IMPLEMENTACION:
https://github.com/heros789-sergio/design-patterns-cc2

DESIGN PATTERNS

INTEGRANTES:

- Sergio Daniel Mogollon Caceres
- Luciana Julissa Huaman Coaquira
- Paul Antony Parizaca Mozo
- Nelzon Jorge Apaza Apaza

Patrones de Diseño

ESTRUCTURALES:

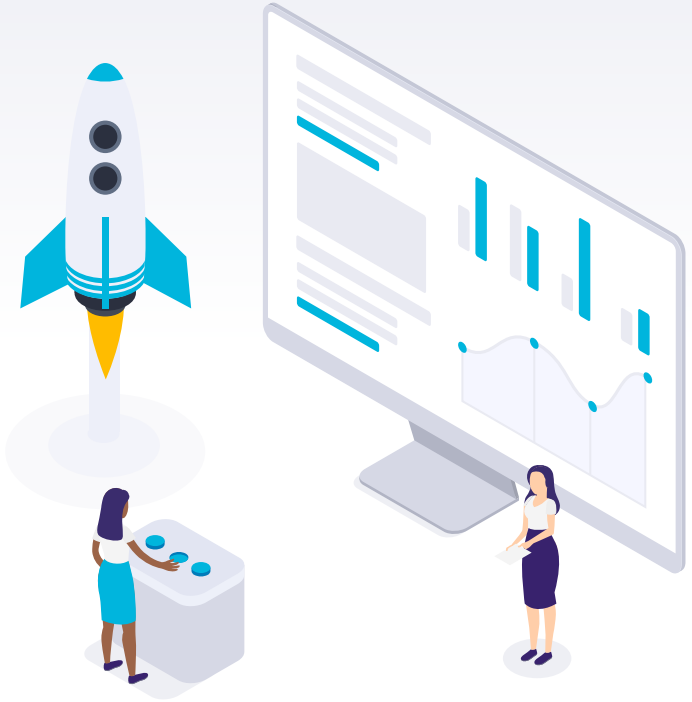
- **Facade**
- **Flyweight**
- **Proxy**

COMPORTAMIENTO

- **Visitor**

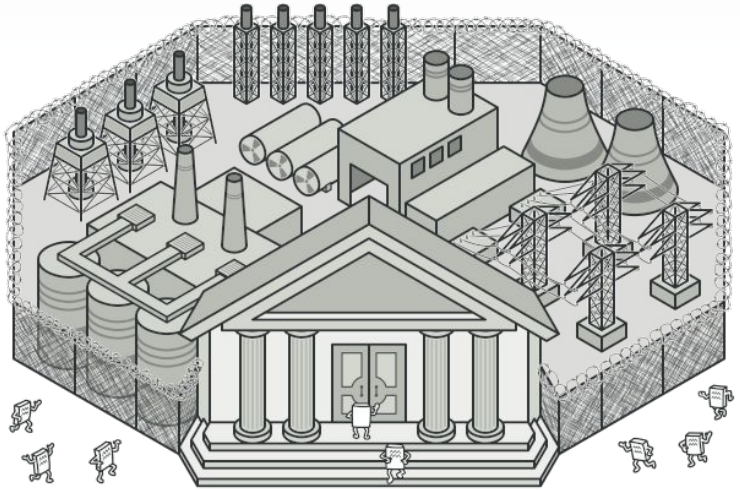
1

Patrones Estructurales

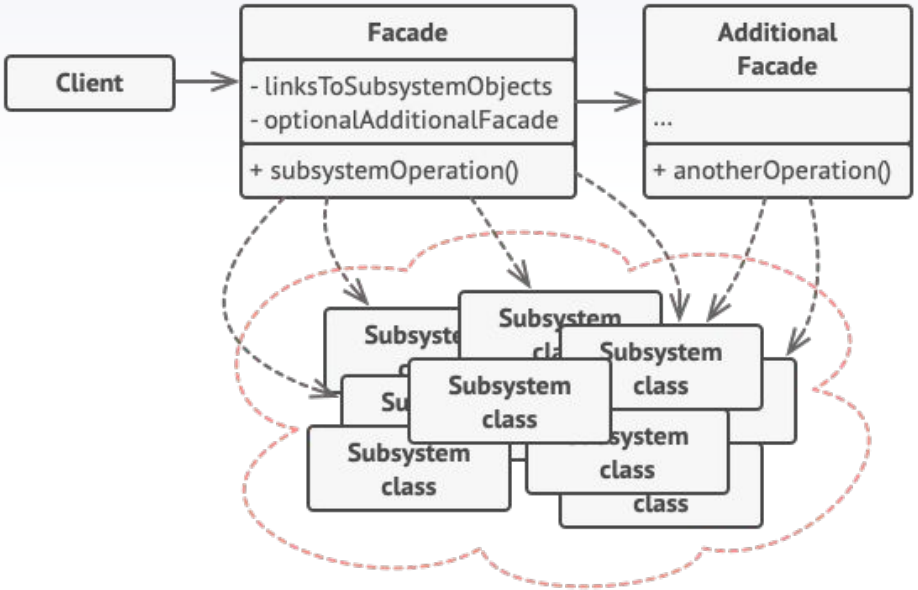


Facade

Es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases. Básicamente significa simplificar el código en una nueva clase



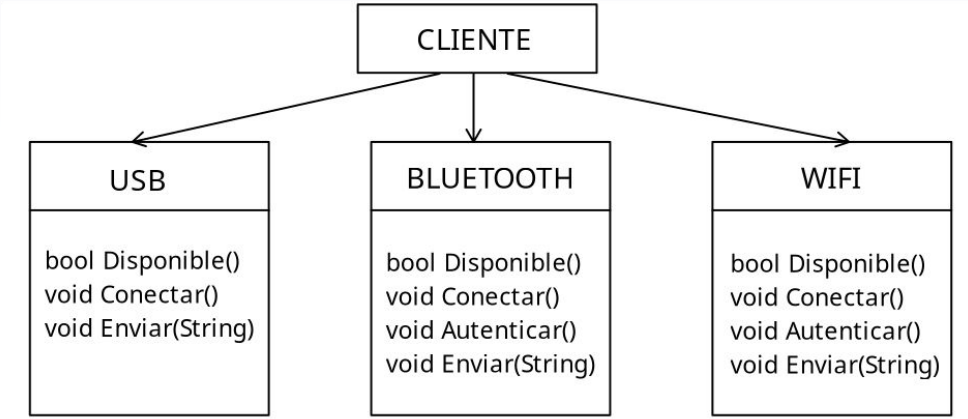
Facade: Estructura



1. El patrón Facade proporciona un práctico acceso a una parte específica de la funcionalidad del subsistema. Sabe a dónde dirigir la petición del cliente y cómo operar todas las partes móviles.
2. Puede crearse una clase Fachada Adicional para evitar contaminar una única fachada con funciones no relacionadas que podrían convertirla en otra estructura compleja. Las fachadas adicionales pueden utilizarse por clientes y por otras fachadas.
3. El Subsistema Complejo consiste en decenas de objetos diversos. Para lograr que todos hagan algo significativo, debes profundizar en los detalles de implementación del subsistema, que pueden incluir inicializar objetos en el orden correcto y suministrarles datos en el formato adecuado. Las clases del subsistema no conocen la existencia de la fachada. Operan dentro del sistema y trabajan entre sí directamente.
4. El Cliente utiliza la fachada en lugar de invocar directamente los objetos del subsistema.

Facade

Ejemplo del funcionamiento sin la clase Facade de envío de archivos.

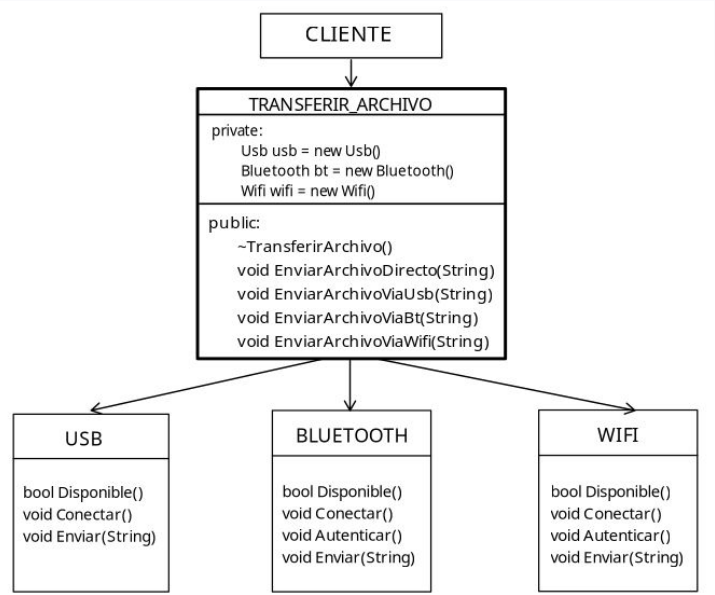


VIA WIFI

1. Si el cliente quiere enviar un archivo ya sea por cualquiera de las 3 vías.
2. Tiene que crear un objeto y separar memoria.
3. Luego ir llamando a las funciones del objeto.
4. Primero ver la disponibilidad de la vía.
5. Si la disponibilidad es true, se podrá conectar
6. Luego se Autentica.
7. Y por último se envía el archivo.

Facade

Ejemplo del funcionamiento con la clase Facade de envío de archivos. (La clase TransferirArchivo seria la clase fachada)



VIA WIFI

1. Se tiene que crear un objeto de tipo TransferirArchivo que seria nuestra clase fachada.
2. Luego en el caso de wifi podemos llamar al método EnviarArchivoViaWifi(String).
3. Ese método se encargará de enviar el archivo.

Facade

APLICABILIDAD

-A menudo los subsistemas se vuelven más complejos con el tiempo. Incluso la aplicación de patrones de diseño suele conducir a la creación de un mayor número de clases. Un subsistema puede hacerse más flexible y más fácil de reutilizar en varios contextos, pero la cantidad de código de configuración que exige de un cliente, crece aún más. El patrón Facade intenta solucionar este problema proporcionando un atajo a las funciones más utilizadas del subsistema que mejor encajan con los requisitos del cliente.

-Crea fachadas para definir puntos de entrada a cada nivel de un subsistema. Puedes reducir el acoplamiento entre varios subsistemas exigiendo que se comuniquen únicamente mediante fachadas.

PROS:

Puedes aislar tu código de la complejidad de un subsistema.

Para aprovechar el patrón al máximo, haz que todo el código cliente se comunique con el subsistema únicamente a través de la fachada.

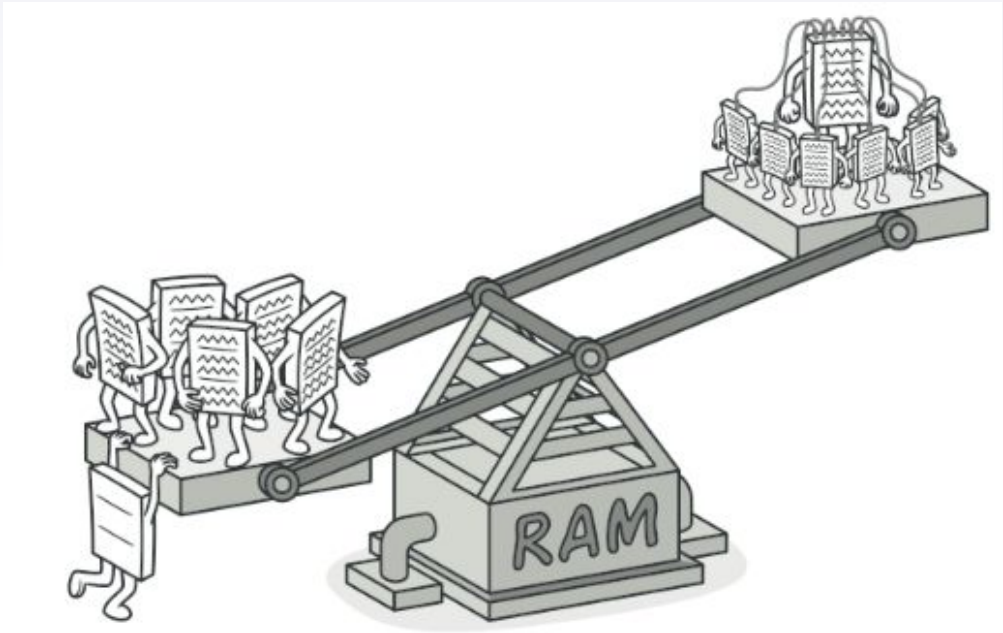
CONTRAS:

Una fachada puede convertirse en un objeto todopoderoso acoplado a todas las clases de una aplicación.

Por eso si la fachada se vuelve demasiado grande, piensa en extraer parte de su comportamiento y colocarlo dentro de una nueva clase fachada refinada.

Flyweight

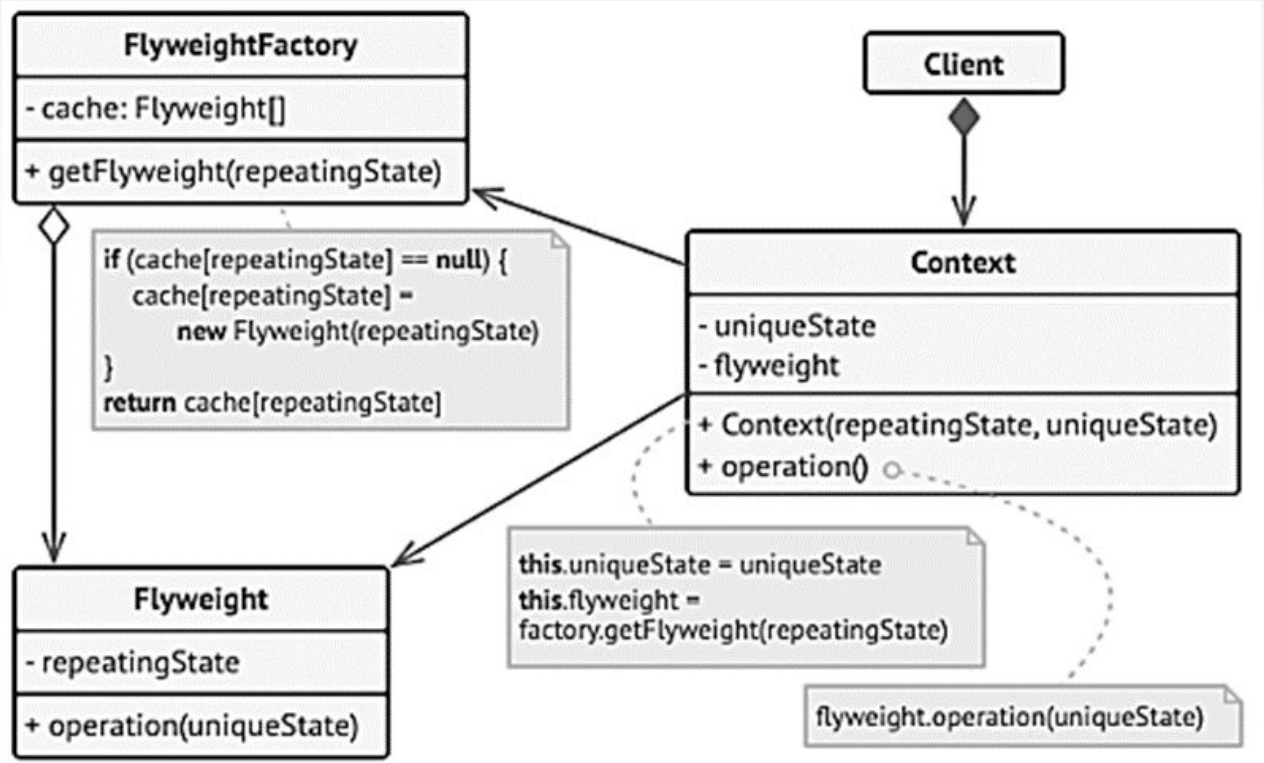
Flyweight es un patrón de diseño **estructural** que te permite mantener más objetos dentro de la cantidad disponible de **RAM** compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.



Flyweight: Estructura

Intervienen:

- Class **UniqueState**
- Class **SharedState**
- Class **FlyWeight** (union de los 2 anteriores)
- Class **FlyWeightFactory** **(**gestiona los flyweights, puede reutilizar los ya existentes o crear nuevos**)**
- Funcion **Cliente**()



Flyweight: Ejemplo 1

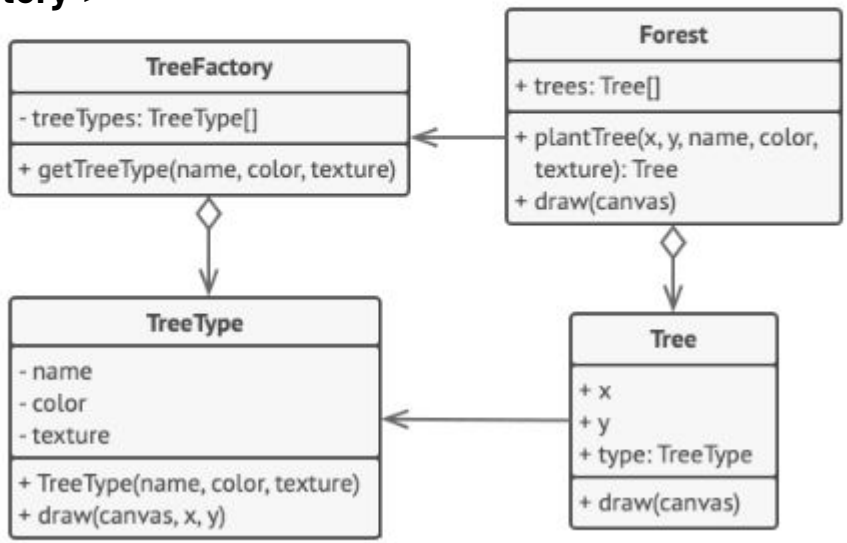
- El patrón extrae el estado intrínseco repetido de una clase principal **Tree** y la mueve dentro de la clase flyweight **TreeType**.
- Ahora, en lugar de almacenar la misma información en varios objetos, se mantiene en unos pocos objetos flyweight vinculados a los objetos de **Tree** adecuados que actúan como contexto. El código cliente crea nuevos objetos árbol utilizando la **FlyWeight Factory** que encapsula la complejidad de buscar el objeto adecuado y reutilizarlo si es necesario.

Pseudocódigo

En este ejemplo, el patrón **Flyweight** ayuda a reducir el uso de memoria a la hora de representar millones de objetos de árbol en un lienzo.

Cliente->

FlyWeight Factory->



Shared State ->

Unique State ->

Flyweight: ¿Cómo Implementarlo?

1. Divide los campos de una clase que se convertirá en flyweight en dos partes:
   - **Estado intrínseco = Shared State (**información invariable **duplicada** en varios objetos**)**
   - **Estado extrínseco = UniqueState (**información contextual **única** de cada objeto**)**
2. Asegurar que los campos del estado intrínseco (**UniqueState**) sean inmutables. Deben llevar sus valores iniciales únicamente dentro del constructor.
3. Crea una **clase fábrica** para gestionar el grupo de objetos flyweight, buscando uno existente antes de crear uno nuevo.
4. Los clientes sólo deberán solicitar objetos flyweight a través de la clase fábrica. Deberán describir el flyweight deseado pasando su estado intrínseco a la fábrica.
5. El cliente deberá almacenar o calcular valores del estado extrínseco (contexto) para poder invocar métodos de objetos flyweight. Por comodidad, el estado extrínseco puede moverse a una clase contexto separada junto con el campo referenciador del flyweight.

Flyweight: Formas de Aplicación

La ventaja de aplicar el patrón depende en gran medida de cómo y dónde se utiliza. Resulta más útil cuando:

- La aplicación necesita generar una cantidad enorme de objetos similares
- Esto consume toda la RAM disponible de un dispositivo objetivo
- **Los objetos contienen estados duplicados que se pueden extraer y compartir entre varios objetos**
- En comparación con otros patrones, **Flyweight** muestra cómo crear muchos pequeños objetos, mientras que **Facade** muestra cómo crear un único objeto que represente un subsistema completo.
- Los objetos flyweight son **inmutables (**no se pueden alterar**)**.

Flyweight: Pros y Contras

Pros y contras

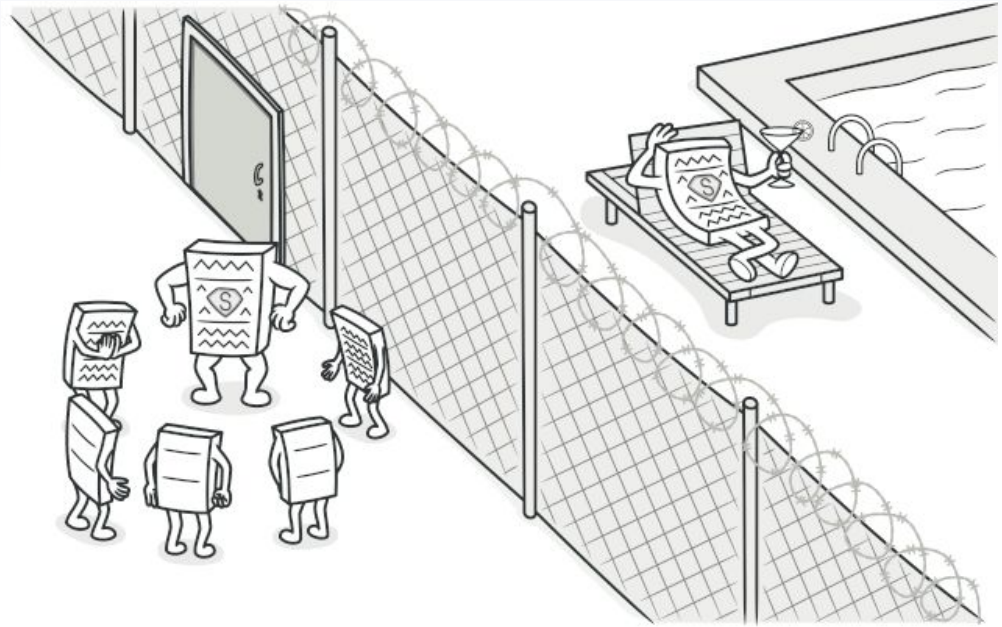
✓ Puedes ahorrar mucha RAM, siempre que tu programa tenga toneladas de objetos similares.

✗ Puede que estés cambiando RAM por ciclos CPU cuando deba calcularse de nuevo parte de la información de contexto cada vez que alguien invoque un método flyweight.

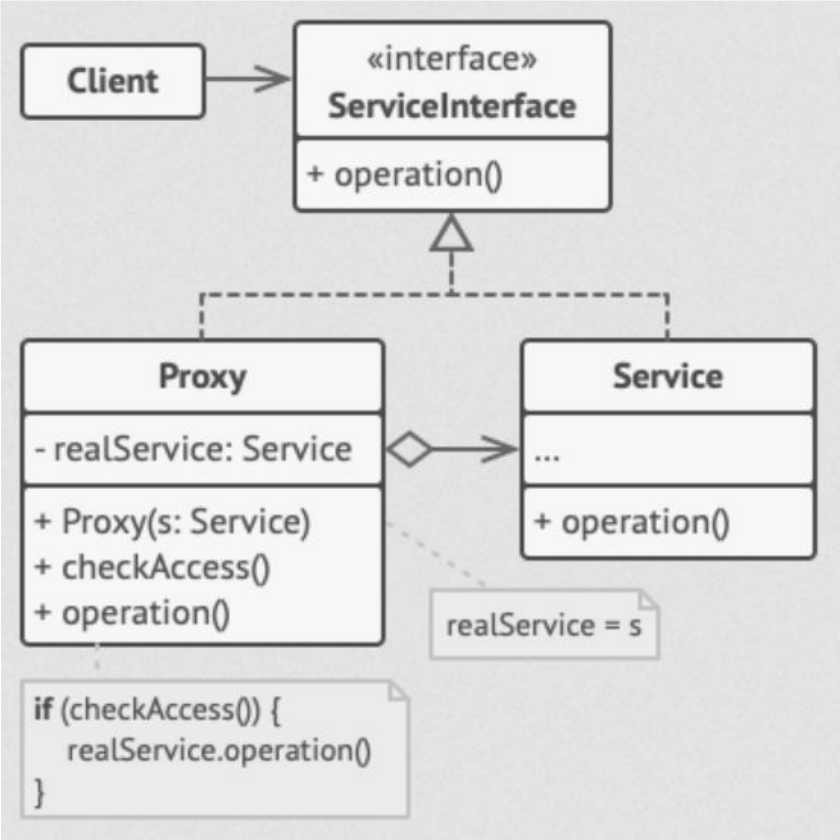
✗ El código se complica mucho. Los nuevos miembros del equipo siempre estarán preguntándose por qué el estado de una entidad se separó de tal manera.

Proxy

Proxy es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.



Proxy: Estructura



Proxy: Pros y Contras

Pros:

- Puedes controlar el objeto 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.

Contras:

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

Proxy: Implementación

El ejemplo del código, con la implementación se encuentra en el repositorio de GitHub (link dejado en la última diapositiva)

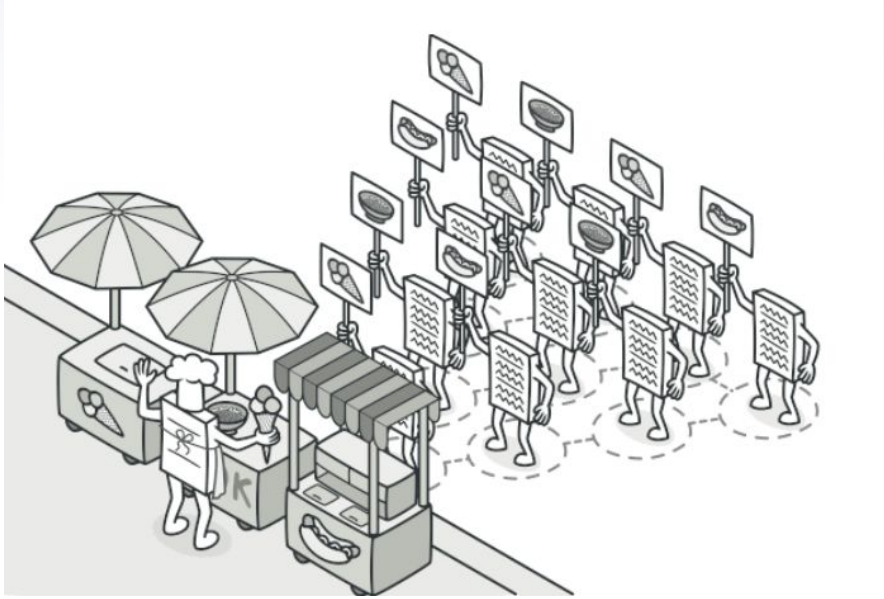
Otras formas para utilizar proxy:

- Inicialización diferida (proxy virtual). Es cuando tienes un objeto de servicio muy pesado que utiliza muchos recursos del sistema al estar siempre funcionando, aunque solo lo necesites de vez en cuando.
- Control de acceso (proxy de protección). Es cuando quieres que únicamente clientes específicos sean capaces de utilizar el objeto de servicio.
- Solicitudes de registro (proxy de registro). Es cuando quieres mantener un historial de solicitudes al objeto de servicio.
- Resultados de solicitudes en caché (proxy de caché). Es cuando necesitas guardar en caché resultados de solicitudes de clientes y gestionar el ciclo de vida de ese caché, especialmente si los resultados son muchos.

2

Patrones de Comportamiento

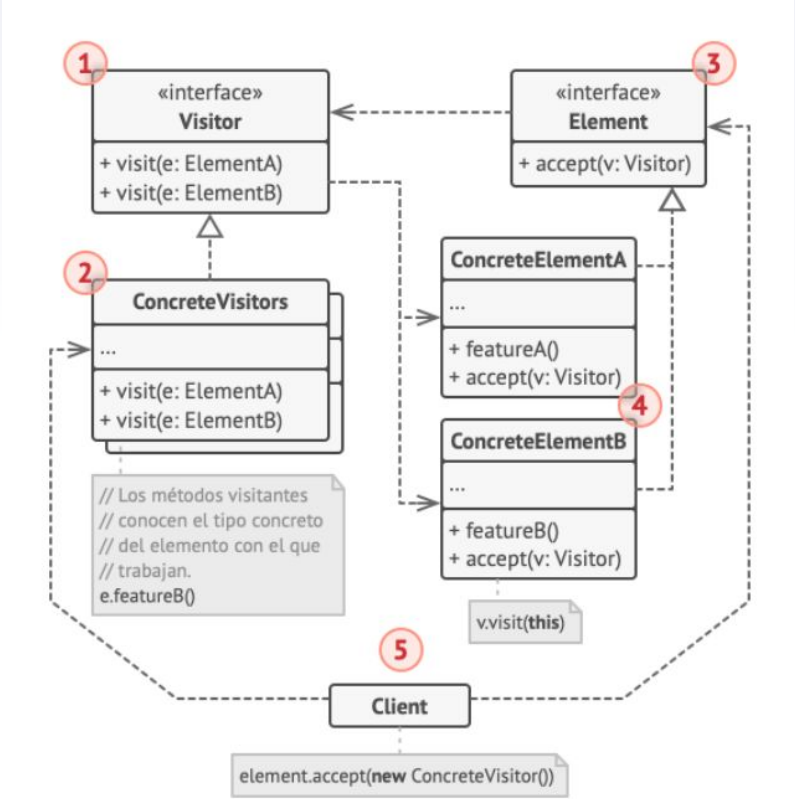
Visitor



Visitor es un patrón de diseño de comportamiento que permite añadir nuevos comportamientos a una jerarquía de clases existente sin alterar el código.

Es decir permite separar algoritmos de los objetos sobre los que operan

Visitor: Estructura



Visitor: Implementación

¿Cómo implementarlo?

- Declara la interfaz visitante con un grupo de métodos “visitantes”, uno por cada clase de elemento concreto existente en el programa.
- Declara la interfaz de elemento. Si estás trabajando con una jerarquía de clases de elementos existente, añade el método abstracto de “aceptación” a la clase base de la jerarquía. Este método debe aceptar un objeto visitante como argumento.
- Implementa los métodos de aceptación en todas las clases de elementos concretos. Estos métodos simplemente deben redirigir la llamada a un método visitante en el objeto visitante entrante que coincida con la clase del elemento actual.
- Las clases de elemento sólo deben funcionar con visitantes a través de la interfaz visitante. Los visitantes, sin embargo, deben conocer todas las clases de elemento concreto, referenciadas como tipos de parámetro de los métodos de visita.
- Por cada comportamiento que no pueda implementarse dentro de la jerarquía de elementos, crea una nueva clase concreta visitante e implementa todos los métodos visitantes.
- El cliente debe crear objetos visitantes y pasarlos dentro de elementos a través de métodos de “aceptación”.

El ejemplo del código, con la implementación se encuentra en el repositorio de GitHub (link dejado en la última diapositiva)

Visitor: Pros y Contras

Pros:

- Se puede ingresar un nuevo comportamiento que puede funcionar con objetos de clases diferentes sin cambiar esas clases.
- Principio de responsabilidad única.Puedes tomar varias versiones del mismo comportamiento y ponerlas en la misma clase.
- Un objeto visitante puede acumular cierta información útil mientras trabaja con varios objetos.

Contras:

- Debes actualizar todos los visitantes cada vez que una clase se añada o elimine de la jerarquía de elementos.
- Los visitantes pueden carecer del acceso necesario a los campos y métodos privados de los elementos