2022

Patrón de Diseño - Proxy

DISEÑO DE SISTEMAS

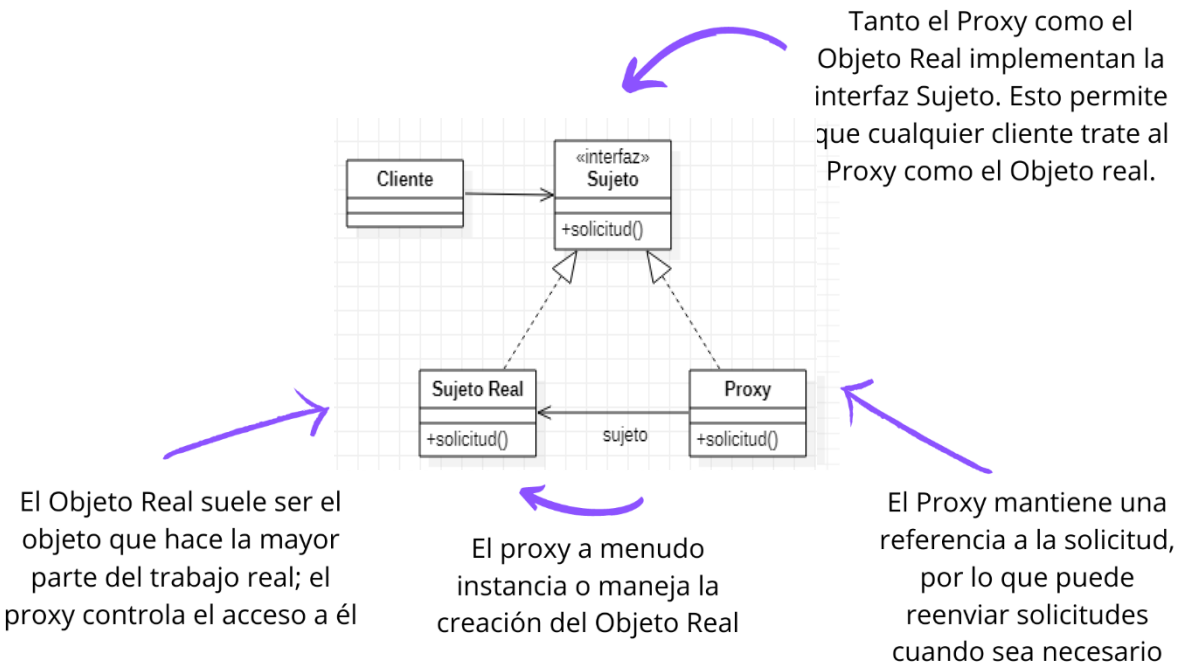
AGOSTINA BERTONI

PATRÓN DE DISEÑO - PROXY

Un proxy es un patrón de diseño estructural el cual centra su atención en la mediación entre un objeto y otro, el mismo, controla el acceso al objeto original, permitiéndole realizar ciertas acciones antes y/o después de realizar la acción deseada por el usuario.

Este patrón se caracteriza porque el cliente no es consciente de la implementación del proxy porque tanto el proxy como el objeto real implementan la misma interfaz.

El uso del proxy puede ser simplemente reenvío al objeto real, o puede proporcionar lógica adicional, por ejemplo, el almacenamiento en caché cuando las operaciones en el objeto real requieren muchos recursos, o la comprobación de las condiciones previas antes de que se invoquen las operaciones en el objeto real.



La forma en que trabaja un proxy:

1. El cliente solicita a la interfaz un Sujeto Real.
2. Se crea un Proxy que encapsule al Sujeto Real.
3. El cliente ejecuta el Proxy sin saberlo.
4. El Proxy realiza una o varias acciones previas a la ejecución del Sujeto Real.

5. El Proxy delega la ejecución al Sujeto Real.
6. El Proxy realiza una o varias acciones después de la ejecución del Sujeto Real.
7. El Proxy regresa un resultado.

Dependiendo de la función que se desea realizar con dicha referencia podemos distinguir diferentes tipos de *proxies*:

- Remoto: hace de intermediario entre las conexiones de un cliente y un servidor de destino, filtrando todos los paquetes entre ambos. Por lo tanto, este oculta el hecho de que un objeto reside en otro espacio de direcciones
- Virtual: puede realizar optimizaciones, como la creación de objetos de bajo demanda, también puede hacer caché de información del objeto real para diferir en lo posible el acceso a este.
- De protección: controla el acceso al objeto original, por ejemplo: comprueba que el cliente tiene los permisos necesarios para realizar una petición. Por lo tanto, permiten realizar diversas tareas de mantenimiento adicionales al acceder a un objeto.
- De referencia inteligente: sustituto de un puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto. También permiten realizar diversas tareas de mantenimiento adicionales al acceder a un objeto.

Se recomienda el uso del patrón Proxy cuando:

- El sistema requiere una representación remota para un objeto en un diferente lugar.
- Se desea ocultar la complejidad de un objeto representándolo con una simple que no requiera mayor conocimiento para de esta manera facilitar su uso.
- Los objetos deben tener diferentes puntos de acceso, controlando el acceso al objeto original, sin que esto signifique instanciar el objeto en diferentes lugares.

Relaciones con otros patrones:

- Adapter, ya que, ambos redirigen la petición del cliente al verdadero sujeto que la ejecuta con la posibilidad de incorporar lógica adicional.
- Similar a Fecade en el sentido de que ambos pueden almacenar temporalmente una entidad compleja e inicializarla por su cuenta. Al contrario que Facade, Proxy tiene la misma interfaz que su objeto de servicio, lo que hace que sean intercambiables.
- Decorate. Aquí hay una diferencia dado que, el decorate añade responsabilidades a un objeto, el proxy solo controla su acceso

Ventajas

- ✓ Puedes controlar el objetivo de servicio sin que los clientes lo sepan
- ✓ Puedes gestionar el ciclo de vida del objeto de servicio cuando a los clientes no les importa
- ✓ El proxy funciona incluso si el objeto de servicio no está listo o no está disponible.
- ✓ Principio de abierto/cerrado. Puedes introducir nuevos proxies sin cambiar el servicio o los clientes.

Desventajas

- × El código puede complicarse ya que debes introducir gran cantidad de clases nuevas
- × La respuesta del servicio puede retrasarse

Estructura:

```
using System;
namespace Proxy.PatronDeDiseño
{
    public class Program
    {
        public static void Main(string[] args)

            Proxy proxy = new Proxy();
            proxy.Request();
            Console.ReadKey();
        }
    }

    public abstract class Subject
    {
        public abstract void Request();
    }


    public class RealSubject : Subject
    {
        public override void Request()
        {
            Console.WriteLine("Called RealSubject.Request()");
        }
    }

    public class Proxy : Subject
    {
        private RealSubject realSubject;

        public override void Request()
        {
            // Use 'lazy initialization'

            if (realSubject == null)
            {
                realSubject = new RealSubject();
            }

            realSubject.Request();
        }
    }
}
```

Ejemplo:

```
using System;

namespace Proxy.RealWorld
{

    0 referencias
    public class Program
    {
        0 referencias
        public static void Main(string[] args)
        {
            // Crea Proxy

            OperacionMatematicaProxy proxy = new OperacionMatematicaProxy();

            // Hace la operacion

            Console.WriteLine("7 + 2 = " + proxy.Add(7, 2));
            Console.WriteLine("7 - 2 = " + proxy.Sub(7, 2));
            Console.WriteLine("7 * 2 = " + proxy.Mul(7, 2));
            Console.WriteLine("7 / 2 = " + proxy.Div(7, 2));

            Console.ReadKey();
        }
    }


    2 referencias
    public interface IOperacion
    {
        4 referencias
        double Add(double x, double y);
        4 referencias
        double Sub(double x, double y);
        4 referencias
        double Mul(double x, double y);
        4 referencias
        double Div(double x, double y);
    }
```

```
    2 referencias
    public class OperacionMatematicaReal : IOperacion
    {
        2 referencias
        public double Add(double x, double y) { return x + y; }
        2 referencias
        public double Sub(double x, double y) { return x - y; }
        2 referencias
        public double Mul(double x, double y) { return x * y; }
        2 referencias
        public double Div(double x, double y) { return x / y; }
    }

    2 referencias
    public class OperacionMatematicaProxy : IOperacion
    {
        private OperacionMatematicaReal operacionMatematicaReal = new OperacionMatematicaReal();

        2 referencias
        public double Add(double x, double y)
        {
            return operacionMatematicaReal.Add(x, y);
        }
        2 referencias
        public double Sub(double x, double y)
        {
            return operacionMatematicaReal.Sub(x, y);
        }
        2 referencias
        public double Mul(double x, double y)
        {
            return operacionMatematicaReal.Mul(x, y);
        }
        2 referencias
        public double Div(double x, double y)
        {
            return operacionMatematicaReal.Div(x, y);
        }
    }
}
```