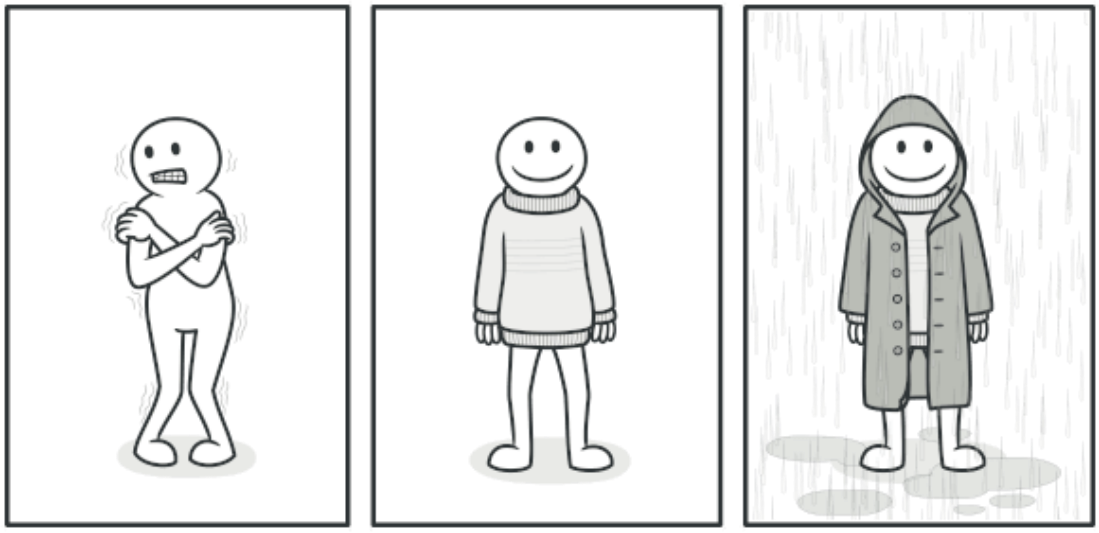
Decorator:

El Patrón Decorator permite agregar funcionalidades y responsabilidades a objetos de forma dinámica y transparente para el usuario; facilitando el problema, evitando tener que crear sucesivas clases que hereden de la primera incorporando la nueva funcionalidad, sino otras clases que la implementan y se asocian a la primera por medio de encapsular dentro de otro objeto, llamado Decorador, las nuevas responsabilidades.

Una analogía de esto seria vestir ropa. Cuando tienes frío, te cubres con un suéter. Si sigues teniendo frío a pesar del suéter, puedes ponerte una campera 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.

De forma similar el decorator permite a un objeto base agregarle diferentes decoradores por separado, permitiendo agregar capas al objeto base modificando su comportamiento.



Aplicabilidad:

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

Además, puede ser utilizado cuando no sea posible extender el comportamiento de un objeto utilizando la herencia.

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.

Ventajas:

- ✓ 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.

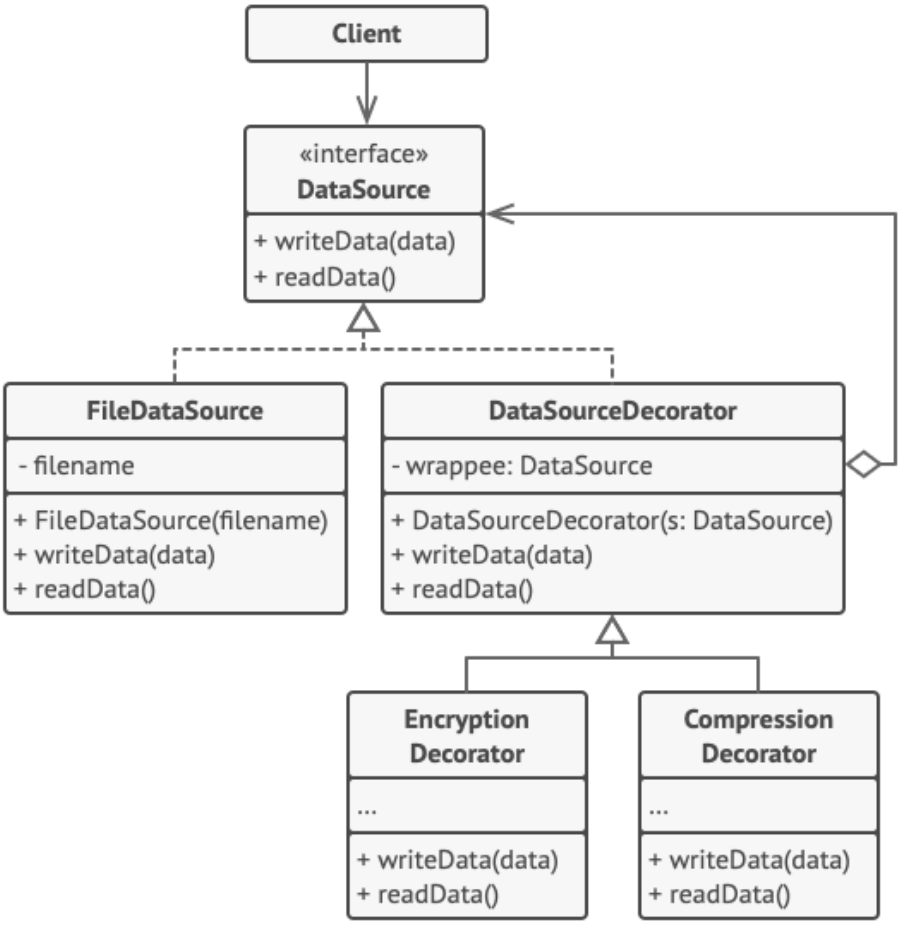
Desventajas:

- × Resulta difícil eliminar un capa específica de la pila de capas.
- × Es complejo implementar un decorador de tal forma que su comportamiento no dependa del orden en la pila de decoradores.

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.

Pseudocódigo:

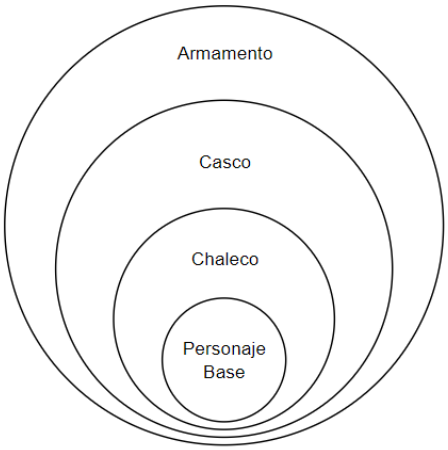


Ejemplo:

En el juego “Couter Strike”, En cada partida los personajes se encuentra en su forma base, pero lo que podemos hacer al comienzo de la partida es comprar una serie de objetos como chaleco, caso y diferentes armas; cada uno de estos objetos alterar la estadísticas de nuestro personaje, como puede ser el daño que recibe o la velocidad del mismo.

Si a la hora de desarrollar esto utilizaríamos herencia tendríamos que tener una clase para el personaje base, para el personaje con chaleco, para el personaje con casco, y deferentes clases para el personaje con las deferentes combinaciones posibles, y también cuando se quiere realizar un cabio en la skin del las armas o el personaje; esto provoca que se tenga una gran numero de clases. Por medio del patrón decorator nos permite tener una solución para poder generar estos objetos de una forma más dinámica sin tener que definir diferentes clases.

Lo que vamos a realizar en crear al personaje base sin ningún tipo de modificación, luego creamos cada decorador por separado; lo que estaríamos haciendo es crear es una organización por capas, en donde se la va ir poniendo capas al personaje de forma dinámica.



El personaje base es el componente a decorar, y cada decorador es un componente.

Código:

```
// interfaz básica a decorar
interface Personaje {
    fun Daño(): Double
    fun Velocidad(): Double
}
```

```
class PersonajeBase : Personaje{
    // Valores del personaje sin decoradores
    override fun Daño(): Double = 10.0
    override fun Velocidad(): Double = 100.0
}
```

```
// delegar el comportamiento de la interfaz `Personaje` a la instancia
`personaje` pasada en el constructor
abstract class Decorator(protected val personaje: Personaje):
Personaje by personaje
```

Decoradores:

```
class DecoradorChaleco (personaje: Personaje) : Decorator (personaje)
{
    override fun Daño() : Double {
        // el método de componente anulado por el decorador, cambiendo
el valor
        return personaje.Daño() / 1.5
    }
}
class DecoradorCasco (personaje: Personaje) : Decorator (personaje) {
    override fun Daño(): Double {
        return personaje.Daño() / 2
    }
}
class DecoradorSubFusil (personaje: Personaje) : Decorator (personaje)
{
    override fun Velocidad(): Double {
        return personaje.Velocidad() - 8
    }
}
class DecoradorFusil (personaje: Personaje) : Decorator (personaje) {
    override fun Velocidad(): Double {
        return personaje.Velocidad() - 12
    }
}
```

```
object Main {
    @JvmStatic

    fun main(args: Array<String>) {

    val Personaje: Personaje = PersonajeBase()
    val PersonajeConChaleco: Personaje = DecoradorChaleco(Personaje)
    val PersonajeConCasco: Personaje =
DecoradorCasco(PersonajeConChaleco)
    val PersonajeConSubFusil: Personaje =
DecoradorSubFusil(PersonajeConCasco)
    val PersonajeConFusil: Personaje =
DecoradorFusil(PersonajeConCasco)

    println("Daño que puede recibir: ${Personaje.Daño()}, Velocidad:
${Personaje.Velocidad()}%")  // Valores del personaje Base
    println("Daño que puede recibir: ${PersonajeConChaleco.Daño()},
Velocidad: ${PersonajeConChaleco.Velocidad()}%") // Valores del
personaje Base con el chaleco
    println("Daño que puede recibir: ${PersonajeConCasco.Daño()},
Velocidad: ${PersonajeConCasco.Velocidad()}%") // Valores del
personaje Base con el chaleco y casco
    println("Daño que puede recibir: ${PersonajeConSubFusil.Daño()},
Velocidad: ${PersonajeConSubFusil.Velocidad()}%") // Valores del
personaje Base con el chaleco, casco y sub fusil
    println("Daño que puede recibir: ${PersonajeConFusil.Daño()},
Velocidad: ${PersonajeConFusil.Velocidad()}%") // Valores del
personaje Base con el chaleco, casco y fusil

    }
}
```