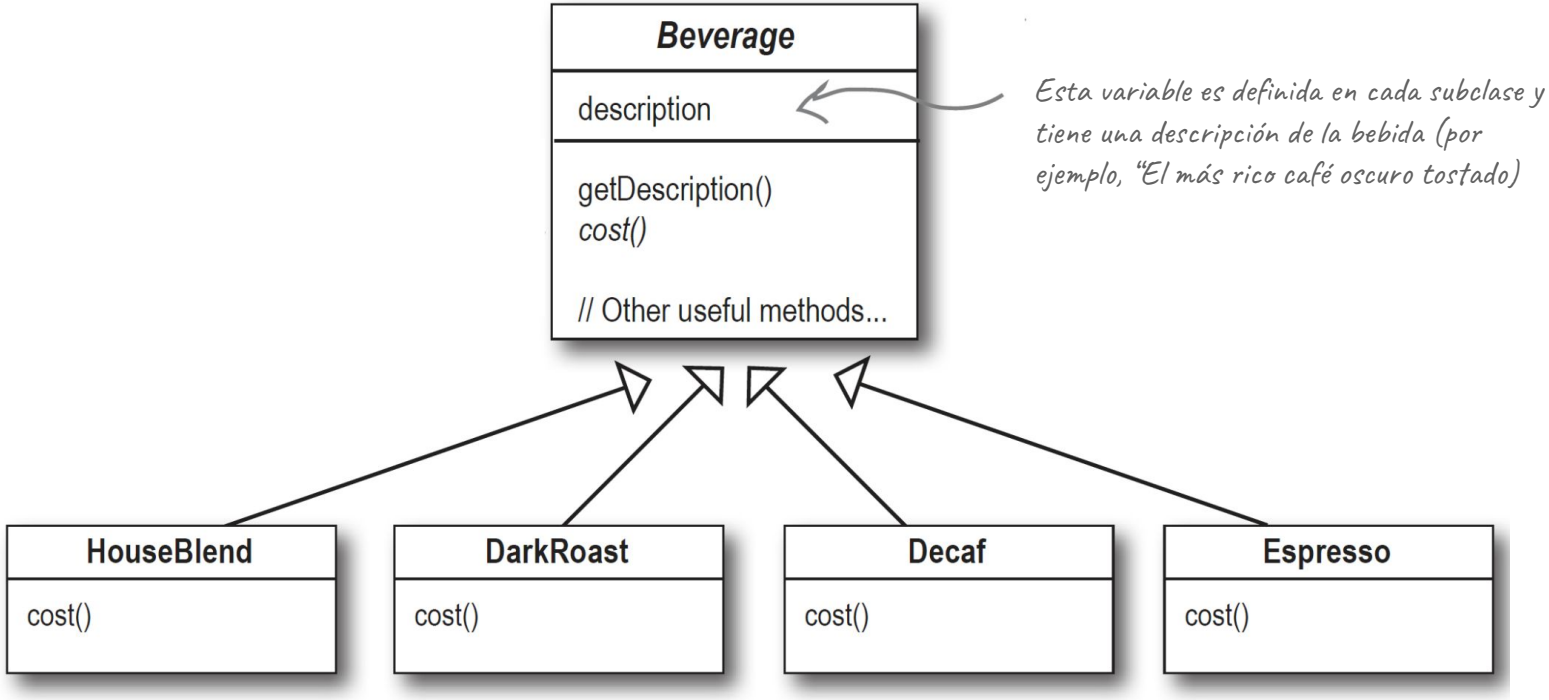
El Patrón Decorator

Qué bueno sería tener el poder de extender los objetos a tiempo de ejecución, en vez de a tiempo de compilación...

Bienvenidos a Starbuzz Coffee! ☕

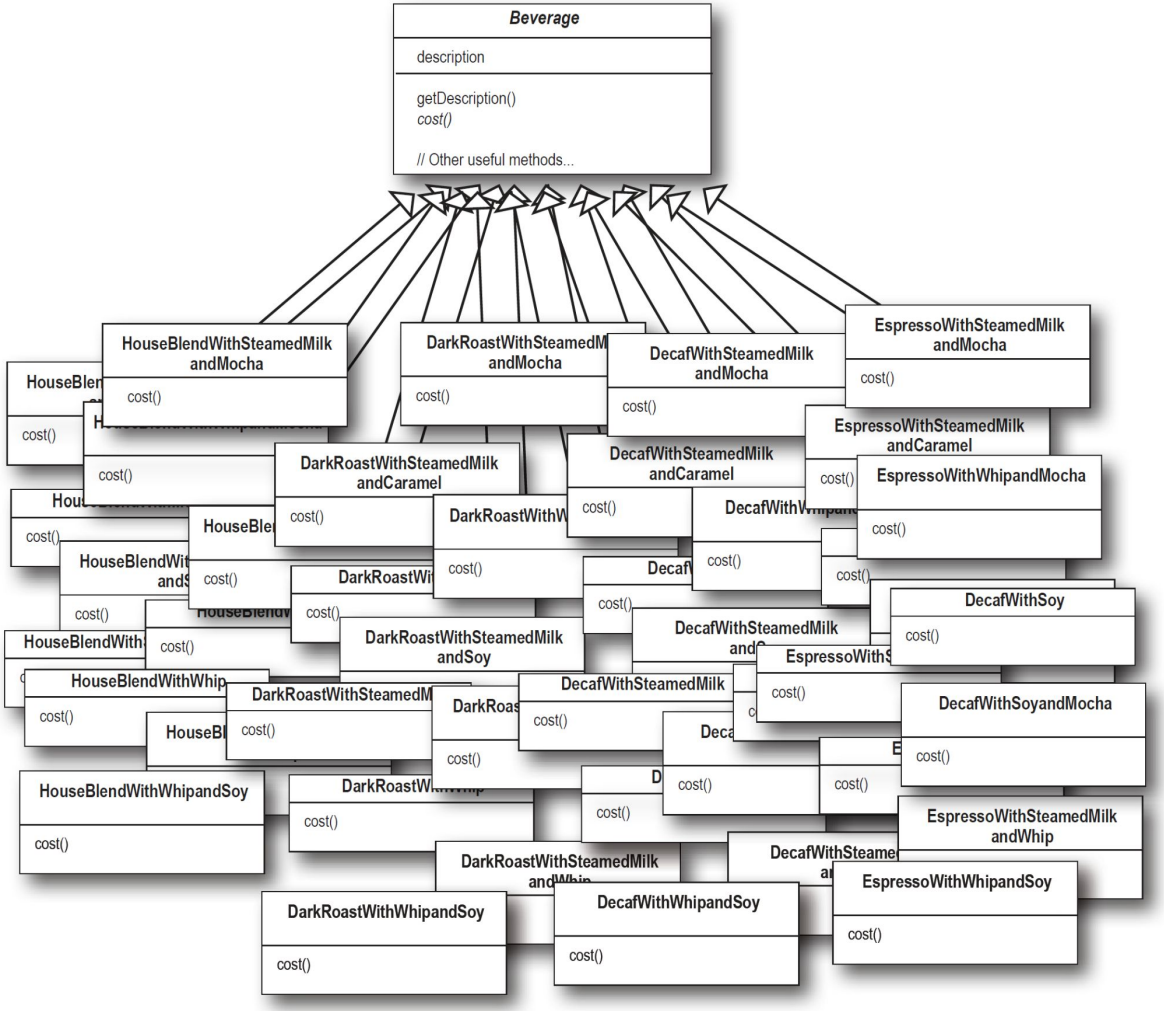
- Imaginemos que tenemos una cafetería, Starbuzz Coffee.
- Requisito inicial: Vender distintos tipos de café: House Blend, Dark Roast, Decaf, Espresso.
- La solución obvia es crear una clase abstracta Beverage y una subclase para cada tipo de café.

Modelando Starbuzz Coffee ☕



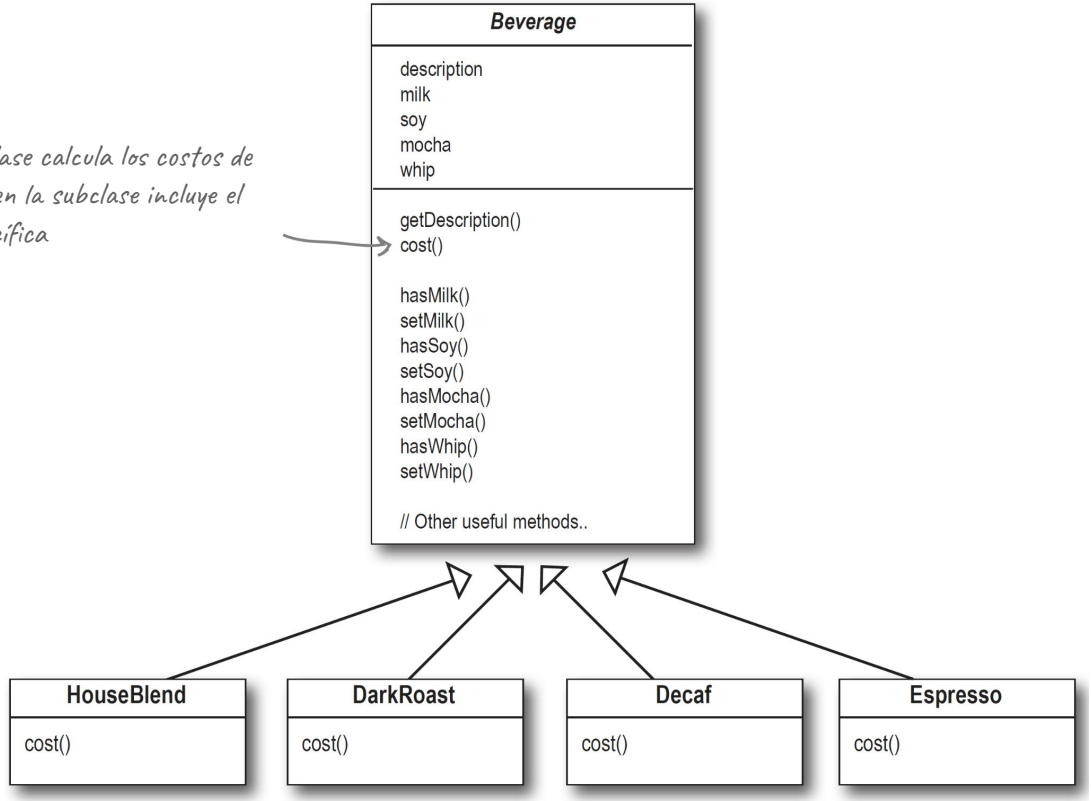
El Problema: La Explosión de Clases 💣💥

- Pero... ¿qué pasa cuando añadimos extras al café?
- Los clientes pueden pedir leche (Milk), soja (Soy), moca (Mocha) o crema (Whip).
- Si creáramos una subclase para cada combinación (DarkRoastWithMocha, HouseBlendWithSoyAndWhip, etc.), ¡tendríamos una explosión de subclases!


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
cost()
DecafWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andCaramel
cost()
DarkRoastWithSteamedMilk andCaramel
cost()
DecafWithSteamedMilk andCaramel
cost()
EspressoWithWhipandMocha
cost()
DecafWithSoy
cost()
DarkRoastWithSteamedMilk andSoy
DecafWithSteamedMilk
cost()
DecafWithSoyandMocha
cost()
HouseBlendWithWhipandSoy
cost()
EspressoWithSteamedMilk andWhip
DarkRoastWithWhipandSoy
cost()
DecafWithWhipandSoy
cost()
EspressoWithWhipandSoy
cost()

Un intento de solución

La función cost en la superclase calcula los costos de los extras, y la función cost en la subclase incluye el costo de la bebida base específica



Un Intento de Solución (y por qué falla)

Alternativa: Variables de Instancia Podríamos intentar añadir variables booleanas (`hasMilk`, `hasSoy`, etc.) en la clase base `Beverage`.

```
// Pseudo-código
public abstract class Beverage {
    boolean hasMilk;
    boolean hasSoy;
    // ...
    public double cost() {
        double total = 0;
        if (hasMilk()) { total += milkCost; }
        if (hasSoy()) { total += soyCost; }
        // ...
        return total;
    }
}
```

Problemas con este enfoque:

- **Rigidez:** El precio de los extras está en la superclase. Un cambio de precio nos obliga a modificar código de la superclase.
- **Nuevos condimentos:** Añadir un nuevo extra (ej. Caramelo) implica modificar la superclase, afectando a todas las subclases.
- **No aplicable a todos:** Algunas subclases (ej. `IcedTea`) heredarían métodos como `hasWhip()` que no tienen sentido para ellas.
- **Flexibilidad limitada:** ¿Cómo manejamos un "doble mocha"?

Este diseño **viola un principio fundamental**.

Principio de Diseño Open-Closed

Las clases deben estar abiertas para la extensión, pero cerradas para la modificación

- **Abierto para la extensión:** Debemos poder añadir nuevas funcionalidades o comportamientos a nuestro sistema sin problemas.
- **Cerrado para la modificación:** No deberíamos tener que cambiar código existente que ya ha sido probado y funciona.

Nuestro objetivo: Crear diseños que sean resilientes al cambio y lo suficientemente flexibles para incorporar nuevos requisitos sin alterar el código existente.

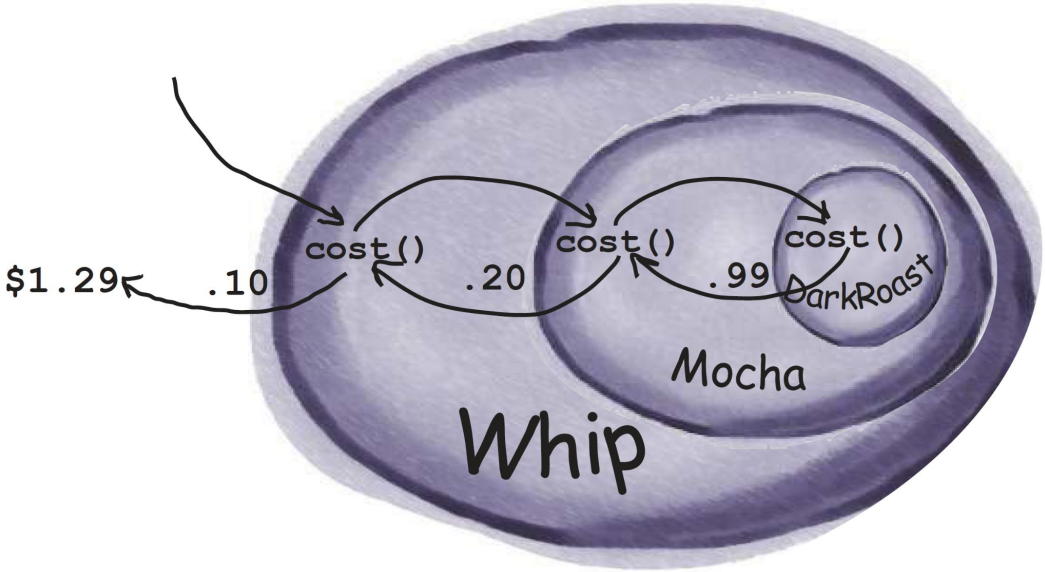
La Solución: "Envolver" Objetos

En lugar de usar la herencia para añadir comportamiento, ¡vamos a **decorar** nuestros objetos en tiempo de ejecución!

Las técnicas de decoración nos dan el poder de darles a nuestros objetos nuevas responsabilidades sin tener que hacer cambios en las clases subyacentes.

La idea es simple:

1. **Empezamos** con un objeto base (nuestro `DarkRoast`).
2. Lo **envolvemos** con un objeto `Mocha` (el primer decorador).
3. **Envolvemos** el resultado con un objeto `Whip` (el segundo decorador).
4. Para calcular el costo, simplemente llamamos al método `cost()` del decorador más externo



El cálculo del costo se delega a través de la cadena de "envolturas":

1. Llamamos a `cost()` en el decorador más externo (`Whip`).
2. `Whip` llama al `cost()` del objeto que envuelve (`Mocha`).
3. `Mocha` llama al `cost()` del objeto que envuelve (`DarkRoast`).
4. `DarkRoast` retorna su costo base (ej. $0.99).
5. `Mocha` recibe ese valor, le suma su propio costo (ej. $0.20) y retorna el nuevo total ($1.19).
6. `Whip` recibe ese total, le suma su costo (ej. $0.10) y retorna el resultado final ($1.29).

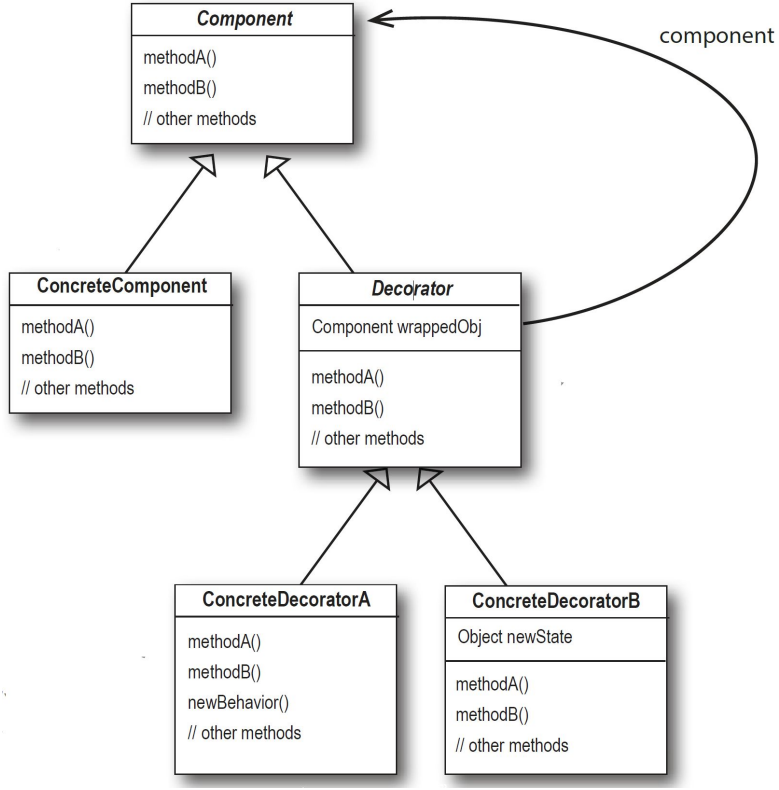
Definición Formal del Patrón Decorator: Definición Formal (GoF)

El Patrón Decorator añade responsabilidades adicionales a un objeto de forma dinámica. Los decoradores proporcionan una alternativa flexible a la herencia para extender la funcionalidad.

Características Clave:

- Los decoradores tienen el **mismo supertipo** que los objetos que decoran. Esto es crucial para la transparencia.
- Se puede envolver un objeto con **uno o más** decoradores.
- El decorador añade su propio comportamiento **antes y/o después** de delegar la llamada al objeto que envuelve.
- Los objetos pueden ser decorados **dinámicamente en tiempo de ejecución**.

Patrón decorador: diagrama de clases



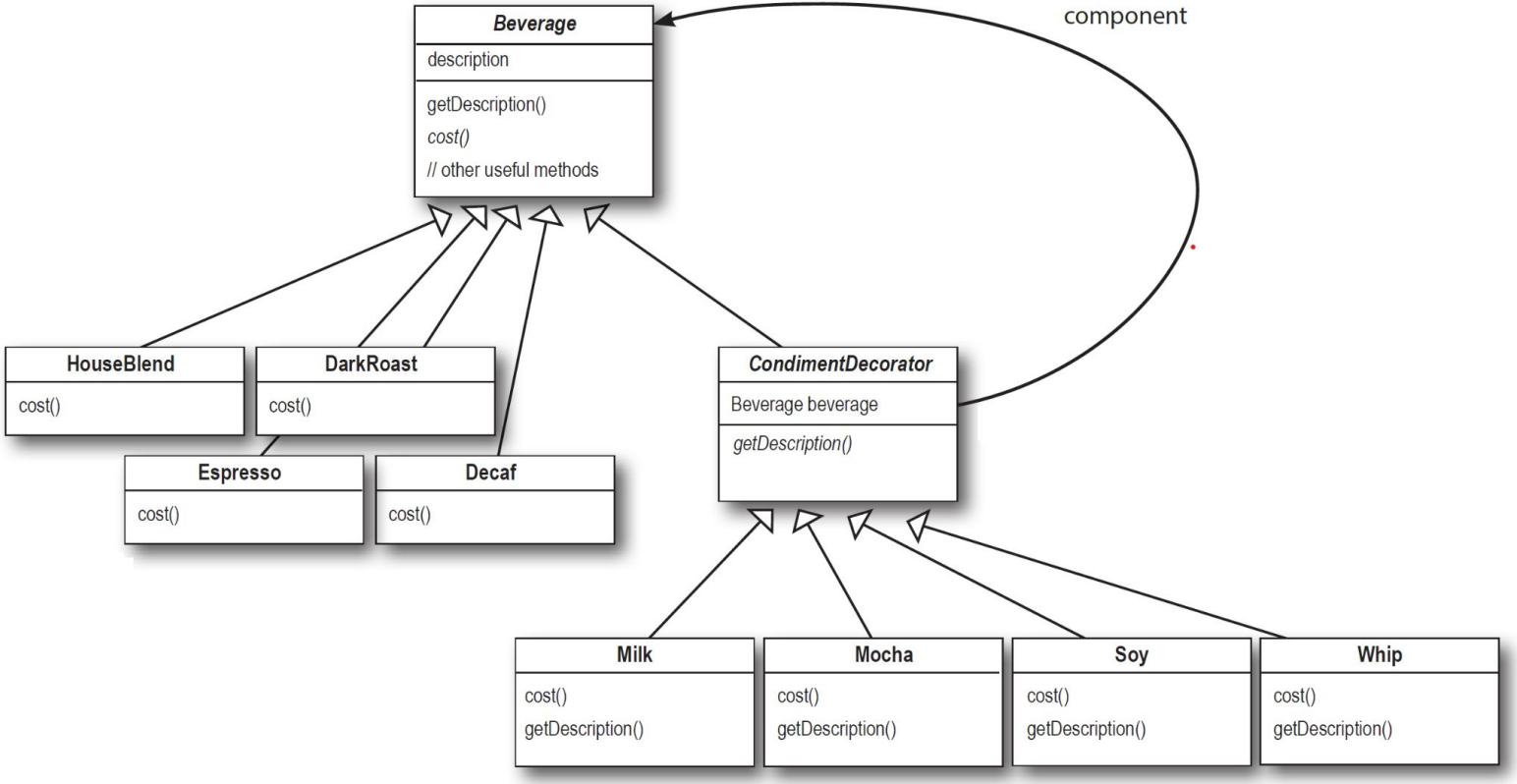
Estructura y Participantes (UML): Diagrama de Clases UML

Participantes:

- **Component (`Beverage`):** Define la interfaz para los objetos que pueden ser decorados.
- **ConcreteComponent (`DarkRoast`, `HouseBlend`):** Es el objeto base al que le añadimos responsabilidades.
- **Decorator (`CondimentDecorator`):** Mantiene una referencia al objeto `Component` que envuelve (composición) y conforma a la misma interfaz `Component` (herencia).
- **ConcreteDecorator (`Mocha`, `Whip`, `Soy`):** Implementa la funcionalidad o estado extra que se añade al `ConcreteComponent`.

El Diagrama Aplicado a Starbuzz: El Diseño de Starbuzz con Decorator

- **`Beverage`** es nuestro `Component` abstracto.
- **`HouseBlend`**, **`Espresso`**, etc., son los `ConcreteComponent`.
- **`CondimentDecorator`** es el `Decorator` abstracto. Hereda de `Beverage` para tener el mismo tipo.
- **`Milk`**, **`Mocha`**, **`Whip`**, etc., son los `ConcreteDecorator`. Cada uno añade el costo y la descripción del condimento.


Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
Beverage beverage
getDescription()
Milk
cost()
getDescription()
Mocha
cost()
getDescription()
Soy
cost()
getDescription()
Whip
cost()
getDescription()

Herencia vs. Composición: Herencia para el Tipo, Composición para el Comportamiento

Una pregunta común: *"¿Pero no estamos usando herencia?"*

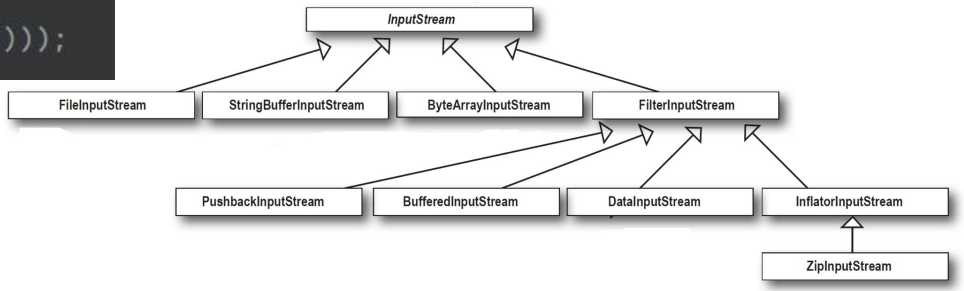
- **SÍ**, pero de una manera muy específica. `CondimentDecorator` hereda de `Beverage` **solo para tener el tipo correcto**. Esto permite que un decorador pueda "hacerse pasar" por un `Beverage` y envolver a otros.
- El **comportamiento adicional no se hereda**, se obtiene **componiendo** objetos. Un decorador `Mocha` **TIENE UN** `Beverage` que envuelve.
- Esta combinación nos da la flexibilidad de mezclar y combinar responsabilidades en tiempo de ejecución, algo que la herencia estática no permite.

Un Ejemplo Clásico: Las Clases de I/O de Java

El paquete `java.io` está construido en gran parte sobre el patrón Decorator.

- **Component:** `InputStream`
- **ConcreteComponent:** `FileInputStream` (lee bytes de un archivo).
- **Decorator:** `FilterInputStream` (el decorador abstracto).
- **ConcreteDecorator:**
  - `BufferedInputStream`: Añade la funcionalidad de buffering para mejorar el rendimiento.
  - `ZipInputStream`: Añade la capacidad de leer datos de un archivo ZIP.

```
// Se envuelve un objeto en múltiples decoradores
InputStream in = new LowerCaseInputStream(
                     new BufferedInputStream(
                         new FileInputStream("test.txt")));
```



Ventajas y Desventajas

Ventajas 👍:

- **Cumple con el Principio Abierto/Cerrado:** Podemos añadir nuevos decoradores sin modificar las clases existentes.
- **Flexibilidad:** Añade y quita responsabilidades de un objeto en tiempo de ejecución.
- **Evita la explosión de subclases:** Ofrece una alternativa más manejable que la herencia para múltiples funcionalidades.

Desventajas 👎:

- **Aumento de la complejidad:** Puede resultar en un gran número de clases pequeñas, lo que puede hacer que el código sea más difícil de entender a primera vista.
- **Código de instanciación complejo:** Crear un objeto con múltiples decoradores puede ser engorroso (aunque patrones como Factory o Builder pueden ayudar).
- **Dependencia del tipo concreto:** El código cliente que depende del tipo específico de un objeto (ej. `if (beverage instanceof DarkRoast)`) puede romperse si se le pasa un objeto decorado.

Ejercicio Práctico en Python

Ejercicio Práctico en Python

Vamos a llevar este diseño a la práctica implementando el sistema de Starbuzz Coffee en Python.

Estructura del Repositorio:

- `beverages.py`: Contendrá las clases `Component` (`Beverage`) y `ConcreteComponent` (`Espresso`, `DarkRoast`, etc.).
- `condiments.py`: Definirá el `Decorator` (`CondimentDecorator`) y los `ConcreteDecorator` (`Mocha`, `Whip`, etc.).
- `main.py`: El script principal donde crearemos y "serviremos" nuestros cafés decorados.

Objetivos de la Clase

Al finalizar esta clase, serás capaz de:

- **Definir el propósito y la intención del patrón Decorator.**
- **Identificar a los participantes clave del patrón y describir sus roles.**
- **Explicar cómo el patrón utiliza tanto la herencia como la composición para lograr su objetivo.**
- **Relacionar el patrón Decorator con el Principio de Diseño Abierto/Cerrado.**
- **Analizar las ventajas y desventajas de su uso.**
- **Reconocer escenarios donde la aplicación del patrón es apropiada.**