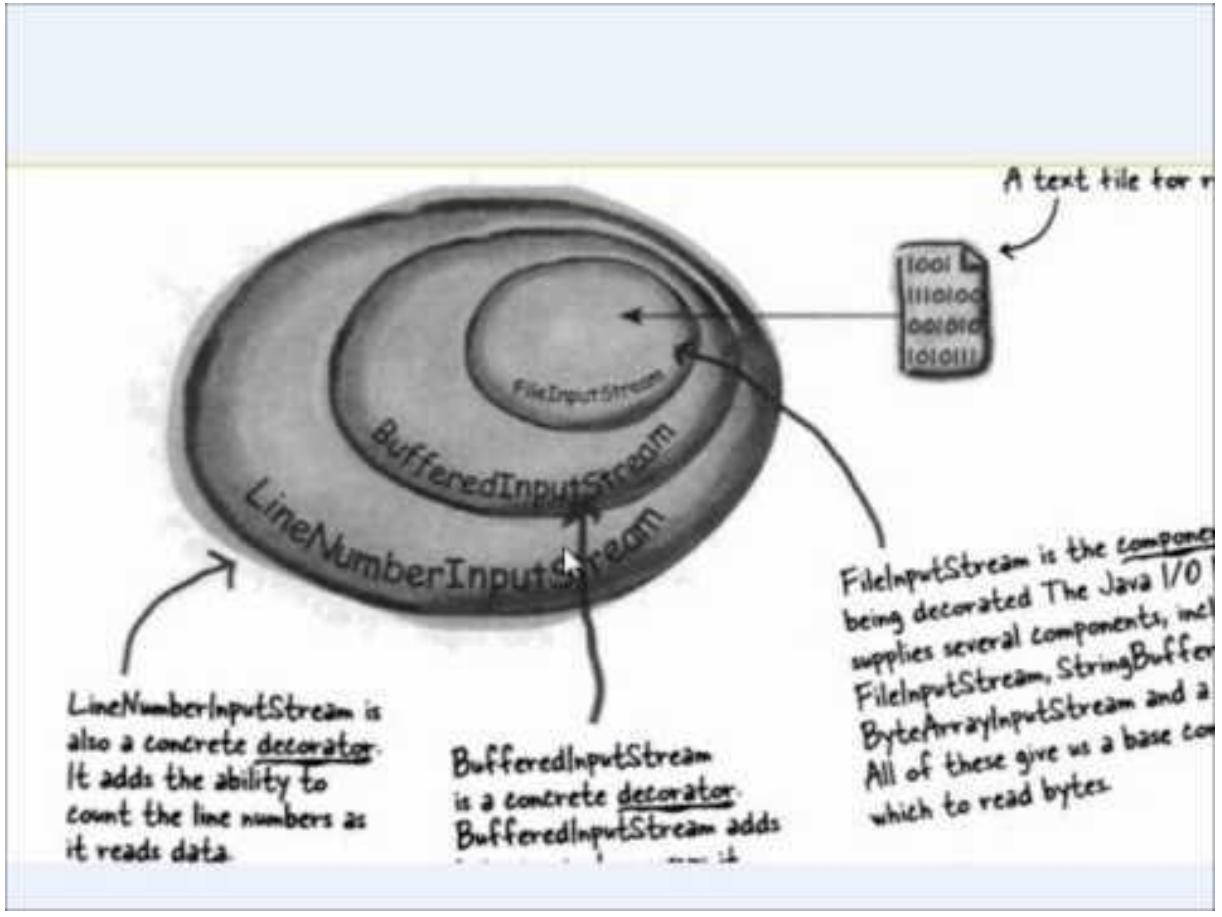
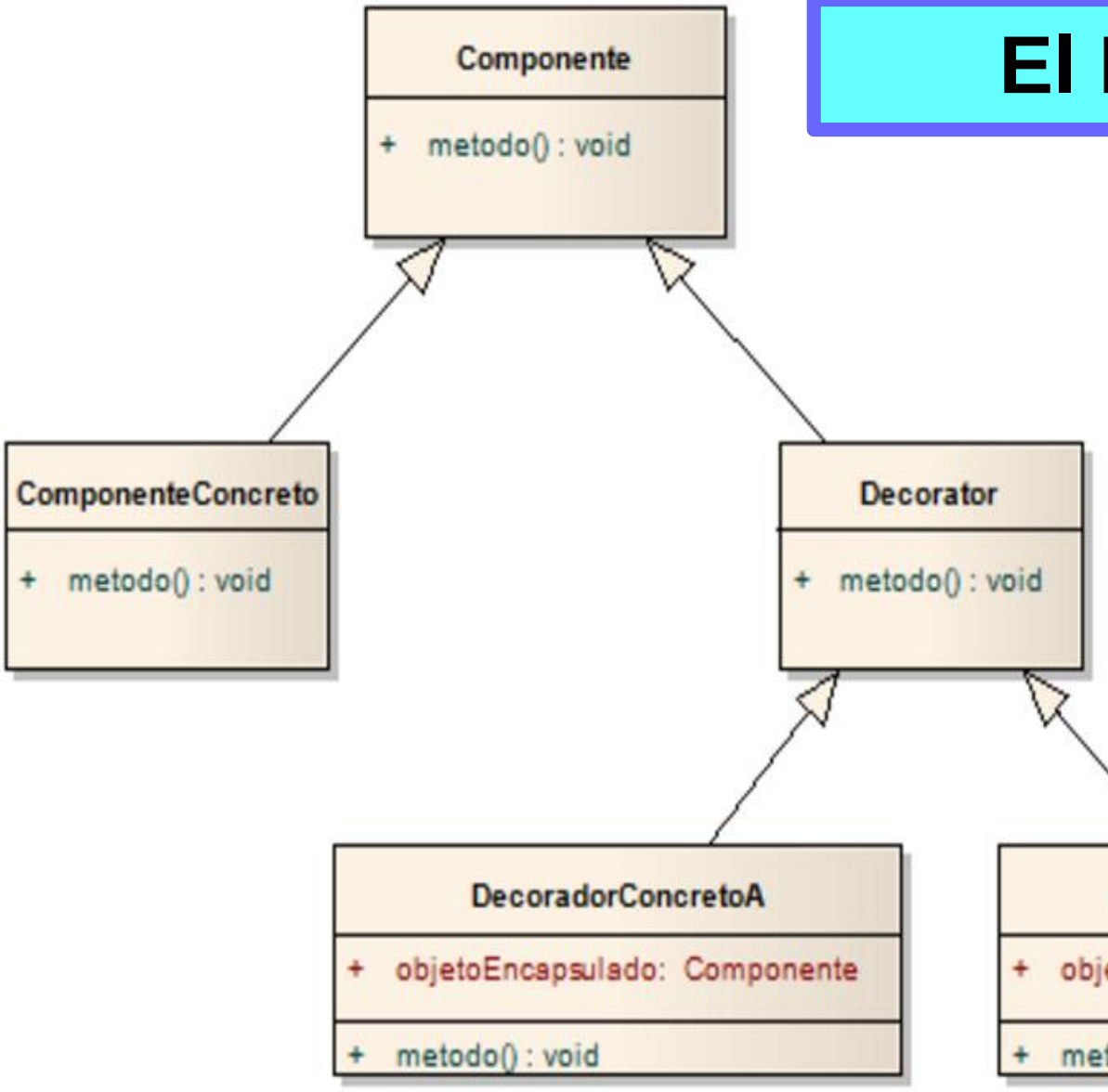
El Patrón Decorator



El Patrón Decorator

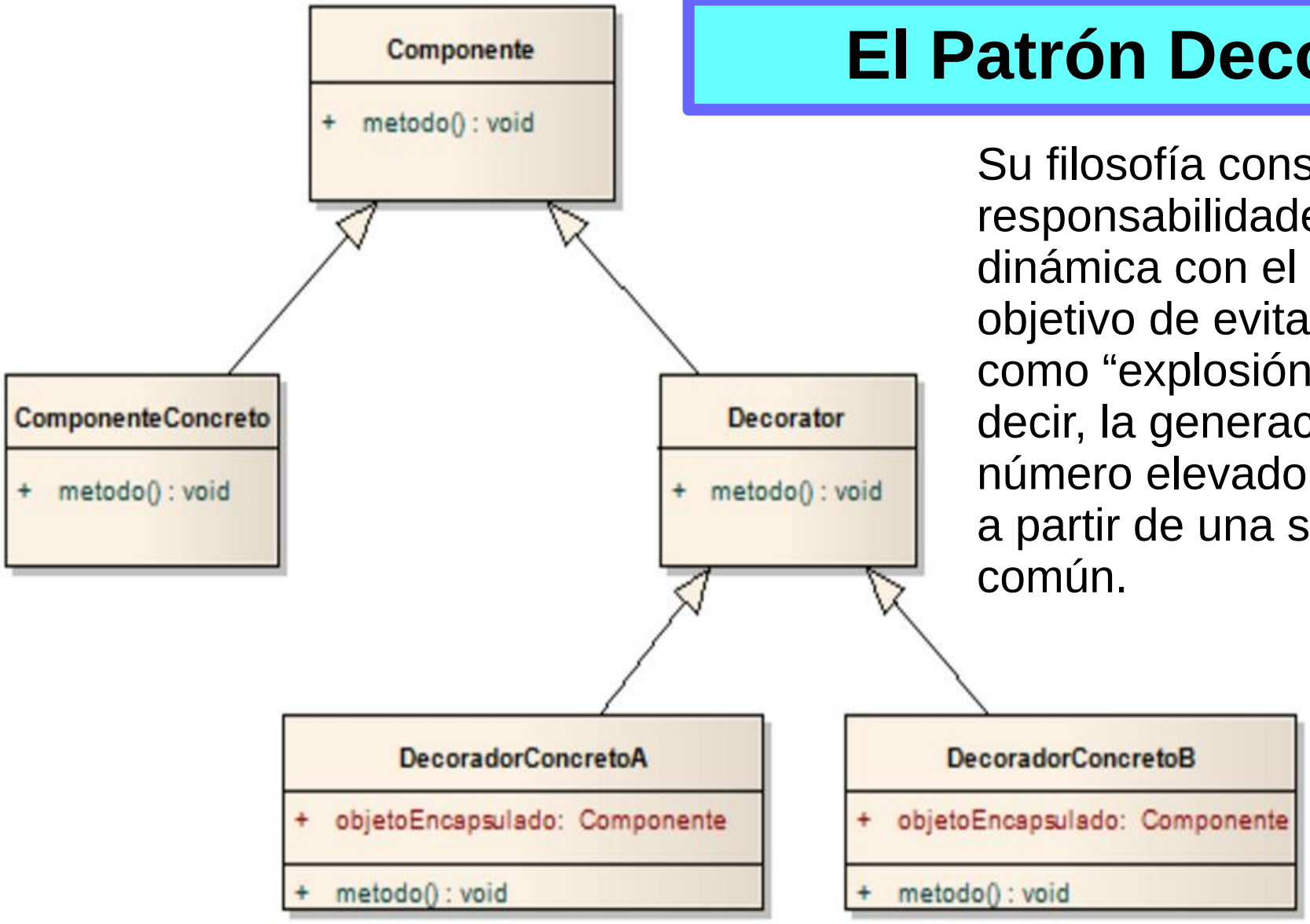


Objetivo:

Añadir responsabilidades a un objeto de forma dinámica.

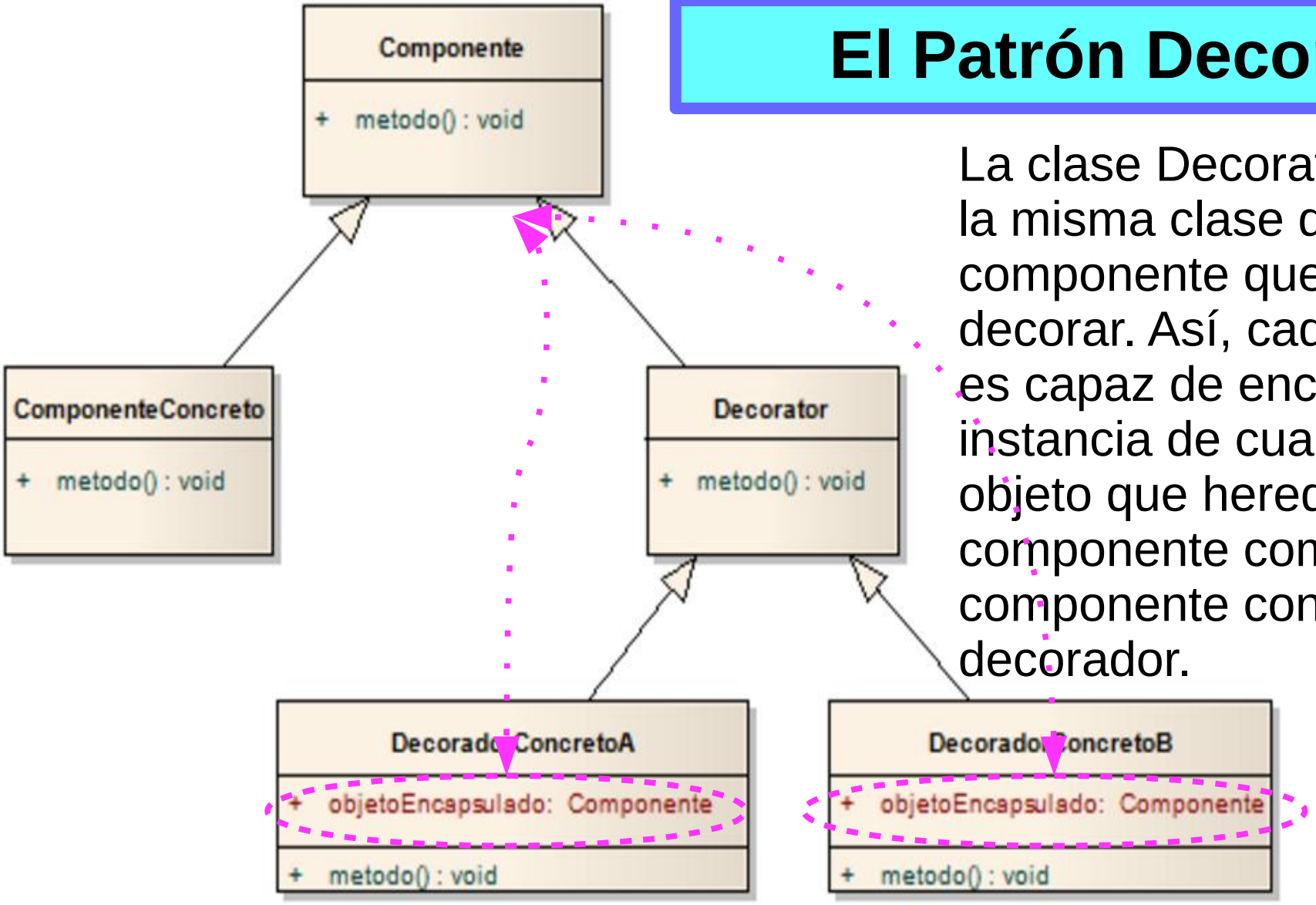
Los decoradores proporcionan una alternativa flexible a la herencia para extender funcionalidad.

El Patrón Decorator



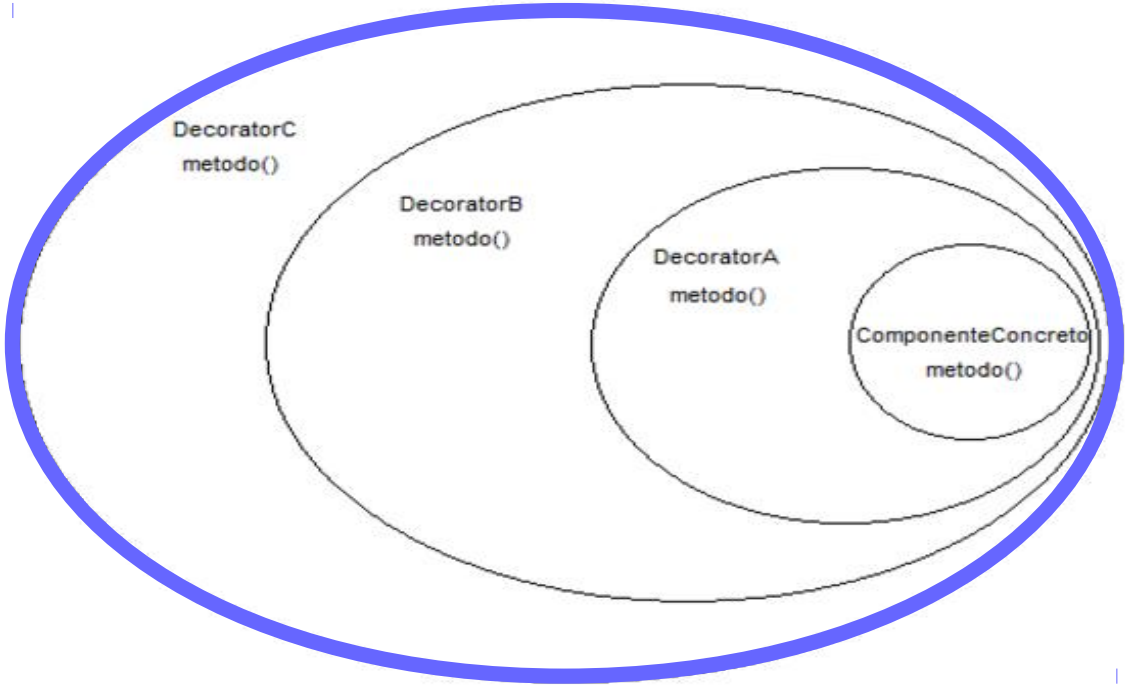
Su filosofía consiste en añadir responsabilidades de forma dinámica con el principal objetivo de evitar la conocida como “explosión de clases”, es decir, la generación de un número elevado de subclases a partir de una superclase común.

El Patrón Decorator



La clase Decorator hereda de la misma clase que el componente que se quiere decorar. Así, cada decorador es capaz de encapsular una instancia de cualquier otro objeto que herede del componente común, bien un componente concreto u otro decorador.

El Patrón Decorator



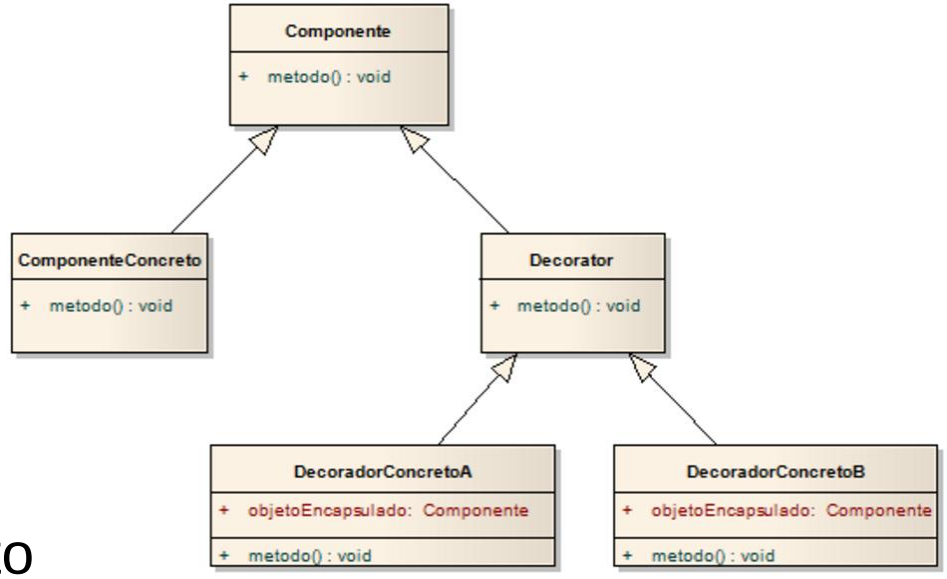
El objeto con el que el objeto cliente interactuará será aquel que se encuentre en la capa más externa (en este caso, DecoratorC), que se encargará de acceder a los objetos contenidos e invocar su funcionalidad, que será devuelta a las capas exteriores.

La encapsulación puede ser iterativa, de modo que un componente concreto puede ser encapsulado por un decorador, que a su vez puede ser encapsulado por otro decorador, y así sucesivamente, añadiendo nueva funcionalidad en cada uno de los pasos.
Resumiendo: el patrón Decorator sustituye la herencia por un proceso iterativo de composición.

El Patrón Decorator

Para comenzar, por tanto, debemos tener claros los siguientes conceptos sobre este patrón:

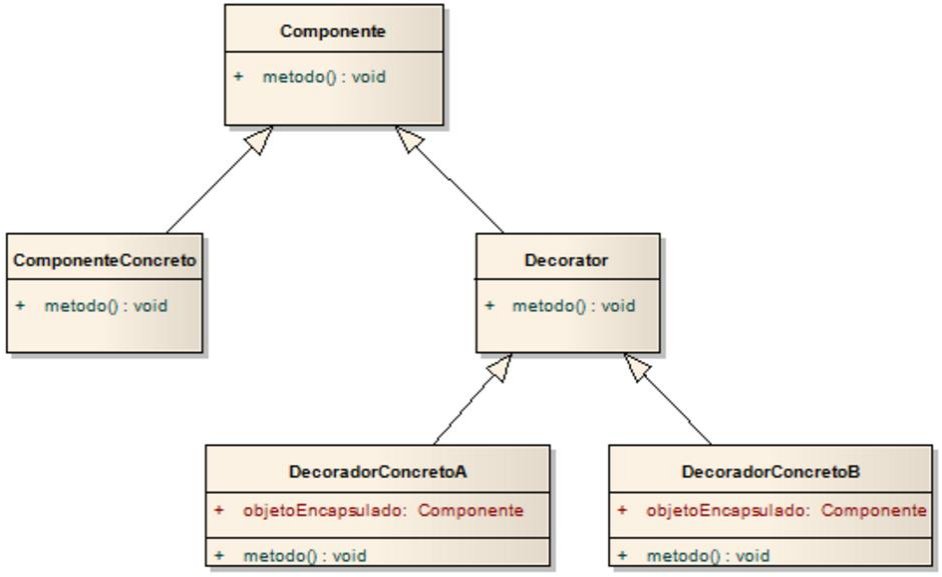
- Un decorador hereda de la misma clase que los objetos que tendrá que decorar.
- Es posible utilizar más de un decorador para encapsular un mismo objeto.
- El objeto decorador añade su propia funcionalidad, bien antes, bien después, de delegar el resto del trabajo en el objeto que está decorando.



- Los objetos pueden decorarse en cualquier momento, por lo que es posible decorar objetos de forma dinámica en tiempo de ejecución.

El Patrón Decorator

La razón por la que la clase Decorator hereda de la misma clase que el objeto que tendrá que decorar no es la de añadir funcionalidad, sino la de asegurarse de que ambos comparten el mismo tipo y puedan intercambiarse: *un decorador podrá sustituir a un objeto decorado, basándonos en el principio SOLID del Principio de sustitución de Liskov.*



Ejemplo
Patrón Decorator

Ejemplo del Patrón Decorator

Se desea modelar diferentes tipos de vehículos (“Berlina” y “Monovolumen”) que funcionan con diferentes tipo de motor (“Diesel”, “Gasolina”, “Inyeccion”, “CommonRail” y “Turbo”).

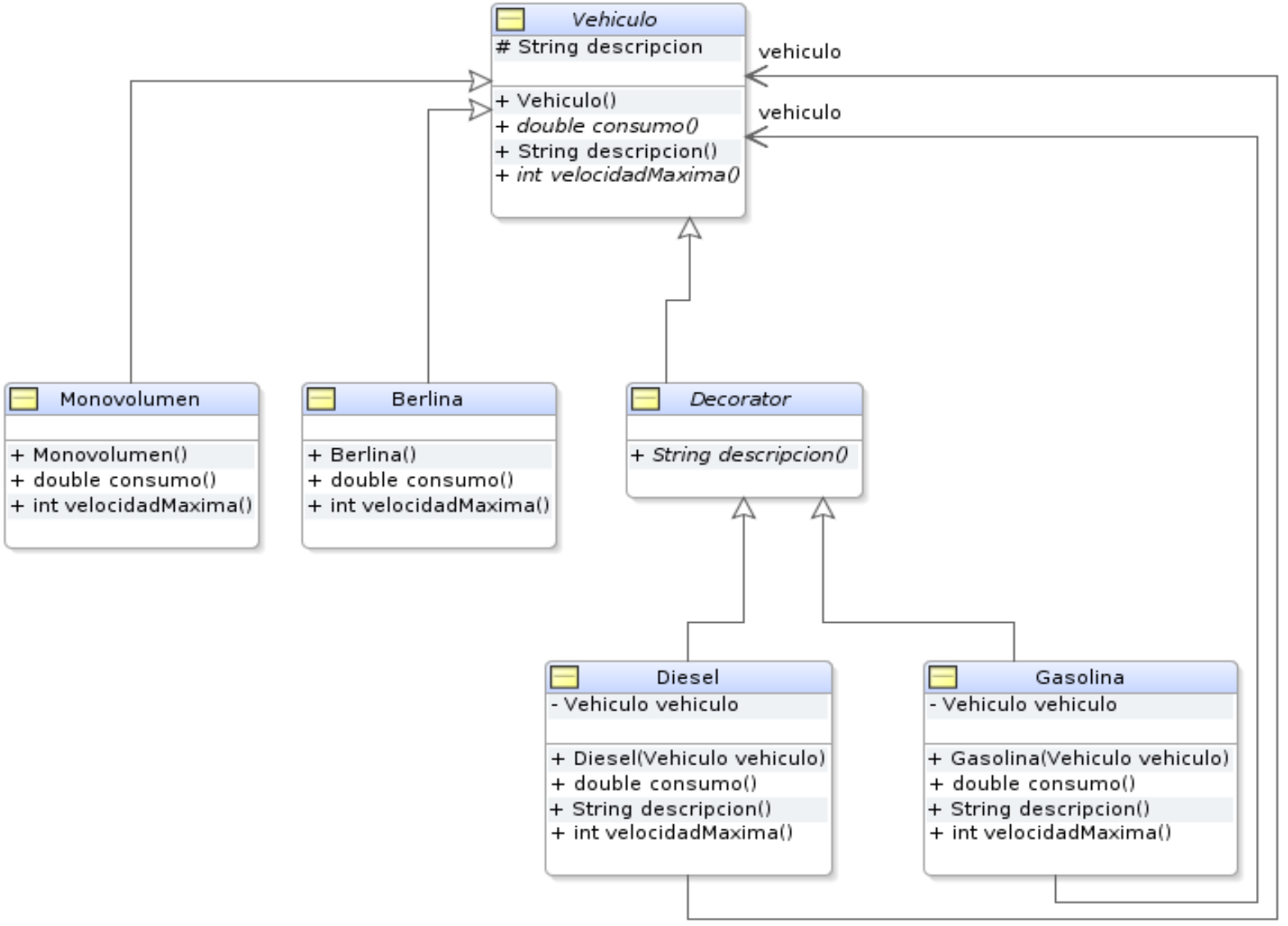
A priori, pareciera ser que la solución está en generar una cantidad de clases que contemple la combinación de tipos de vehículo con tipos de motor. Esto daría un total de 10 clases (hojas), cinco para Berlina y cinco para Monovolumen.

El Patrón Decorator nos permite evitar esta explosión de clases.

Utilizaremos una clase abstracta, llamada Vehiculo, del que heredarán las clases funcionales a las que llamaremos “Berlina” y “Monovolumen”, y los decoradores, que se limitarán a añadir funcionalidad a estas clases funcionales. Los decoradores que diseñaremos serán “Diesel”, “Gasolina”, “Inyeccion”, “CommonRail” y “Turbo”.

Estos decoradores se caracterizarán por:

- Disponer de una referencia a un vehículo que será inyectada en el constructor.
- Modificar el funcionamiento original de la clase que decoran, sobrecargando los métodos y llamando a los métodos de las clases encapsuladas para modificar su información o funcionamiento.


Vehiculo
String descripcion
+ Vehiculo()
+ double consumo()
+ String descripcion()
+ int velocidadMaxima()
vehiculo
vehiculo
Monovolumen
+ Monovolumen()
+ double consumo()
+ int velocidadMaxima()
Berlina
+ Berlina()
+ double consumo()
+ int velocidadMaxima()
Decorator
+ String descripcion()
Diesel
- Vehiculo vehiculo
+ Diesel(Vehiculo vehiculo)
+ double consumo()
+ String descripcion()
+ int velocidadMaxima()
Gasolina
- Vehiculo vehiculo
+ Gasolina(Vehiculo vehiculo)
+ double consumo()
+ String descripcion()
+ int velocidadMaxima()


```java
public abstract class Vehiculo
{
    public Vehiculo()
    {
        super();
    }
    // Atributo común a todos los objetos que heredarán de esta clase
        protected String descripcion = "Vehículo genérico";

        // Método no abstracto que devolverá el contenido de la descripción
        public String descripcion()
        {
            return descripcion;
        }

        // Métodos abstractos
        public abstract int velocidadMaxima();
        public abstract double consumo();
}
```

```java
public class Monovolumen extends Vehiculo
{
    public Monovolumen()
    {
        descripcion = "Monovolumen";
    }

    @Override
    public int velocidadMaxima()
    {
        return 160;
    }

    @Override
    public double consumo()
    {
        return 7.5;
    }
}
```

```java
public class Berlina extends Vehiculo
{
    public Berlina()
    {
        descripcion = "Berlina";
    }

    @Override
    public int velocidadMaxima()
    {
        return 180;
    }

    @Override
    public double consumo()
    {
        return 6.2;
    }
}
```


```java
public abstract class Decorator extends Vehiculo
{
    //reescribe Vehiculo.descripcion() y lo transforma en abstracto
    public abstract String descripcion();
}
```

```java
public class Gasolina extends Decorator
{
    // Instancia de la clase vehiculo
    private Vehiculo vehiculo;

    // Constructor que recibe el vehículo que encapsulará el decorator
    public Gasolina(Vehiculo vehiculo)
    {
        this.vehiculo = vehiculo;
    }

    // Los métodos utilizan la información del objeto encapsulado y le
    // incorporan su propia funcionalidad.
    @Override
    public String descripcion()
    {
        return vehiculo.descripcion() + "Gasolina";
    }
```

```java
    // Un vehículo gasolina proporciona más potencia, por lo que "decora"
    // vehiculo añadiendo mayor velocidad máxima
    @Override
    public int velocidadMaxima()
    {
        return vehiculo.velocidadMaxima() + 60;
    }

    // La gasolina es menos energética que el diesel, por lo que el consumo
    // de combustible es mayor. Decoraremos el vehículo añadiéndole un consumo
    // de 1.2 litros adicionales a los 100 km.
    @Override
    public double consumo()
    {
        return vehiculo.consumo() + 1.2;
    }
}
```

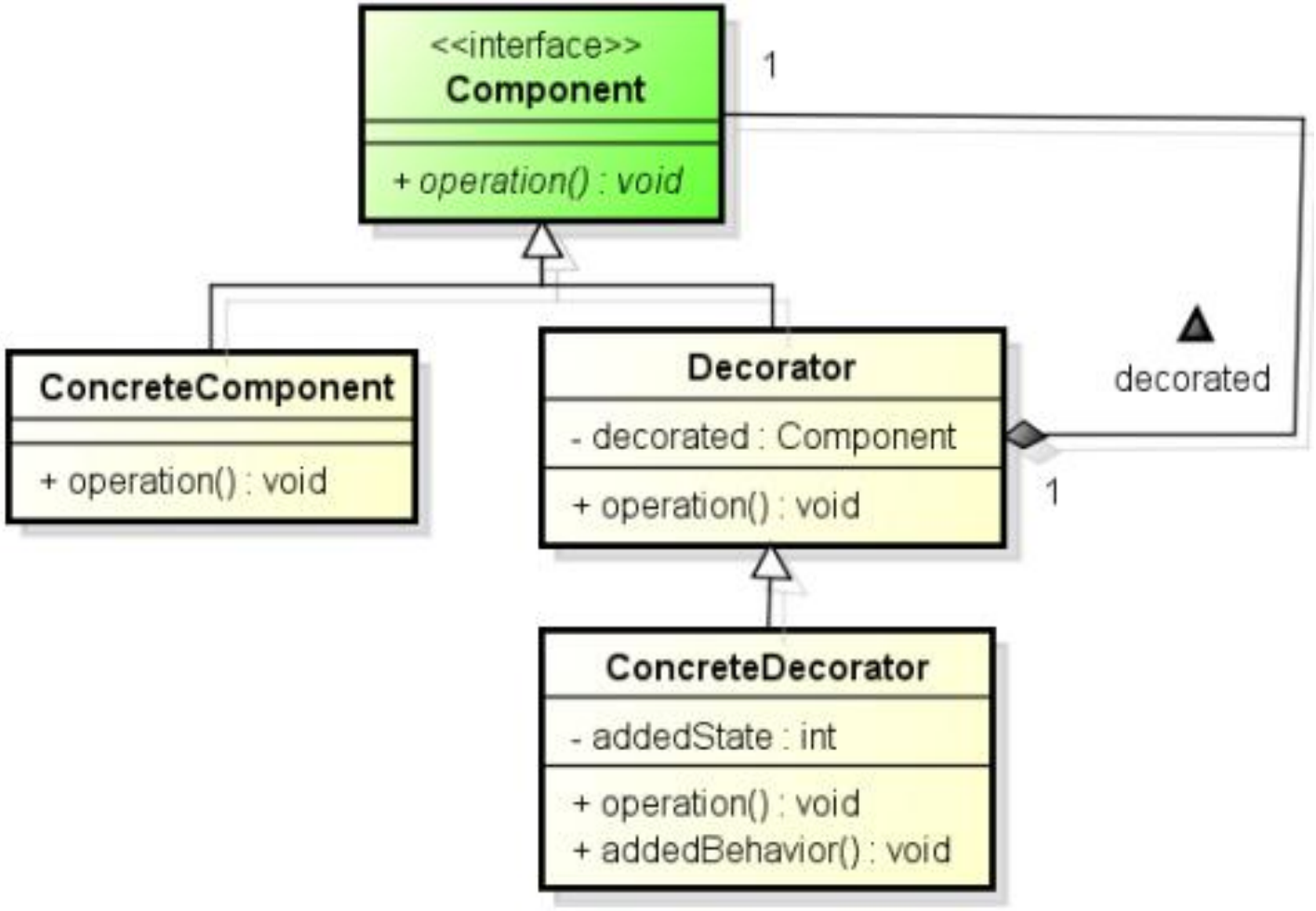
```java
public class Diesel extends Decorator
{
    // Instancia de la clase vehiculo
    private Vehiculo vehiculo;

    // Constructor que recibe el vehículo que encapsulará el decorator
    public Diesel(Vehiculo vehiculo)
    {
        this.vehiculo = vehiculo;
    }

    // Los métodos utilizan la información del objeto encapsulado y le
    // incorporan su propia funcionalidad.
    @Override
    public String descripcion()
    {
        return vehiculo.descripcion() + " Diesel";
    }

    @Override
    public int velocidadMaxima()
    {
        return vehiculo.velocidadMaxima() + 20;
    }

    @Override
    public double consumo()
    {
        return vehiculo.consumo() - 0.8;
    }
}
```

Decorator otra vez, pero más simple

Contexto: se quiere agregar nuevas responsabilidades a una clase.

Problema: cómo agregar nuevo comportamiento a una clase sin modicarla?

Solución: crear una clase Decorador que contiene a un Decorado. Los Decoradores son wrappers. Son del mismo tipo que los objetos que decoran, pero añaden nuevo comportamiento.

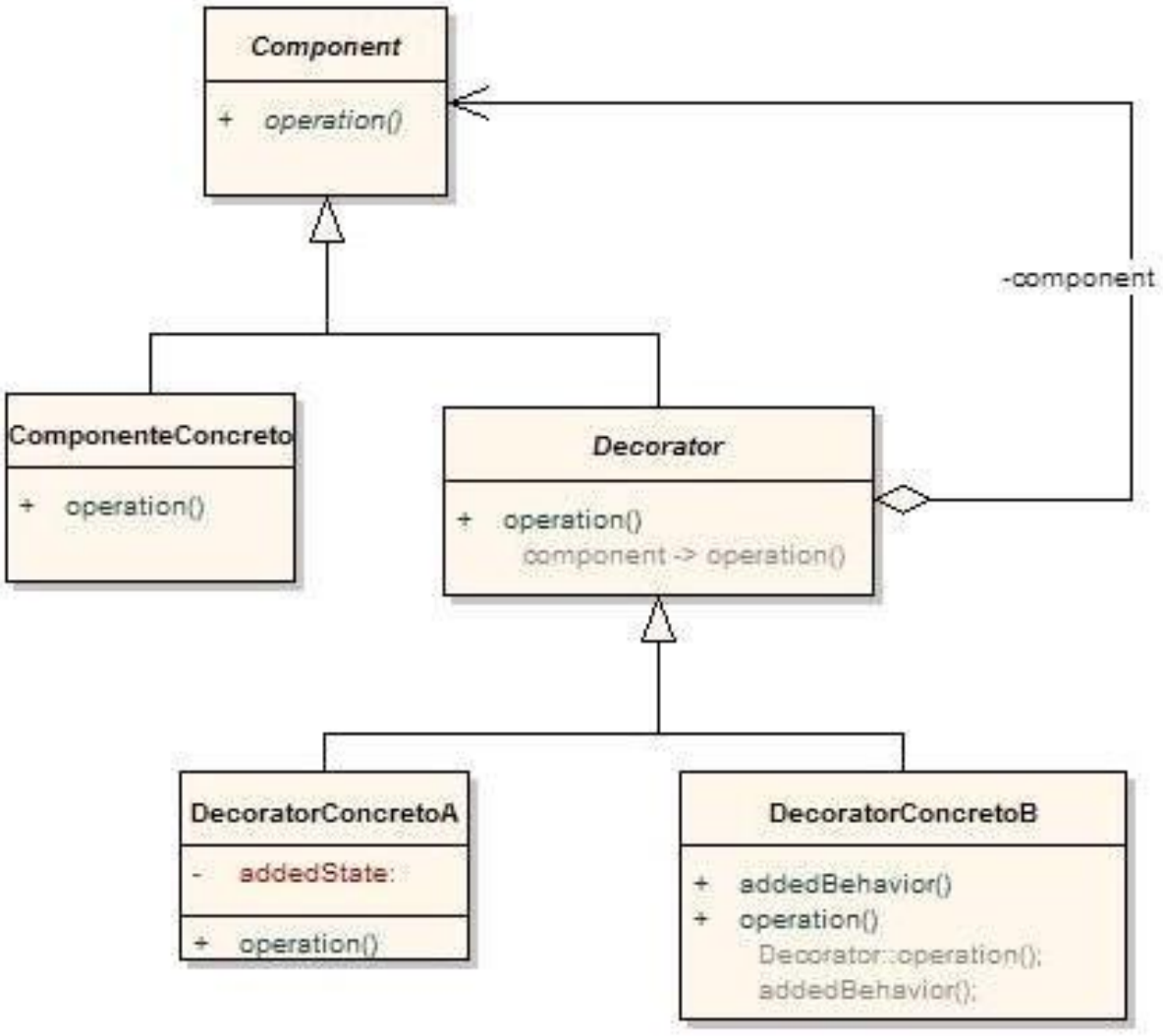

<<interface>>
Component
+ operation() : void
1
decorated
ConcreteComponent
+ operation() : void
Decorator
- decorated : Component
+ operation() : void
1
ConcreteDecorator
- addedState : int
+ operation() : void
+ addedBehavior() : void

Decorator otra vez

El patrón decorator permite añadir responsabilidades a objetos concretos de forma dinámica. Los decoradores ofrecen una alternativa más flexible que la herencia para extender las funcionalidades.

Es conocido como Wrapper (igual que el patrón Adapter).

A veces se desea adicionar responsabilidades a un objeto pero no a toda la clase. Las responsabilidades se pueden adicionar por medio de los mecanismos de Herencia, pero este mecanismo no es flexible porque la responsabilidad es adicionada estáticamente. La solución flexible es la de rodear el objeto con otro objeto que es el que adiciona la nueva responsabilidad. Este nuevo objeto es el Decorator.



Component: define la interface de los objetos a los que se les pueden adicionar responsabilidades dinámicamente.

ComponenteConcreto: define el objeto al que se le puede adicionar una responsabilidad.

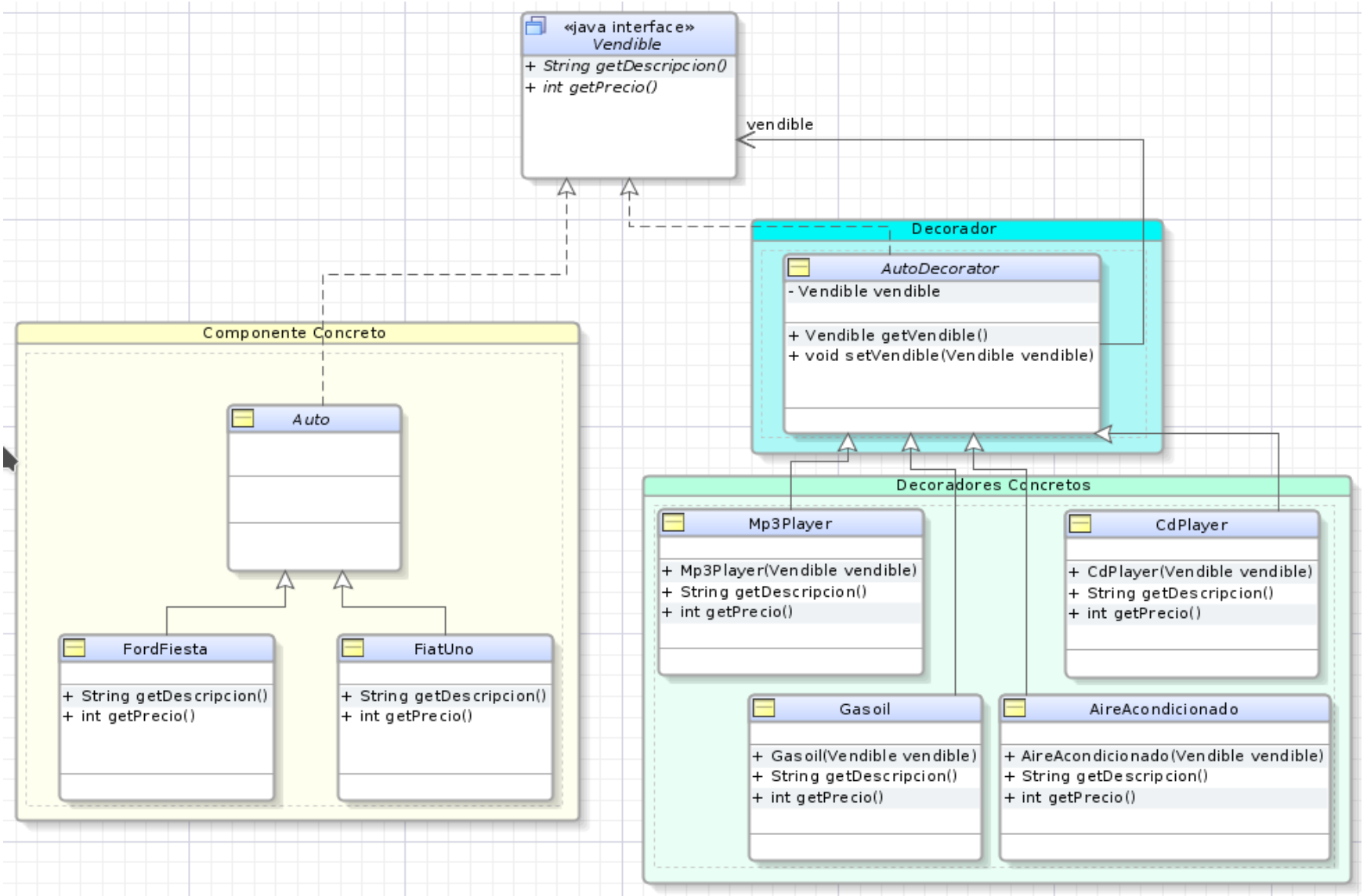
Decorator: mantiene una referencia al objeto Component y define una interface de acuerdo con la interface de Component.

DecoratorConcreto: adiciona la responsabilidad al Component.

Decorator propaga los mensajes a su objeto Component. Opcionalmente puede realizar operaciones antes y después de enviar el mensaje.

Ejemplo

Imaginemos que vendemos automóviles y el cliente puede opcionalmente adicionar ciertos componentes (aire acondicionado, mp3 player, etc). Por cada componente que se adiciona, el precio varía.


«java interface»
Vendible
+ String getDescripcion()
+ int getPrecio()
vendible
Decorador
AutoDecorator
- Vendible vendible
+ Vendible getVendible()
+ void setVendible(Vendible vendible)
Componente Concreto
Auto
FordFiesta
+ String getDescripcion()
+ int getPrecio()
FiatUno
+ String getDescripcion()
+ int getPrecio()
Decoradores Concretos
Mp3Player
+ Mp3Player(Vendible vendible)
+ String getDescripcion()
+ int getPrecio()
CdPlayer
+ CdPlayer(Vendible vendible)
+ String getDescripcion()
+ int getPrecio()
Gasoil
+ Gasoil(Vendible vendible)
+ String getDescripcion()
+ int getPrecio()
AireAcondicionado
+ AireAcondicionado(Vendible vendible)
+ String getDescripcion()
+ int getPrecio()

```java
public interface Vendible
{
    public String getDescripcion();
    public int getPrecio();
}
```

```java
public abstract class Auto implements Vendible
{
    //aca irían todos los atributos propios
    //de un auto, pero no nos interesa para el ejemplo
}
```

```java
public abstract class Auto implements Vendible
{
    //aca irían todos los atributos propios
    //de un auto, pero no nos interesa para el ejemplo
}
```

```java
public class FordFiesta extends Auto
{
    @Override
    public String getDescripcion()
    {
        return "Ford Fiesta modelo 2008";
    }

    @Override
    public int getPrecio()
    {
        return 25000;
    }
}
```

```java
public class FiatUno extends Auto
{
    @Override
    public String getDescripcion()
    {
        return "Fiat Uno modelo 2006";
    }

    @Override
    public int getPrecio()
    {
        return 1500;
    }
}
```

```
public abstract class AutoDecorator implements Vendible
{
    private Vendible vendible;

    public Vendible getVendible()
    {
        return vendible;
    }

    public void setVendible(Vendible vendible)
    {
        this.vendible = vendible;
    }
}
```

```java
public class Mp3Player extends AutoDecorator
{
    public Mp3Player(Vendible vendible)
    {
        super.setVendible(vendible);
    }

    @Override
    public String getDescripcion()
    {
        return getVendible().getDescripcion() + " +MP3 Player";
    }

    @Override
    public int getPrecio()
    {
        return getVendible().getPrecio() + 250;
    }
}
```

```java
public class CdPlayer extends AutoDecorator
{
    public CdPlayer(Vendible vendible)
    {
        super.setVendible(vendible);
    }

    @Override
    public String getDescripcion()
    {
        // TODO Implement this method
        return getVendible().getDescripcion() + " + CD player";
    }

    @Override
    public int getPrecio()
    {
        // TODO Implement this method
        return getVendible().getPrecio() + 100;
    }
}
```

```java
public class Gasoil extends AutoDecorator
{
    public Gasoil(Vendible vendible)
    {
        super.setVendible(vendible);
    }

    @Override
    public String getDescripcion()
    {
        return getVendible().getDescripcion()+" + Gasoil";
    }

    @Override
    public int getPrecio()
    {
        return getVendible().getPrecio()+1200;
    }
}
```

```java
public class AireAcondicionado extends AutoDecorator
{
    public AireAcondicionado(Vendible vendible)
    {
        super.setVendible(vendible);
    }

    @Override
    public String getDescripcion()
    {
        return getVendible().getDescripcion() + " + Aire acondicionado";
    }

    @Override
    public int getPrecio()
    {
        return getVendible().getPrecio() + 1500;
    }
}
```

```java
public class Cliente
{
    public static void main(String[] args)
    {
        Vendible auto = new FiatUno();
        System.out.println(auto.getDescripcion());
        System.out.println("Su precio es:" + auto.getPrecio());
        auto = new CdPlayer(auto);
        System.out.println(auto.getDescripcion());
        System.out.println("Su precio es:" + auto.getPrecio());
        auto = new Gasoil(auto);
        System.out.println(auto.getDescripcion());
        System.out.println("Su precio es:" + auto.getPrecio());

        Vendible auto2 = new FordFiesta();
        auto2 = new Mp3Player(auto2);
        auto2 = new Gasoil(auto2);
        auto2 = new AireAcondicionado(auto2);
        System.out.println(auto2.getDescripcion());
        System.out.println("Su precio es:" + auto2.getPrecio());
    }
}
```

```
public class Cliente
{
    public static void main(String[] args)
    {
        Vendible auto = new FiatUno();
        System.out.println(auto.getDescripcion());
        System.out.println("Su precio es:" + auto.getPrecio());
        auto = new CdPlayer(auto);
        System.out.println(auto.getDescripcion());
        System.out.println("Su precio es:" + auto.getPrecio());
        auto = new Gasoil(auto);
        System.out.println(auto.getDe
        System.out.println("Su precio

        Vendible auto2 = new FordFies
        auto2 = new Mp3Player(auto2);
        auto2 = new Gasoil(auto2);
        auto2 = new AireAcondicionado
        System.out.println(auto2.getD
        System.out.println("Su precio
    }
}
```

```
Fiat Uno modelo 2006
Su precio es:1500
Fiat Uno modelo 2006 + CD player
Su precio es:1600
Fiat Uno modelo 2006 + CD player + Gasoil
Su precio es:2800
Ford Fiesta modelo 2008 +MP3 Player + Gasoil +
Aire acondicionado
Su precio es:27950
```

Consecuencias

- Es más flexible que la herencia: utilizando diferentes combinaciones de unos pocos tipos distintos de objetos decorator, se puede crear muchas combinaciones distintas de comportamientos. Para crear esos diferentes tipos de comportamiento con la herencia se requiere que definas muchas clases distintas.
- Evita que las clases altas de la jerarquía estén demasiado cargadas de funcionalidad.
- Un componente y su decorador no son el mismo objeto.
- Provoca la creación de muchos objetos pequeños encadenados, lo que puede llegar a complicar la depuración.
- La flexibilidad de los objetos decorator los hace más propenso a errores que la herencia. Por ejemplo, es posible combinar objetos decorator de diferentes formas que no funcionen, o crear referencias circulares entre los objetos decorator.