POLITÉCNICA

"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Patrones de Diseño

12. *Decorator*

Luis Fernández Muñoz

https://www.linkedin.com/in/luisfernandezmunyoz

setillofm@gmail.com

INDICE

1. Problema

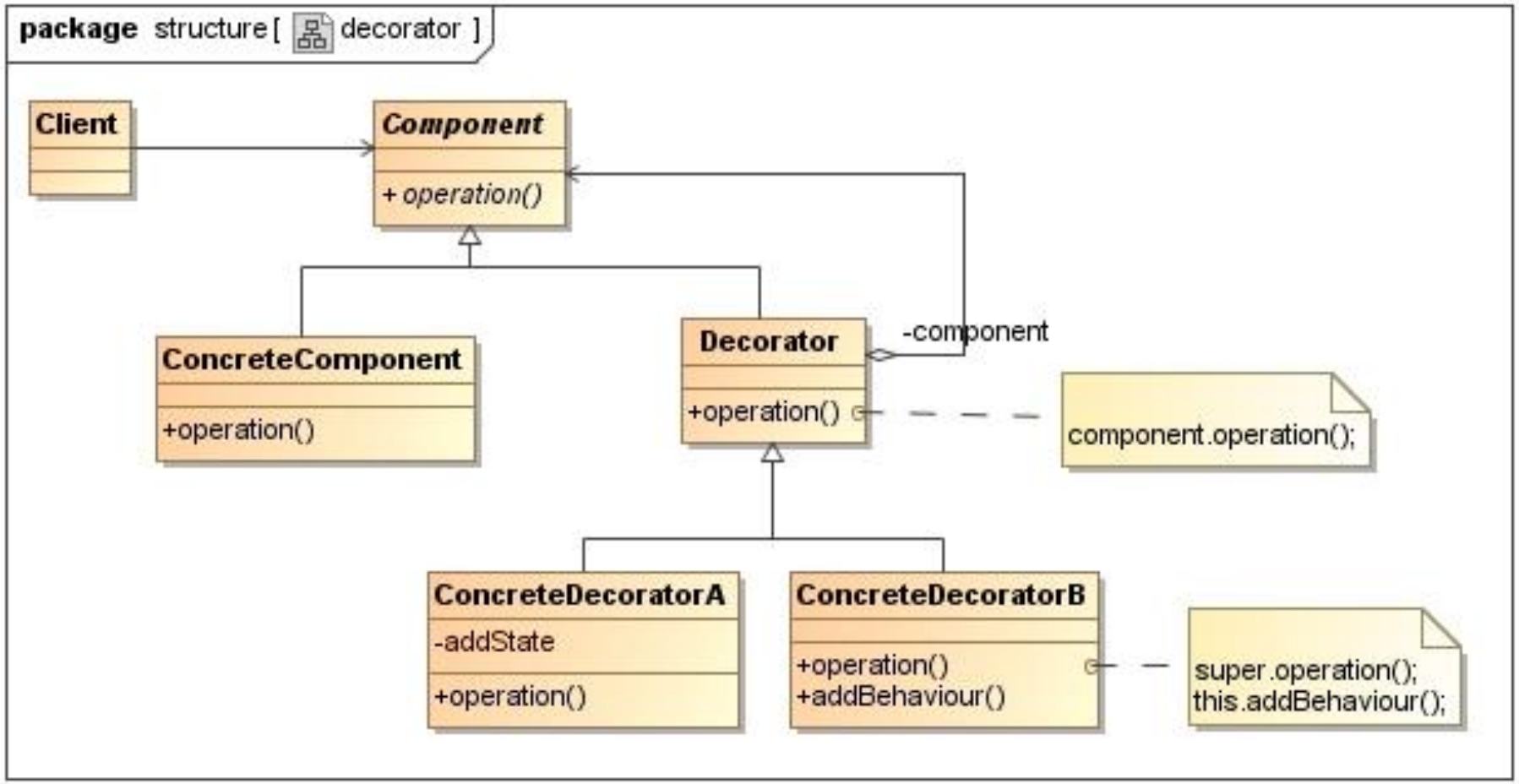
- *Un aprendiz de un oficio puede llevar a cabo ciertas tareas pero sobre aspectos muy básicos seleccionadas/encargadas por el gran artesano. Puntualmente, para alguna tarea más delicada, el artesano explica la tarea a un oficial para que supervise según la está realizando el aprendiz sobre aspectos más avanzados.*
- *De esta manera, se consigue añadir nuevas responsabilidades al aprendiz (nuevos aspectos más avanzados) con la intermediación/supervisión del oficial.*

- *Agrega responsabilidades adicionales a un objeto dinámicamente, proveyendo una alternativa flexible a la subclasificación extendiendo la funcionalidad*

1. Problema

- Úsese cuando:
  - A veces queremos añadir responsabilidades a objetos individuales en vez de a toda una clase, sin afectar a otros objetos
  - Para responsabilidad que pueden ser retiradas
  - Cuando la extensión mediante la herencia no es viable. A veces es posible tener un gran número de extensiones independientes, produciéndose una explosión de subclases para permitir todas las combinaciones. O puede ser que una definición de una clase esté oculta o no que no esté disponible para ser heredada.

2. Solución


package structure [decorator]
Client
Component
+ operation()
ConcreteComponent
+operation()
Decorator
-component
+operation()
component.operation();
ConcreteDecoratorA
-addState
+operation()
ConcreteDecoratorB
+operation()
+addBehaviour()
super.operation();
this.addBehaviour();

2. Solución

- Un enfoque más flexible es encerrar el componente en otro objeto. Al objeto confinante se le denomina decorador. El decorador se ajusta a la interfaz del componente que decora de manera que su presencia es transparente a sus clientes. El decorador reenvía las peticiones al componente y puede realizar acciones adicionales antes o después del reenvío. Dicha transparencia permite anidar decoradores recursivamente, permitiendo así un número de responsabilidades añadidas.
  - Las subclases de Decorador son libres de añadir operaciones para determinadas funcionalidades. Lo importante de este patrón es que permite que los decoradores aparezcan en cualquier lugar en el que pueda ir un componente.

2. Solución

- La interfaz de un objeto decorador debe ajustarse a la interfaz del componente que decora. Las clases DecoradorConcreto deben por tanto heredar de una clase común.
- No hay necesidad de definir una clase abstracta Decorador cuando sólo necesitamos añadir una responsabilidad. Eso es lo que suele ocurrir cuando estamos tratando con una jerarquía de clases existente y no diseñando una nueva. En ese caso, podemos obtener la responsabilidad del Decorador reenviando peticiones al componente en el DecoradorConcreto

2. Solución

- Para garantizar una interfaz compatible, los componentes y los decoradores deben descender de una clase Componente común. Es importante que esta clase común se mantenga ligera; es decir, debería centrarse en definir una interfaz, no en guardar datos. La definición de cómo se representan los datos debería delegarse a las subclases; de no ser así, la complejidad de la clase Componente puede hacer que los decoradores sean demasiado pesados como para usar un gran número de ellos. Poner mucha funcionalidad en el Componente también incrementa la probabilidad de que las subclases concretas estén pagando por características que no necesitan

2. Solución

- Podemos pensar en un decorador como un revestimiento de un objeto que cambia su comportamiento. Una alternativa es cambiar las interioridades del objeto. El patrón *Strategy* es un buen ejemplo de patrón para cambiar las tripas. Las estrategias son una mejor elección en aquellas situaciones en las que la clase Componente es intrínsecamente pesada, lo que hace que el patrón *Decorator* sea demasiado costoso de aplicar. En el patrón *Strategy*, el componente delega parte de su responsabilidad en un objeto estrategia aparte. El patrón *Strategy* nos permite alterar o extender la funcionalidad del componente cambiando el objeto estrategia.
  - Puesto que el patrón *Decorator* sólo cambia la parte exterior de una objeto, el componente no tiene que saber nada de sus decoradores;
  - Con estrategias, el componente conoce sus posibles extensiones.

3. Consecuncias

- El patrón *Decorator* proporciona una manera más flexible de añadir responsabilidades a los objetos que la que podía obtenerse a través de la herencia (múltiple) estática. Con los decoradores se pueden añadir y eliminar responsabilidades en tiempo de ejecución simplemente poniéndolas y quitándolas. Por el contrario, la herencia requiere crear una nueva clase para cada responsabilidad adicional. Esto da lugar a muchas clases diferentes e incrementa la complejidad de un sistema. Por otro lado, proporcionar diferentes clases *Decorador* para una determinada clase Componente permite mezclar responsabilidades

3. Consecuncias

- Ofrece un enfoque para añadir responsabilidades que consiste en pagar sólo por aquello que se necesita. En vez de tratar de permitir todas las funcionalidades inimaginables en una clase compleja y adaptable, podemos definir primero una clase simple y añadir luego funcionalidad incrementalmente con objetos Decorador. La funcionalidad puede obtenerse componiendo partes simples. Como resultado, una aplicación no necesita pagar por características que no usa. También resulta fácil definir nuevos tipos de Decoradores independientemente de las clases de objetos de las que hereden, incluso para extensiones que no hubieran sido previstas. Extender una clase compleja tiende a exponer detalles no relacionados con las responsabilidades que estamos añadiendo

3. Consecuncias

- Un decorador se comporta como un revestimiento transparente. Pero desde el punto de vista de la identidad de un objeto, un componente decorado no es idéntico al componente en sí. Por tanto, no deberíamos apoyarnos en la identidad de objetos cuando estamos usando decoradores
- Un diseño que usa el patrón *Decorator* suele dar como resultado sistemas formados por muchos objetos pequeños muy parecidos. Los objetos sólo se diferencian en la forma en que están interconectados, y no en su clase o en el valor de sus variables. Aunque dichos sistemas son fáciles de adaptar por parte de quienes los comprenden bien, pueden ser difíciles de aprender y de depurar

4. Relaciones

- ***Composite*** para estructurar los decoradores. Cuando se usan juntos *Decorator* y ***Composite***, normalmente ambos tendrán una clase padre común. Por tanto, los *Decorator* tendrán que admitir la interfaz *Component* con operaciones como *add*, *remove* y *getChild*.
- ***Prototipo*** para crear los decoradores

5. Comparativa

- ***Decorator** vs **Adapter.***
  - Ver *Adapter* vs *Decorator*
- ***Decorator** vs **Composite.***
  - Ver *Composite* vs *Decorator*
- ***Decorator** vs **Strategy***
  - Son dos formas alternativas de modificar un objeto.
    - *Decorator* permite cambiar el exterior de un objeto;
    - *Strategy* permite cambiar sus tripas.

5. Comparativa

- ***Decorator** vs **Proxy.***
  - Tienen una estrucutra similar porque proporcionan un nivel de indirección a un objeto y sus implementaciones mantienen una referencia a otro objeto, al cual reenvían peticiones. Pero están pensados para propósitos diferentes.
    - *Decorator*, el componente proporciona sólo parte de funcionalidad, y uno o más decoradores hacen el resto. *Decorator* se encarga de aquellas situaciones en la que no se puede determinar toda la funcionalidad de un objeto en tiempo de compilación, o al menos no resulta conveniente hacerlo.
    - *Proxy* no tiene que ver con asignar o quitar propiedades dinámicamente, y tampoco está diseñado para la composición recursiva. Su propósito es proporcionar un sustituto para un sujeto cuando no es conveniente o deseable acceder directamente a él. El sujeto define la principal funcionalidad y *Proxy* proporciona (o rechaza) el acceso al mismo. Se centra en una relación entre el representante y su sujeto que puede expresarse estáticamente.