su complejidad, nivel de detalle y escala de aplicabilidad. Además, pueden clasificarse por su propósito y dividirse en tres grupos: creacionales, estructurales y de comportamiento.

PATRONES ESTRUCTURALES

Explican cómo ensamblar objetos y clases en estructuras más grandes, mientras se mantiene la flexibilidad y eficiencia de la estructura.

Son: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

PATRÓN ADAPTER

Es una herramienta en la programación que permite que objetos con interfaces incompatibles trabajen juntos de manera armoniosa. Es como un traductor que ayuda a que dos cosas diferentes puedan comunicarse sin problemas.

Se trata de un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

Envuelve uno de los objetos y el objeto envuelto no es consciente de la existencia del adaptador. Los adaptadores además de convertir datos en varios formatos ayudan a objetos con distintas interfaces a colaborar.

Funcionamiento

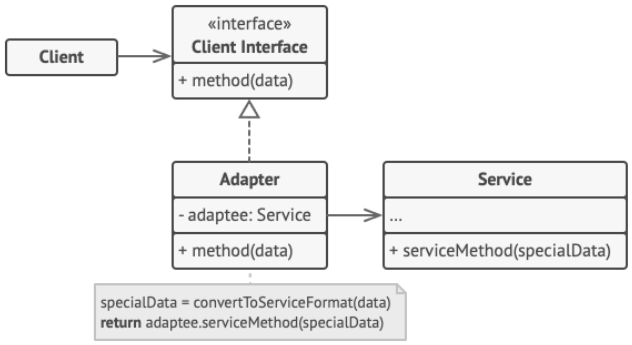
1) El adaptador obtiene una interfaz compatible con uno de los objetos existentes.
2) Utilizando esta interfaz, el objeto existente puede invocar con seguridad los métodos del adaptador.
3) Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que ese segundo objeto espera.

Se puede crear un adaptador bidireccional que pueda convertir las llamadas en ambos sentidos.

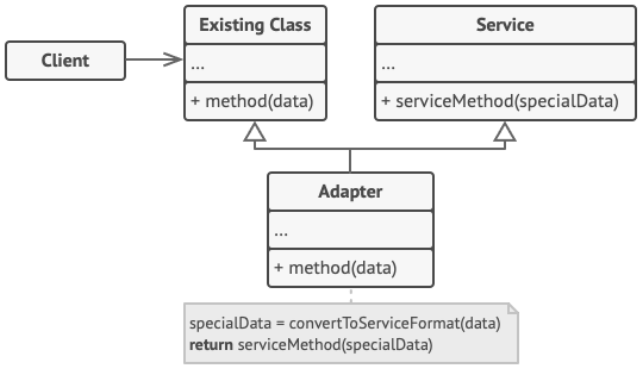
Estructura

- **Adaptador de objetos**: usa el principio de composición de objetos, el adaptador usa la interfaz de un objeto y envuelve el otro. Puede implementarse en todos los lenguajes de programación populares.
  - Cliente: tiene la lógica de negocio del programa
  - Interfaz con el cliente: describe un protocolo que otras clases deben seguir para poder colaborar con el código cliente.
  - Servicio: es alguna clase útil (de una tercera parte o heredada). El cliente no puede utilizar directamente esta clase porque tiene una interfaz incompatible.

- Clase adaptadora: puede trabajar con la clase cliente y con la de servicio: implementa la interfaz con el cliente, mientras envuelve el objeto de la clase de servicio. Recibe llamadas del cliente a través de la interfaz de cliente y las traduce en llamadas al objeto envuelto de la clase de servicio, en un formato que pueda comprender.
- El código cliente no se acopla a la clase adaptadora concreta siempre y cuando funcione con la clase adaptadora a través de la interfaz con el cliente. Gracias a esto, se pueden introducir nuevos adaptadores en el programa sin descomponer el código cliente existente. Esto es útil cuando la interfaz de la clase de servicio se cambia o sustituye, ya que puedes crear una nueva clase adaptadora sin cambiar el código cliente.



- **Clase adaptadora**: usa la herencia, porque la clase adaptadora hereda interfaces de ambos objetos al mismo tiempo. Puede implementarse en lenguajes de programación que soporten la herencia múltiple, como C++.
  - La Clase adaptadora no necesita envolver objetos porque hereda comportamientos de la clase cliente y de la clase de servicio. La adaptación tiene lugar dentro de los métodos sobrescritos. La clase adaptadora resultante puede usarse en lugar de una clase cliente existente.



Aplicabilidad

- Usa la clase adaptadora cuando quieras usar una clase existente, pero su interfaz no sea compatible con el resto del código: el patrón te permite crear

una clase intermedia que sea traductora entre tu código y una clase heredada, una clase de un tercero o cualquier otra clase con una interfaz extraña.

- Usa el patrón cuando quieras reutilizar varias subclases existentes que no tengan alguna funcionalidad común que no pueda añadirse a la superclase: una solución elegante 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. Para que esto funcione, las clases en cuestión deben tener una interfaz común y el campo de la clase adaptadora debe seguir dicha interfaz. Este procedimiento es similar al del patrón Decorator.

Ventajas y desventajas

- Ventajas:
  - Principio de responsabilidad única: puedes separar la interfaz o el código de conversión de datos de la lógica de negocio primaria del programa.
  - Principio de abierto/cerrado: 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.
- Desventajas:
  - La complejidad general del código aumenta, ya que debes introducir un grupo de nuevas interfaces y clases.

Código

```python
class ARGPlugConectorInterface:
    def give_electricity(self):
        pass

class ARGPlugConector (ARGPlugConectorInterface):
    def give_electricity(self):
        print("This is an ARG plug")

class ARGElectricalSocket:
    def plug_in(self, arg_plug):
        arg_plug.give_electricity()

class UKPlugConectorInterface:
    def provide_electricity(self):
        pass

class UKElectricalSocket:
    def plug_in(self, uk_plug):
        print("This is a UK electrical socket")
        uk_plug.provide_electricity()
```

```python
class ARGtoUKPlugAdapter(UKPlugConectorInterface):
    def __init__(self, arg_plug):
        self.arg_plug = arg_plug

    def provide_electricity(self):
        self.arg_plug.give_electricity()

def main ():
    arg_plug = ARGPlugConector()
    uk_electrical_socket = UKElectricalSocket()
    uk_adapter = ARGtoUKPlugAdapter(arg_plug)
    uk_electrical_socket.plug_in(uk_adapter)

if __name__ == "__main__":
    main()
```

```
This is a UK electrical socket
This is an ARG plug
```

Suponemos que nos vamos de Argentina (ARG) a Reino Unido (UK) y queremos conectar el enchufe argentino (ARGPlug) al tomacorriente de UK

(UKElectricalSocket), el enchufe no se podrá conectar porque no tienen la misma forma para conectarse correctamente. Para ello vamos a necesitar un adaptador.

En el código tenemos nuestro enchufe argentino `class ARGPlugConector (ARGPlugConectorInterface)`, que usa de la interfaz del enchufe argentino la función `def give_electricity(self):` (dar electricidad). Luego tenemos el tomacorriente de UK, que usa la función `def plug_in(self, uk_plug):` (permitir enchufarse) y dentro de ésta función se usa la función `def provide_electricity(self):` (dar electricidad), que es del enchufe de UK. Si bien give y provide son sinónimos, es para diferenciar que la función plug_in solo acepta la función del enchufe de UK, y solo permitirá que se conecte el enchufe de UK.

Entonces necesitaremos un adaptador que permita envolver el enchufe de Argentina, y mientras usar la interfaz de un enchufe de UK para que el tomacorriente de UK permita recibir el enchufe de Argentina, que es `class ARGtoUKPlugAdapter(UKPlugConectorInterface):`.

En resumen, crearemos un enchufe de argentina `arg_plug = ARGPlugConector()`, crearemos un tomacorriente de reino unido `uk_electrical_socket = UKElectricalSocket()`, usamos el adaptador, mandando internamente a la clase del adaptador el enchufe de Argentina y en realidad está recibiendo la interfaz de un enchufe de UK `uk_adapter = ARGtoUKPlugAdapter(arg_plug)`. El enchufe de Argentina va a usar su función give_electricity() como si en realidad estuviera conectado, y se va a guardar dentro de la función provide_electricity() que es la que realmente acepta el tomacorriente de UK. Luego se conecta el enchufe al tomacorriente como si fuese un enchufe de UK `uk_electrical_socket.plug_in(uk_adapter)`.

Lo que devuelve es ilustrativo, para ver cómo se usa el adaptador.

El diagrama de clases se vería de la siguiente manera:

- Cliente: `def main ():`
- Interfaz del cliente: `class ARGPlugConectorInterface:`
- Servicio: `class UKElectricalSocket:`
- Clase adaptadora: `class ARGtoUKPlugAdapter(UKPlugConectorInterface):`