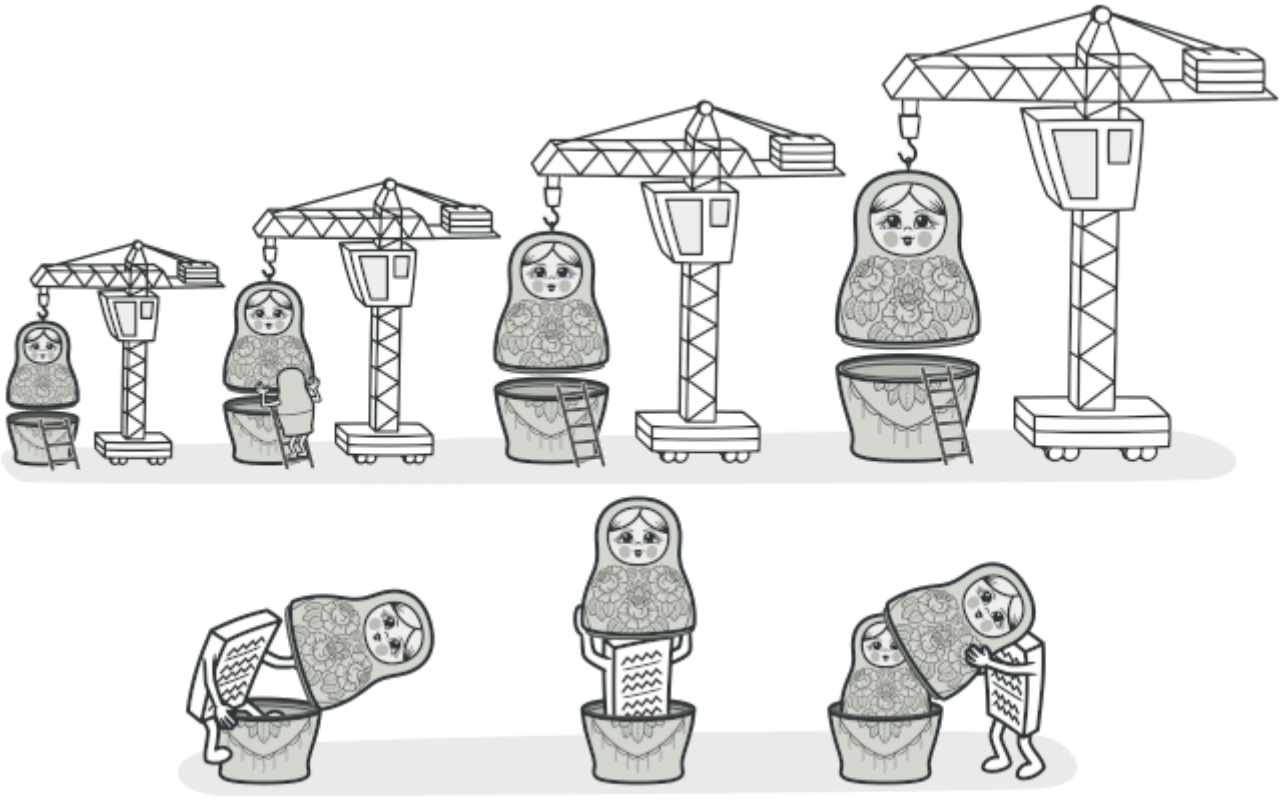
Decorator

También llamado: Decorador, Envoltorio, Wrapper

Propósito

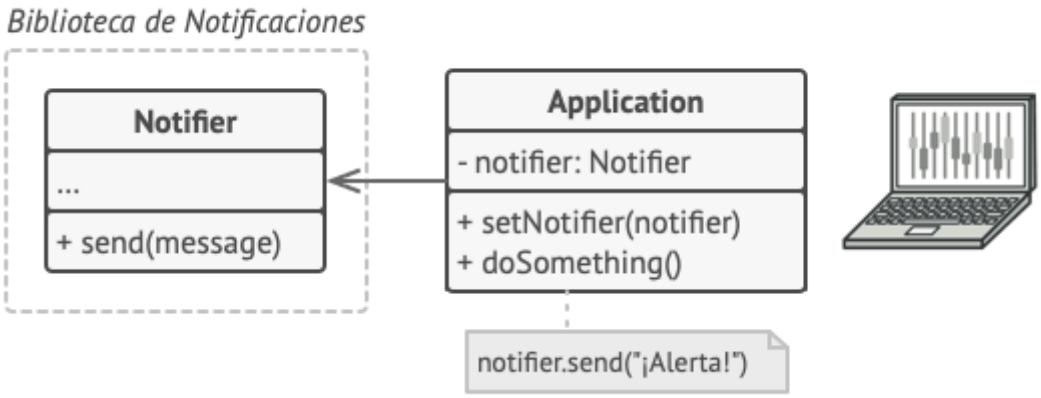
Decorator es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.



Problema

Imagina que estás trabajando en una biblioteca de notificaciones que permite a otros programas notificar a sus usuarios acerca de eventos importantes.

La versión inicial de la biblioteca se basaba en la clase `Notificador` que solo contaba con unos cuantos campos, un constructor y un único método `send`. El método podía aceptar un argumento de mensaje de un cliente y enviar el mensaje a una lista de correos electrónicos que se pasaban a la clase notificadora a través de su constructor. Una aplicación de un tercero que actuaba como cliente debía crear y configurar el objeto notificador una vez y después utilizarlo cada vez que sucediera algo importante.



Un programa puede utilizar la clase notificadora para enviar notificaciones sobre eventos importantes a un grupo predefinido de correos electrónicos.

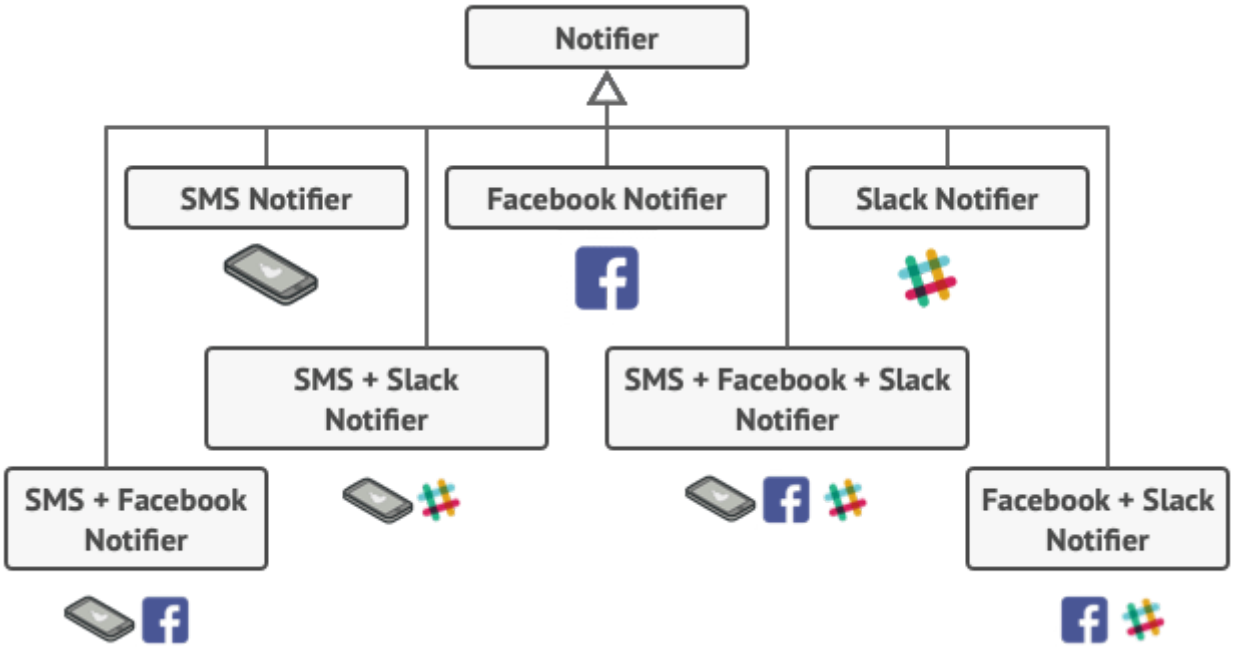
En cierto momento te das cuenta de que los usuarios de la biblioteca esperan algo más que unas simples notificaciones por correo. A muchos de ellos les gustaría recibir mensajes SMS sobre asuntos importantes. Otros querrían recibir las notificaciones por Facebook y, por supuesto, a los usuarios corporativos les encantaría recibir notificaciones por Slack.

Cada tipo de notificación se implementa como una subclase de la clase notificadora.

No puede ser muy complicado ¿verdad? Extendiste la clase `Notificador` y metiste los métodos adicionales de notificación dentro de nuevas subclases. Ahora el cliente debería instanciar la clase notificadora deseada y utilizarla para el resto de notificaciones.

Pero entonces alguien te hace una pregunta razonable: “¿Por qué no se pueden utilizar varios tipos de notificación al mismo tiempo? Si tu casa está en llamas, probablemente quieras que te informen a través de todos los canales”.

Intentaste solucionar ese problema creando subclases especiales que combinaban varios métodos de notificación dentro de una clase. Sin embargo, enseguida resultó evidente que esta solución inflaría el código en gran medida, no sólo el de la biblioteca, sino también el código cliente.



Explosión combinatoria de subclases.

Debes encontrar alguna otra forma de estructurar las clases de las notificaciones para no alcanzar cifras que rompan accidentalmente un récord Guinness.

😊 Solución

Cuando tenemos que alterar la funcionalidad de un objeto, lo primero que se viene a la mente es extender una clase. No obstante, la herencia tiene varias limitaciones importantes de las que debes ser consciente.

- La herencia es estática. No se puede alterar la funcionalidad de un objeto existente durante el tiempo de ejecución. Sólo se puede sustituir el objeto completo por otro creado a partir de una subclase diferente.
- Las subclases sólo pueden tener una clase padre. En la mayoría de lenguajes, la herencia no permite a una clase heredar comportamientos de varias clases al mismo tiempo.

Una de las formas de superar estas limitaciones es empleando la *Agregación* o la *Composición* ⓘ en lugar de la *Herencia*. Ambas alternativas funcionan prácticamente del mismo modo: un objeto *tiene una* referencia a otro y le delega parte del trabajo, mientras que con la herencia, el propio objeto *puede* realizar ese trabajo, heredando el comportamiento de su superclase.

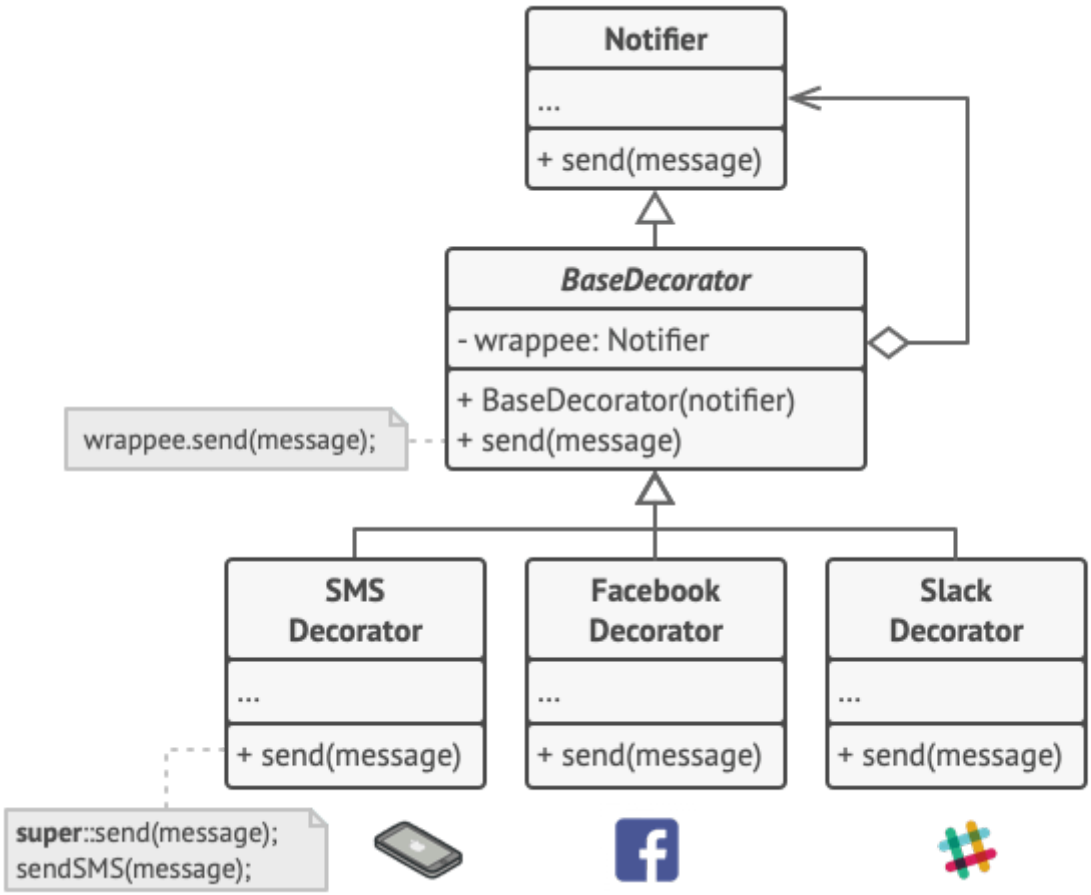
Con esta nueva solución puedes sustituir fácilmente el objeto "ayudante" vinculado por otro, cambiando el comportamiento del contenedor durante el tiempo de ejecución. Un objeto puede utilizar el comportamiento de varias clases con referencias a varios objetos, delegándoles todo tipo de tareas. La agregación/composición es el principio clave que se esconde tras muchos patrones de diseño, incluyendo el Decorator. A propósito, regresemos a la discusión sobre el patrón.

Herencia vs. Agregación

"Wrapper" (envoltorio, en inglés) es el sobrenombre alternativo del patrón Decorator, que expresa claramente su idea principal. Un *wrapper* es un objeto que puede vincularse con un objeto *objetivo*. El wrapper contiene el mismo grupo de métodos que el objetivo y le delega todas las solicitudes que recibe. No obstante, el wrapper puede alterar el resultado haciendo algo antes o después de pasar la solicitud al objetivo.

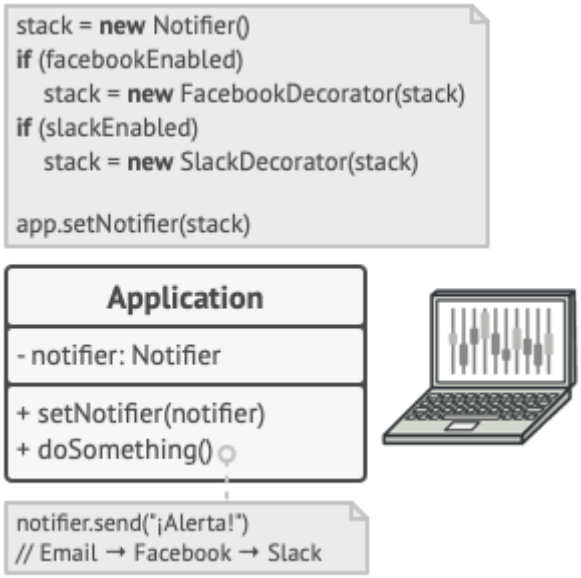
¿Cuándo se convierte un simple wrapper en el verdadero decorador? Como he mencionado, el wrapper implementa la misma interfaz que el objeto envuelto. Éste es el motivo por el que, desde la perspectiva del cliente, estos objetos son idénticos. Haz que el campo de referencia del wrapper acepte cualquier objeto que siga esa interfaz. Esto te permitirá *envolver* un objeto en varios wrappers, añadiéndole el comportamiento combinado de todos ellos.

En nuestro ejemplo de las notificaciones, dejemos la sencilla funcionalidad de las notificaciones por correo electrónico dentro de la clase base `Notificador`, pero convirtamos el resto de los métodos de notificación en decoradores.



Varios métodos de notificación se convierten en decoradores.

El código cliente debe envolver un objeto notificador básico dentro de un grupo de decoradores que satisfagan las preferencias del cliente. Los objetos resultantes se estructurarán como una pila.

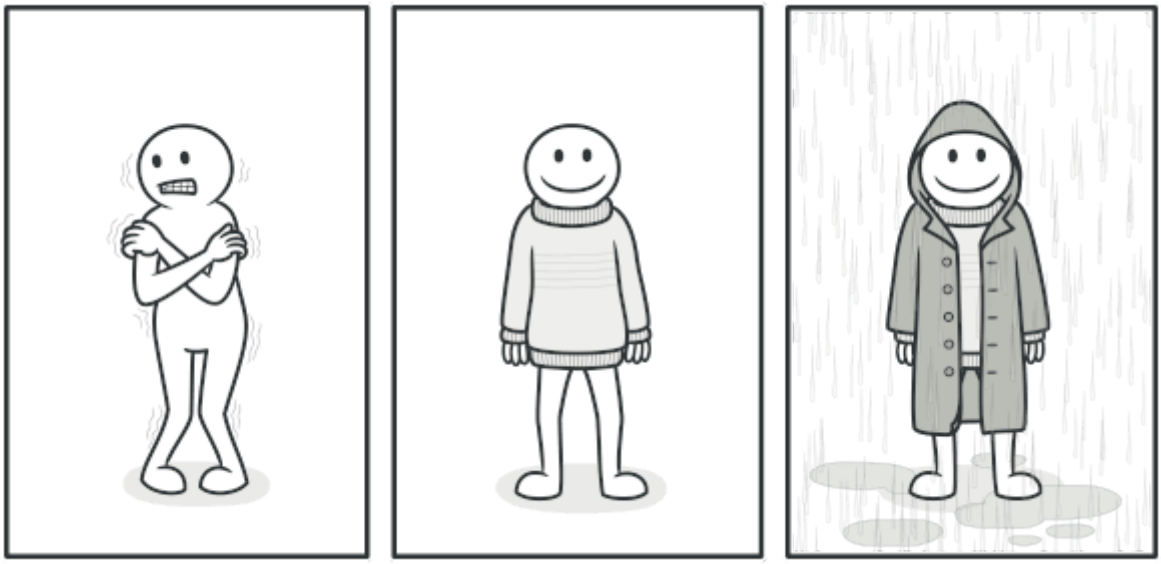


Las aplicaciones pueden configurar pilas complejas de decoradores de notificación.

El último decorador de la pila será el objeto con el que el cliente trabaja. Debido a que todos los decoradores implementan la misma interfaz que la notificadora base, al resto del código cliente no le importa si está trabajando con el objeto notificador “puro” o con el decorado.

Podemos aplicar la misma solución a otras funcionalidades, como el formateo de mensajes o la composición de una lista de destinatarios. El cliente puede decorar el objeto con los decoradores personalizados que desee, siempre y cuando sigan la misma interfaz que los demás.

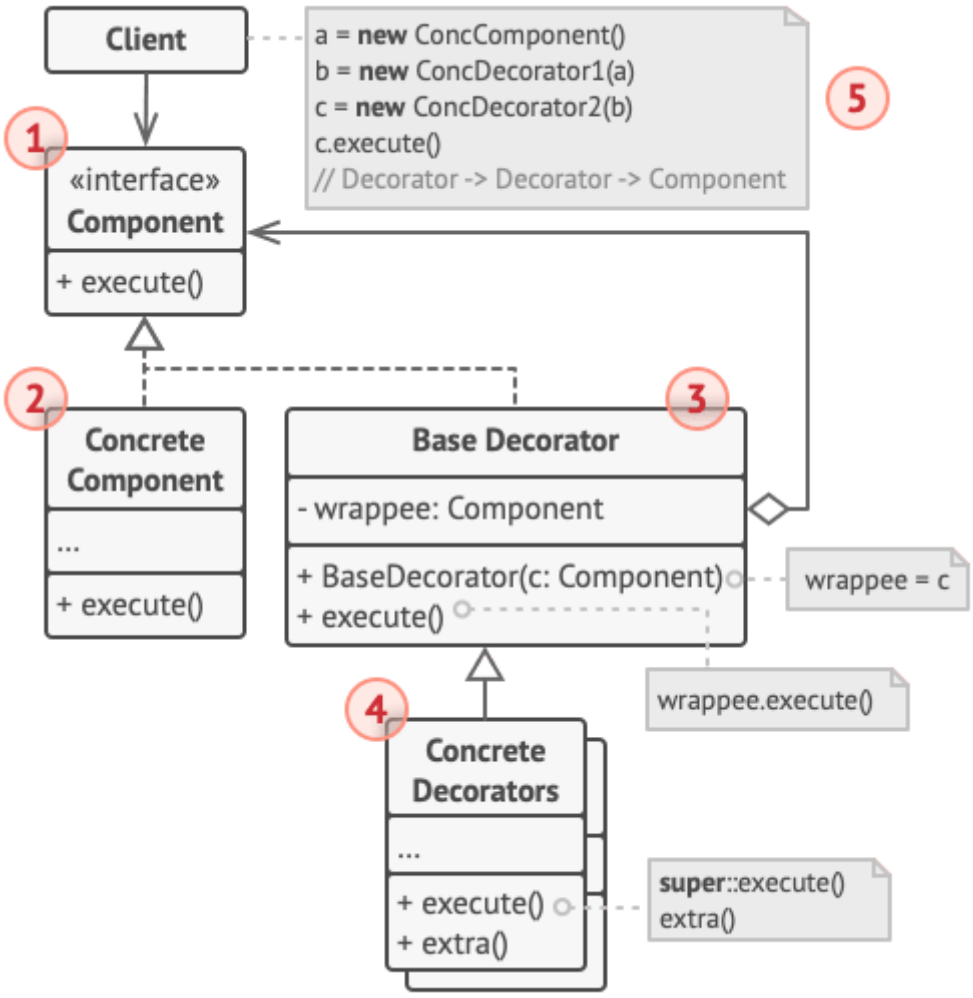
Analogía en el mundo real



Obtienes un efecto combinado vistiendo varias prendas de ropa.

Vestir ropa es un ejemplo del uso de decoradores. Cuando tienes frío, te cubres con un suéter. Si sigues teniendo frío a pesar del suéter, puedes ponerte una chaqueta encima. Si está lloviendo, puedes ponerte un impermeable. Todas estas prendas “extienden” tu comportamiento básico pero no son parte de ti, y puedes quitarte fácilmente cualquier prenda cuando lo desees.

Estructura



1. El **Componente** declara la interfaz común tanto para wrappers como para objetos envueltos.

2. **Componente Concreto** es una clase de objetos envueltos. Define el comportamiento básico, que los decoradores pueden alterar.

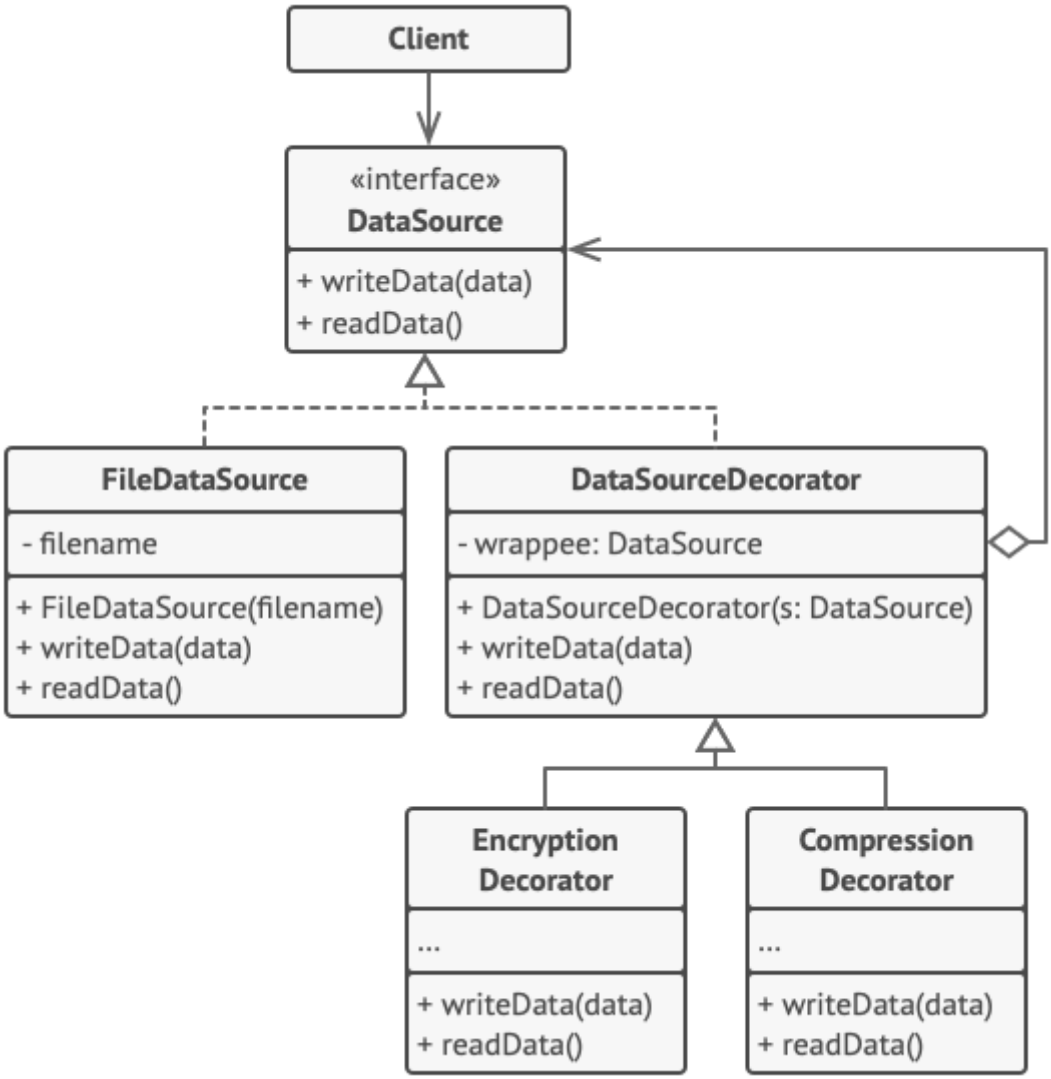
3. La clase **Decoradora Base** tiene un campo para referenciar un objeto envuelto. El tipo del campo debe declararse como la interfaz del componente para que pueda contener tanto los componentes concretos como los decoradores. La clase decoradora base delega todas las operaciones al objeto envuelto.

4. Los **Decoradores Concretos** definen funcionalidades adicionales que se pueden añadir dinámicamente a los componentes. Los decoradores concretos sobrescriben métodos de la clase decoradora base y ejecutan su comportamiento, ya sea antes o después de invocar al método padre.

5. El **Cliente** puede envolver componentes en varias capas de decoradores, siempre y cuando trabajen con todos los objetos a través de la interfaz del componente.

Pseudocódigo

En este ejemplo, el patrón **Decorator** te permite comprimir y encriptar información delicada independientemente del código que utiliza esos datos.



Ejemplo de la encriptación y compresión de decoradores.

La aplicación envuelve el objeto de la fuente de datos con un par de decoradores. Ambos wrappers cambian el modo en que los datos se escriben y se leen en el disco:

- Justo antes de que los datos se **escriban en el disco**, los decoradores los encriptan y comprimen. La clase original escribe en el archivo los datos encriptados y protegidos, sin conocer el cambio.

- Después de que los datos son **leídos del disco**, pasan por los mismos decoradores, que los descomprimen y decodifican.

Los decoradores y la clase fuente de datos implementan la misma interfaz, lo que los hace intercambiables en el código cliente.

```
// La interfaz de componente define operaciones que los
// decoradores pueden alterar.
interface DataSource is
    method writeData(data)
    method readData():data

// Los componentes concretos proporcionan implementaciones por
// defecto para las operaciones. En un programa puede haber
// muchas variaciones de estas clases.
class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }

    method writeData(data) is
        // Escribe datos en el archivo.

    method readData():data is
        // Lee datos del archivo.

// La clase decoradora base sigue la misma interfaz que los
// demás componentes. El principal propósito de esta clase es
// definir la interfaz de encapsulación para todos los
// decoradores concretos. La implementación por defecto del
// código de encapsulación puede incluir un campo para almacenar
// un componente envuelto y los medios para inicializarlo.
class DataSourceDecorator implements DataSource is
    protected field wrappee: DataSource

    constructor DataSourceDecorator(source: DataSource) is
        wrappee = source

    // La decoradora base simplemente delega todo el trabajo al
    // componente envuelto. En los decoradores concretos se
    // pueden añadir comportamientos adicionales.
    method writeData(data) is
        wrappee.writeData(data)

    // Los decoradores concretos pueden invocar la
    // implementación padre de la operación en lugar de invocar
    // directamente al objeto envuelto. Esta solución simplifica
    // la extensión de las clases decoradoras.
    method readData():data is
        return wrappee.readData()

// Los decoradores concretos deben invocar métodos en el objeto
// envuelto, pero pueden añadir algo de su parte al resultado.
// Los decoradores pueden ejecutar el comportamiento añadido
// antes o después de la llamada a un objeto envuelto.
class EncryptionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Encripta los datos pasados.
        // 2. Pasa los datos encriptados al método writeData
```

```
            // (escribirDatos) del objeto envuelto.

        method readData():data is
            // 1. Obtiene datos del método readData (leerDatos) del
            // objeto envuelto.
            // 2. Intenta descifrarlo si está encriptado.
            // 3. Devuelve el resultado.

    // Puedes envolver objetos en varias capas de decoradores.
    class CompressionDecorator extends DataSourceDecorator is
        method writeData(data) is
            // 1. Comprime los datos pasados.
            // 2. Pasa los datos comprimidos al método writeData del
            // objeto envuelto.

        method readData():data is
            // 1. Obtiene datos del método readData del objeto
            // envuelto.
            // 2. Intenta descomprimirlo si está comprimido.
            // 3. Devuelve el resultado.


    // Opción 1. Un ejemplo sencillo del montaje de un decorador.
    class Application is
        method dumbUsageExample() is
            source = new FileDataSource("somefile.dat")
            source.writeData(salaryRecords)
            // El archivo objetivo se ha escrito con datos sin
            // formato.

            source = new CompressionDecorator(source)
            source.writeData(salaryRecords)
            // El archivo objetivo se ha escrito con datos
            // comprimidos.

            source = new EncryptionDecorator(source)
            // La variable fuente ahora contiene esto:
            // Cifrado > Compresión > FileDataSource
            source.writeData(salaryRecords)
            // El archivo se ha escrito con datos comprimidos y
            // encriptados.


    // Opción 2. El código cliente que utiliza una fuente externa de
    // datos. Los objetos SalaryManager no conocen ni se preocupan
    // por las especificaciones del almacenamiento de datos.
    // Trabajan con una fuente de datos preconfigurada recibida del
    // configurador de la aplicación.
    class SalaryManager is
        field source: DataSource

        constructor SalaryManager(source: DataSource) { ... }
```

```
    method load() is
        return source.readData()

    method save() is
        source.writeData(salaryRecords)
    // ...Otros métodos útiles...


// La aplicación puede montar distintas pilas de decoradores
// durante el tiempo de ejecución, dependiendo de la
// configuración o el entorno.
class ApplicationConfigurator is
    method configurationExample() is
        source = new FileDataSource("salary.dat")
        if (enabledEncryption)
            source = new EncryptionDecorator(source)
        if (enabledCompression)
            source = new CompressionDecorator(source)

        logger = new SalaryManager(source)
        salary = logger.load()
    // ...
```

Aplicabilidad

Utiliza el patrón Decorator cuando necesites asignar funcionalidades adicionales a objetos durante el tiempo de ejecución sin descomponer el código que utiliza esos objetos.

El patrón Decorator te permite estructurar tu lógica de negocio en capas, crear un decorador para cada capa y componer objetos con varias combinaciones de esta lógica, durante el tiempo de ejecución. El código cliente puede tratar a todos estos objetos de la misma forma, ya que todos siguen una interfaz común.

Utiliza el patrón cuando resulte extraño o no sea posible extender el comportamiento de un objeto utilizando la herencia.

Muchos lenguajes de programación cuentan con la palabra clave `final` que puede utilizarse para evitar que una clase siga extendiéndose. Para una clase final, la única forma de reutilizar el comportamiento existente será envolver la clase con tu propio wrapper, utilizando el patrón Decorator.

Cómo implementarlo

1. Asegúrate de que tu dominio de negocio puede representarse como un componente primario con varias capas opcionales encima.

2. Decide qué métodos son comunes al componente primario y las capas opcionales. Crea una interfaz de componente y declara esos métodos en ella.

3. Crea una clase concreta de componente y define en ella el comportamiento base.

4. Crea una clase base decoradora. Debe tener un campo para almacenar una referencia a un objeto envuelto. El campo debe declararse con el tipo de interfaz de componente para permitir la vinculación a componentes concretos, así como a decoradores. La clase decoradora base debe delegar todas las operaciones al objeto envuelto.

5. Asegúrate de que todas las clases implementan la interfaz de componente.

6. Crea decoradores concretos extendiéndolos a partir de la decoradora base. Un decorador concreto debe ejecutar su comportamiento antes o después de la llamada al método padre (que siempre delega al objeto envuelto).

7. El código cliente debe ser responsable de crear decoradores y componerlos del modo que el cliente necesite.

Pros y contras

- ✓ Puedes extender el comportamiento de un objeto sin crear una nueva subclase.
- ✓ Puedes añadir o eliminar responsabilidades de un objeto durante el tiempo de ejecución.
- ✓ Puedes combinar varios comportamientos envolviendo un objeto con varios decoradores.
- ✓ *Principio de responsabilidad única*. Puedes dividir una clase monolítica que implementa muchas variantes posibles de comportamiento, en varias clases más pequeñas.
- ✗ Resulta difícil eliminar un wrapper específico de la pila de wrappers.
- ✗ Es difícil implementar un decorador de tal forma que su comportamiento no dependa del orden en la pila de decoradores.
- ✗ El código de configuración inicial de las capas pueden tener un aspecto desagradable.

Relaciones con otros patrones

- **Adapter** cambia la interfaz de un objeto existente mientras que **Decorator** mejora un objeto sin cambiar su interfaz. Además, *Decorator* soporta la composición recursiva, lo cual no es posible al utilizar *Adapter*.

- **Adapter** proporciona una interfaz diferente al objeto envuelto, **Proxy** le proporciona la misma interfaz y **Decorator** le proporciona una interfaz mejorada.

- **Chain of Responsibility** y **Decorator** tienen estructuras de clase muy similares. Ambos patrones se basan en la composición recursiva para pasar la ejecución a través de una serie de objetos. Sin embargo, existen varias diferencias fundamentales:

  Los manejadores de *CoR* pueden ejecutar operaciones arbitrarias con independencia entre sí. También pueden dejar de pasar la solicitud en cualquier momento. Por otro lado, varios *decoradores* pueden extender el comportamiento del objeto manteniendo su consistencia con la interfaz base. Además, los decoradores no pueden romper el flujo de la solicitud.

- **Composite** y **Decorator** tienen diagramas de estructura similares ya que ambos se basan en la composición recursiva para organizar un número indefinido de objetos.

  Un *Decorator* es como un *Composite* pero sólo tiene un componente hijo. Hay otra diferencia importante: *Decorator* añade responsabilidades adicionales al objeto envuelto, mientras que *Composite* se limita a "recapitular" los resultados de sus hijos.

  No obstante, los patrones también pueden colaborar: puedes utilizar el *Decorator* para extender el comportamiento de un objeto específico del árbol *Composite*.

- Los diseños que hacen un uso amplio de **Composite** y **Decorator** a menudo pueden beneficiarse del uso del **Prototype**. Aplicar el patrón te permite clonar estructuras complejas en lugar de reconstruirlas desde cero.

- **Decorator** te permite cambiar la piel de un objeto, mientras que **Strategy** te permite cambiar sus entrañas.

- **Decorator** y **Proxy** tienen estructuras similares, pero propósitos muy distintos. Ambos patrones se basan en el principio de composición, por el que un objeto debe delegar parte del trabajo a otro. La diferencia es que, normalmente, un *Proxy* gestiona el ciclo de vida de su objeto de servicio por su cuenta, mientras que la composición de los *Decoradores* siempre está controlada por el cliente.

</> Ejemplos de código