Diseño de Software

Patrones de Diseño

Patron 3

DECORADOR

Autor
Ekaitz Arriola Garcia

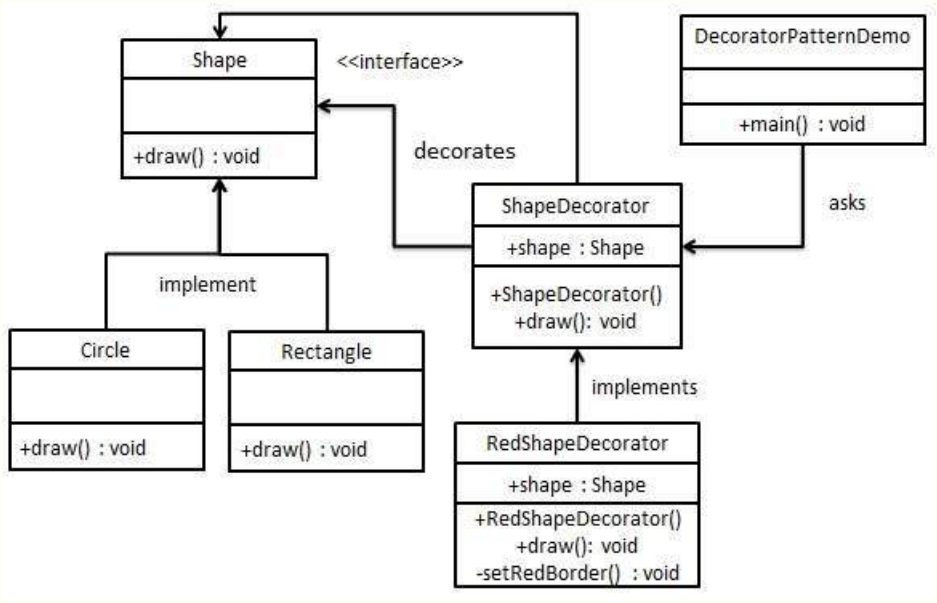
Profesor
Miguel Angel Mesas Uzal

March 28, 2021

Índice

Descripción general de la solución - Patron decorator

(a) Contexto, restricciones, alcance

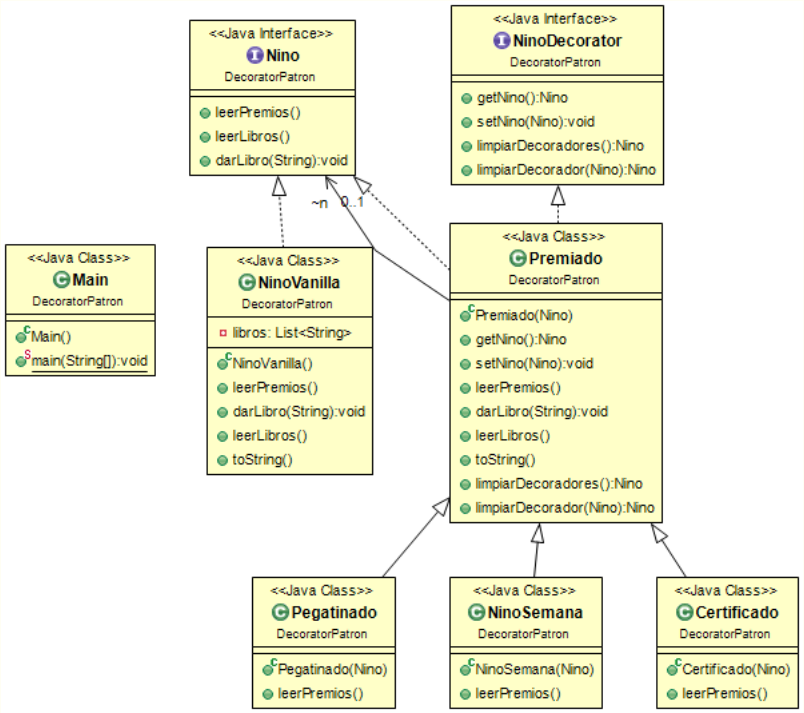


Este patron permite agregar funcionalidades dinámicamente a los objetos, así como alterar las relaciones entre clases en tiempo de ejecución. Su versión mal seria crear muchas clases que hereden de la misma, incorporando una nueva funcionalidad. En vez de eso, este patrón nos permite crear clases que implementan estas funcionalidades, y vincularla a la primera.

Usar este patrón es aprovechar las relaciones de herencia para poder tratar a un objeto base igual que el decorado. Esto es, no limita el uso de la herencia.
Es más, es más flexible que la herencia. Al usarlo, se pueden alterar las responsabilidades en tiempo de ejecución, aparte de salvar del uso de la herencia en algunos casos.

Componentes del patrón utilizado

(b) Descripción UML - Uso de composición/agregación



Este ejemplo minimalista del patrón decorator esta compuesto por dos interfaces, una clase que hereda de uno de ellos y otra abstracta que hereda de la otra, que a su vez otras heredan de esta última. Por último, esta tambien la clase Main para probar el programa.

La interfaz Nino es la que define como deben de ser los elementos que se manipularan en el programa. Es la que cumple la función de relacionar la clase NinoVanilla, que es la base, y los decoradores. La otra interfaz impone como debe de ser un decorador. Esta última la implementa la clase abstracta Premiado. Apartir de esta heredan los decoradores concretos, los cuales alteran el componente leerLibro.

(c) Uso de Interface/clase abstracta

Se usan dos interfaces. La primera es para definir que debe de implementar lo que lo heredan. Este es el que relaciona la base y los decoradores. La otra es la que define como debe modelarse un decorador.
Tambien se usa una clase abstracta para el decorador, para concretar unas cosas y posteriormente crear diferentes decoradores a partir de esta.

Problema continua en la siguiente página...

(d) Motivo del diseño utilizado

El principal motivo para usar este diseño es evitar crear muchas clases que hereden de la misma, incorporando una nueva funcionalidad, al crear diferentes clases que las implementan y se relacionan.

Resultado de la ejecución mediante una clase Test

(e) Descripción del test realizado

En el main se ejecutara una simulación de un año de guardería. En el se preguntara si se quiere ver los libros de algún niño y después se preguntara de quien. El niño al enseñarte los libros también enseñan sus premios. De estos premios, se irán acumulando menos el de NinoSemana, que solo podrá tenerlo un niño a la vez. El certificado también se acumula, pero si hay una vez que no se consigue, se eliminan todos los acumulados hasta ahora. Las pegatinas no se quitan.
Los premios se dirán en orden descendente, para que se pueda ver fácilmente los últimos en aplicarse.

(f) Resultado de la ejecución

Resultado de la ejecución

```
Cuantos ninios? (se va a colar uno extra)
...
Quieres preguntar a algun niño por sus libros? (y/n)
A cual? [1-4] | n: 0
Pegatinado
Certificado
Pegatinado
Pegatinado
NinoSemana
[ Libro123,Libro78 ]
...
```

El resultado del test, al ser una solución muy minimalista, los métodos solo operan con Strings.

Conclusiones

(g) Adaptación al cambio

Cumpliendo con su propósito, al aplicar este patrón se puedes añadir dinámicamente funcionalidades o enriquecer los componentes de un objeto. Si hiciese falta cambiar algo relacionado con una funcionalidad o componente, solo se tendría que modificar el decorador. Y por supuesto, en el caso de tener que añadir uno nuevo, mientras se corresponda con lo establecido por la interfaz, solo se tendría que crear el decorador, sin alterar ninguna otra parte del código.

(h) Posibles mejoras - Otras soluciones

Una posible mejora podría ser aplicar el patron singleton a NinoSemana, ya que solo se aplica una vez, y solo a una persona.

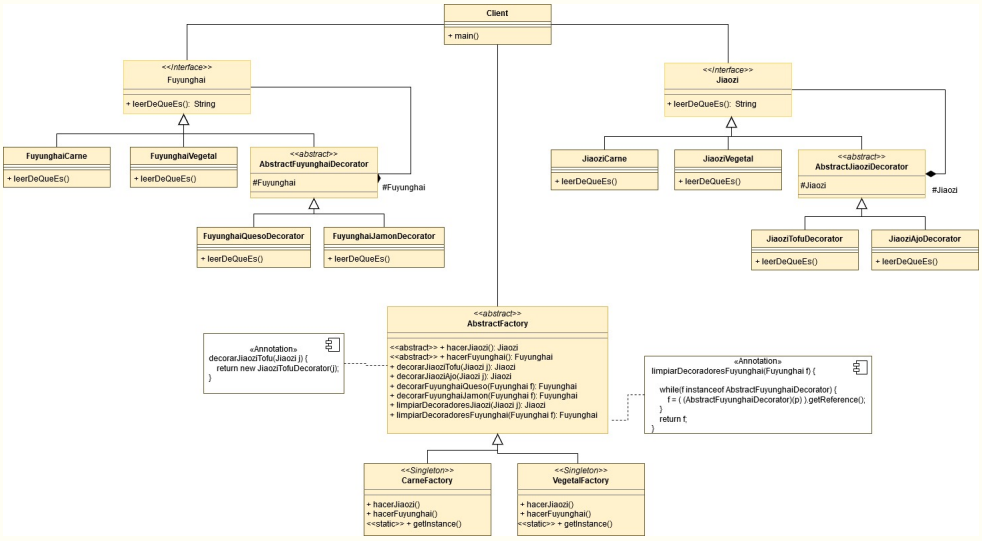
```
public class NinoSemanaSingleton extends Premiado {

    private static Nino instance = null;

    public static Nino setNinoSemana(Nino n) {
        if(instance==null || !(instance instanceof NinoSemanaSingleton))
            instance = new NinoSemanaSingleton(n);
        ((NinoDecorator)instance).setNino(n);
        return instance;
    }

    public static Nino getNinoSemana() {
        return instance;
    }

    private NinoSemanaSingleton(Nino n) {
        super(n);
    }

    @Override
    public String leerPremios() {
        return super.leerPremios()+"El niño de la semana";
    }

    @Override
    public void darLibro(String libro) {
        super.darLibro(libro);
    }
}
```

Para eliminar un decorador (aunque tener que eliminar un decorador...) se me ocurre añadirle o bien un identificador a cada decorador concreto, o una implementación de un método por cada uno, para poder eliminar más fácilmente ese decorador en concreto. Una variante de esta ultima es hacer eso pero por diferentes tipos de decoradores abstractos de los que heredan otros decoradores concretos, así pudiéndose eliminar todos los de un tipo a la vez. Incluso se podría hacer que solo pudiese tener un decorador de cada tipo.

Aún así, la solución por la que me decante fue tratar a los decoradores como lo que son, una lista enlazada. Para retirar uno bastaría con cambiar la referencia a un objeto por la referencia a un objeto de ese mismo objeto, saltándose así un elemento, esto es, retirandolo de la lista.

Problema continua en la siguiente página...

El patrón en si se da mucho a aplicarse con abstract factory. El ejemplo de a continuación no guarda relación con el ejercicio. Es un ejemplo de la composición de ambos patrones.



La imagen se ve un poco... pero si se le hace zoom se ve bien.

Bibliografía

[1] Código y UML
[2] Tutorials Point - decorator
[3] Y claro está, las clases