POLITÉCNICA

"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Patrones de Diseño

11. *Composite*

Luis Fernández Muñoz

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

setillofm@gmail.com

INDICE

1. Problema

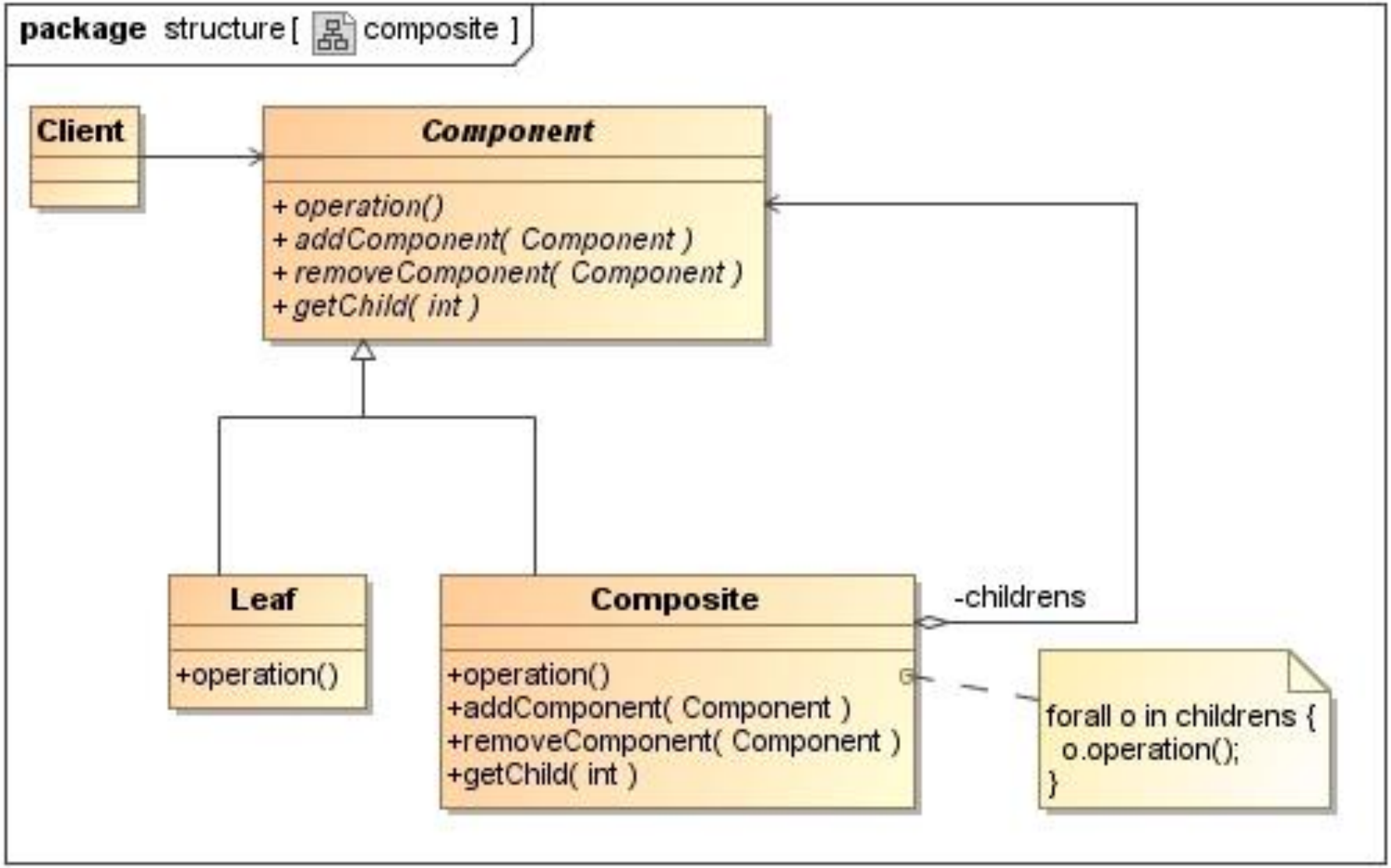
- *En muchos juegos de mesa (p.e. trivial pursuit, parchis, …), hay limitaciones en el tablero para el número máximo de jugadores. Cuando hay más personas que este máximo existe la posibilidad de formar equipo para que se entretengan todos.*
- *Pero si no se buscan las fórmulas adecuadas puede ser un caos cuando el turno de un "jugador/equipo" lo realizan varias personas del equipo a la vez.*
- *La solución más habitual es nombrar un representante del equipo y, de cara al resto de jugadores, es con el único que interactúa. Internamente el representante se organizará transparentemente con el resto de personas del equipo para tomar decisiones.*

- ***Compone objetos en estructuras arbóreas para representar las jerarquía de todo-parte y permite a los clientes tratar con objetos individuales y objetos compuestos uniformemente***

1. Problema

- Use el patrón *Composite* cuando:
  - Quiera representar jerarquías de objetos parte-todo
  - Quiera que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales. Los clientes tratarán a todos los objetos de la estructura compuesta de manera uniforme

2. Solución


package structure [composite]
Client
Component
+ operation()
+ addComponent(Component)
+ removeComponent(Component)
+ getChild(int)
Leaf
+operation()
Composite
+operation()
+addComponent(Component)
+removeComponent(Component)
+getChild(int)
-childrens
forall o in childrens {
o.operation();
}

2. Solución

- Mantener referencias de los componentes hijos a sus padre puede simplificar el recorrido y la gestión de una estructura compuesta. La referencia al padre facilita ascender por la estructura y borrar un componente.
  - El lugar habitual donde definir la referencia al padre es en la clase Componente. Las clases Hoja y Compuesto pueden heredar la referencia y las operaciones que la gestionan.
  - Con referencias al padre, es esencial mantener el invariante de que todos los hijos de un compuesto tienen como padre al compuesto que a su vez los tiene a ellos como hijos. El modo más fácil de garantizar esto es cambiar el padre de un componente sólo cuando se añade o se elimina a éste de un compuesto. Si se puede implementar una vez en las operaciones Añadir y Eliminar de la clase Compuesto entonces puede ser heredado por todas las subclases.

2. Solución

- Muchas veces es útil compartir componentes, por ejemplo para reducir los requisitos de almacenamiento. Pero cuando un componente no puede tener más de un padre, compartir componentes se hace más fácil
  - Una posible solución es que los hijos almacenen múltiples padres. Pero eso puede llevarnos a ambigüedades cuando se propaga una petición de arriba en la estructura.
- Uno de los objetivos del patrón *Composite* es hacer que los clientes se despreocupen de las clases Hoja o Compuesto que están usando. Para conseguirlo, la clase Componente debería definir tantas operaciones comunes a las clase Compuesto y Hoja como sea posible. Las clase Componente normalmente proporciona implementaciones predeterminadas para estas ocasiones, que serán redefinidas por las subclases Hoja y Compuesto

2. Solución

- No obstante, este objetivo a veces entra en conflicto con el principio de diseño de jerarquías de clases que dice que una clase debería definir operaciones que tienen sentido en sus subclases. Hay muchas operaciones permitidas por Componente que no parecen tener sentido en las clase Hoja.
- A veces un poco de creatividad muestra cómo una operación que podría parecer que sólo tiene sentido en el caso de los Compuestos puede implementarse para todos los Componentes, moviéndola a la clase Componente. Por ejemplo, la interfaz para acceder a los hijos es una parte fundamental de la clase Compuesto pero no de las clase Hoja. Pero si vemos a una Hoja como un Componente para acceder a los hijos que nunca devuelve ningún hijo. Las clases Hoja pueden usar esa implementación predeterminada, pero las clases Compuesto la re-implementarán para que devuelva sus hijos

2. Solución

- Aunque la clase Compuesto implementa las operaciones Añadir y Eliminar para controlar los hijos, un aspecto importante del patrón Compuesto es qué clases declaran estas operaciones en la jerarquía de clases Compuesto. La decisión implica un equilibrio entre seguridad y transparencia:
  - Definir la interfaz de gestión de los hijos en la raíz de la jerarquía de clases nos da transparencia, puesto que podemos tratar a todos los componentes de manera uniforme. Sin embargo, sacrifica la seguridad, ya que los clientes pueden intentar hacer cosas sin sentido, como añadir y eliminar objetos de las hojas
  - Definir la gestión de los hijos en la clase Compuesto nos proporciona seguridad, ya que cualquier intento de añadir o eliminar objetos de las hojas será detectado en tiempo de compilación en un lenguaje estáticamente tipado. Pero perdemos transparencia porque las hojas y los compuestos tienen interfaces diferentes

2. Solución

- Podríamos estar tentados de definir el conjunto de hijos como una variable de instancia de la clase Componente en la que se declaren las operaciones de acceso y gestión de hijos. Pero poner el puntero al hijo en la clase base incurre en una penalización de espacio para cada hoja, incluso aunque una hoja nunca tenga hijos. Esto sólo merece la pena si hay relativamente pocos hijos en la estructura.
- Muchos diseños especifican una ordenación de los hijos de Compuesto. Cuando la ordenación de los hijos es una cuestión a tener en cuenta, debemos diseñar las interfaces de acceso y gestión de hijos cuidadosamente para controlar la secuencia

3. Consecuencias

- Define jerarquías de clases formadas por objetos primitivos y compuestos. Los objetos primitivos pueden componerse en otros objetos más complejos, que a su vez pueden ser compuestos y así de manera recurrente. Allí donde el código espere un objeto primitivo, también podrá recibir un objeto compuesto.
- Los clientes pueden tratar uniformemente a las estructuras compuestas y a los objetos individuales. Los clientes normalmente no conocen (y no les debería importar) si están tratando con una hoja o con un componente compuesto. Esto simplifica el código del cliente, puesto que evita tener que escribir funciones con instrucciones *if* anidadas en las clases que definen la composición

3. Consecuencias

- Facilita añadir nuevos tipos de componentes. Si se definen nuevas subclases Compuesto u Hoja, éstas funcionarán automáticamente con las estructuras y el código cliente existentes. No hay que cambiar los clientes para las nuevas clases Componentes
- Puede hacer que un diseño sea demasiado general. Las desventaja de facilitar añadir nuevos componentes es que hace más difícil restringir los componentes de un compuesto. A veces queremos que un compuesto sólo tenga ciertos componentes. Con el patrón *Composite*, no podemos confiar en el sistema de tipos para que haga cumplir estas restricciones por nosotros. En vez de eso, tendremos que usar comprobaciones en tiempo de ejecución

4. Relaciones

- ***Iterator*** para recorrer las estructuras recursivas definidas por el patrón *Composite*.
- ***Prototype*** para crear los componentes
- ***Flyweight*** para implementar una estructura lógica jerárquica en términos de una grafo dirigido acíclico con nodos hojas compartidos
- ***Visitor*** para realizar operaciones sobre los componentes de la composición.
- ***Chain of Responsability*** para que los componentes puedan acceder a las propiedades globales a través de su padre.
- ***Decorator*** para redefinir las propiedades sobre partes de la composición.

5. Comparativa

- ***Composite** vs **Decorator***
  - Tienen diagramas de estructura parecidos, lo que refleja el hecho de que ambos se basan en la composición recursiva para organizar un número indeterminado de objetos. Esto puede tentar a pensar en un objeto decorador como un compuesto degenerado, pero esto no representa las diferencias entre sus propósitos:
    - *Composite* consiste en estructurar subclases para que se puedan tratar de manera uniforme muchos objetos relacionados, y que múltiples objetos puedan ser tratados como uno solo. Es decir, no se centra en la decoración sino en la representación.
    - *Decorator* está diseñado para permitir responsabilidades a objetos sin crear subclases. Esto evita la explosión de subclases a la que puede dar lugar al intentar cubrir cada combinación de responsabilidades estáticamente.

5. Comparativa

- ***Composite** vs **Decorator*** (cont.)
  - Son propósitos distintos pero complementarios. Por tanto, pueden usarse conjuntamente.
    - Habrá una clase abstracta con algunas subclases que son compuestos, otra que son decoradores y otra que implementan los principales bloques de construcción del sistema.
    - En este caso, tanto compuesto como decoradores tendrán una interfaz común.
    - Para *Composite* , un decorador es una hoja.
    - Para *Decorator*, un compuesto es un componente compuesto.
  - Por supuesto, no tienen por qué ser usados juntos, y sus propósitos son bastante distintos.