POLITÉCNICA

"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Patrones de Diseño

27. *Visitor*

Luis Fernández Muñoz

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

setillofm@gmail.com

INDICE

1. Problema

- *Una posible organización dentro de un hospital sería que un mismo empleado fuera responsable de todas las tareas a realizar (p.e. transportar, curar heridas, diagnosticar, intervención quirurjica, …) con los órganos de un paciente (p.e. piernas, bazo, menisco, …). Esto complicaría muchísimo la responsabilidad de dichos empleados que saben de todo y tendiendo a crecer con nuevas técnicas en cualquier area.*
- *Una alternativa sencilla y sin tendencia desmesurada a crecer es que cada tarea la realicen personas especializadas diferentes (p.e. celador, enfermero, médico, cirujano, …) visitando en el momento oportuno los órganos del paciente.*

- *Representa una operación para ser realizadas sobre los elementos de una estructura de objetos y permite definir nuevas operaciones sin cambiar las clases de elementos sobre los que opera*

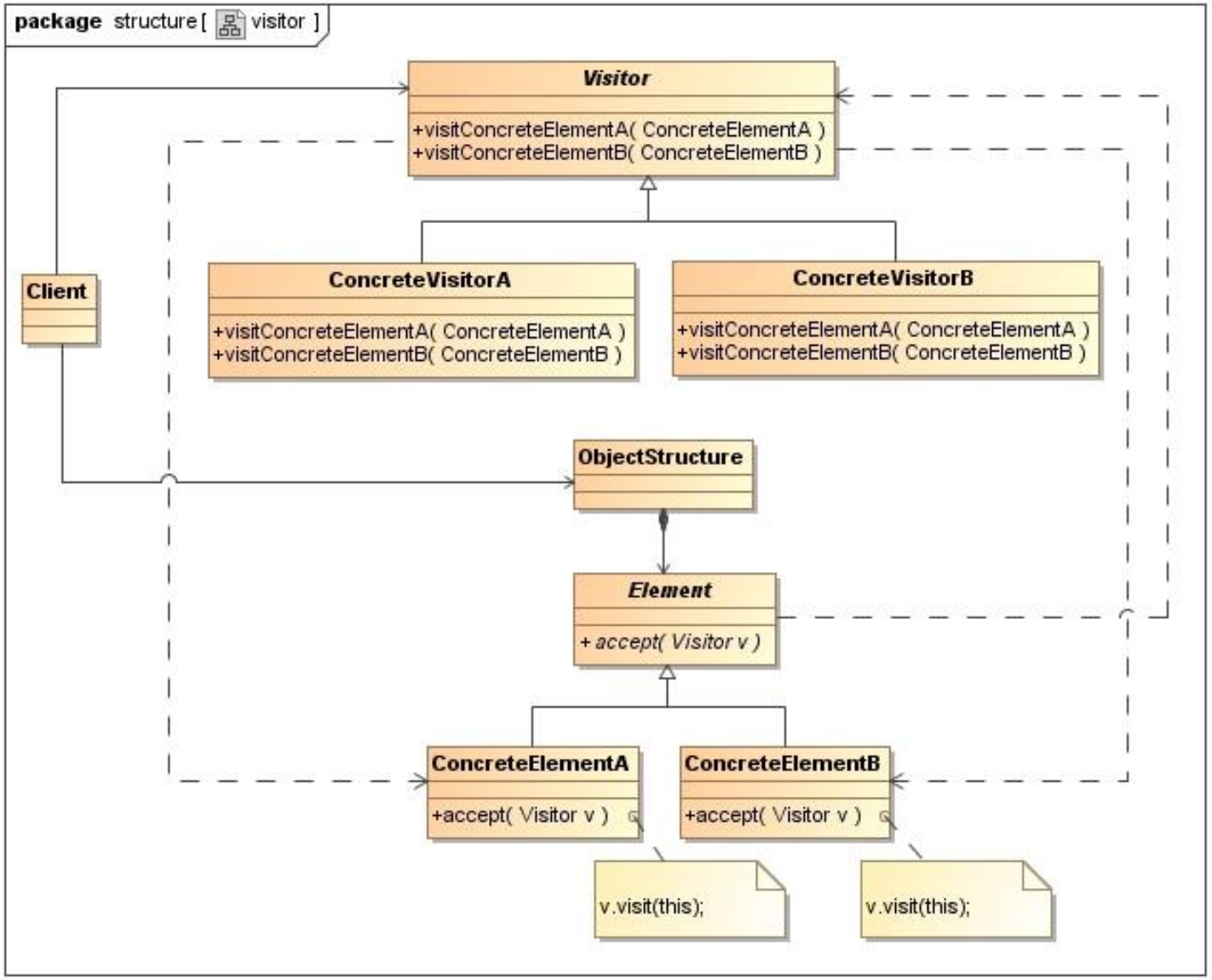
1. Problema

- Úsese cuando:
  - Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces y queremos realizar operaciones sobre esos elementos que dependen de la clase concreta
  - Se necesitan realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos y queremos evitar "contaminar" sus clases con dichas operaciones. El patrón *Visitor* permite mantener juntas operaciones relacionadas definiéndolas en una clase. Cuando la estructuras de objetos es compartida por varias aplicaciones, el patrón *Visitor* permite poner operaciones sólo en aquellas aplicaciones que las necesitan

1. Problema

- Las clase que definen la estructura de objetos rara vez cambian, pero muchas veces queremos definir nuevas operaciones sobre la estructura. Cambiar las clases de la estructura de objetos requiere redefinir la interfaz para todos los visitantes, lo que es potencialmente costoso. Si las clases de la estructura cambian con frecuencia, probablemente sea mejor definir las operaciones en las propias clases

2. Solución


package structure [visitor]
Visitor
+visitConcreteElementA(ConcreteElementA)
+visitConcreteElementB(ConcreteElementB)
Client
ConcreteVisitorA
+visitConcreteElementA(ConcreteElementA)
+visitConcreteElementB(ConcreteElementB)
ConcreteVisitorB
+visitConcreteElementA(ConcreteElementA)
+visitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
+ accept(Visitor v)
ConcreteElementA
+accept(Visitor v)
ConcreteElementB
+accept(Visitor v)
v.visit(this);
v.visit(this);

2. Solución

- Esto se logra mediante una técnica llamada doble-despacho. Simplemente significa que la operación que se ejecuta depende del tipo del solicitante y de los tipos de dos receptores.
  - Aceptar es una operación de doble-despacho. Su significado depende de dos tipos: el del Visitante y el del Elemento. El doble-despacho permite a los visitantes solicitar diferentes operaciones en cada clase de elemento.
  - Esta es la clave del patrón *Visitor*: la operación que se ejecuta depende tanto del tipo de Visitante como del tipo del Elemento visitado. En vez de enlazar operaciones estáticamente en la interfaz de Elemento, podemos fusionar las operaciones en un Visitante y usar aceptar para hacer el enlace en tiempo de ejecución. Extender la interfaz de Elemento consiste en definir una nueva subclase de Visitante en vez de muchas nuevas subclases de Elemento

2. Solución

- Un visitante debe visitar cada elemento de la estructura de objetos. La cuestión es ¿cómo lo logra? Podemos poner la responsabilidad del recorrido en cualquier de estos tres sitios: en la estructura de objetos, en el visitante o en un objeto iterador aparte.
  - Muchas veces es la estructura de objetos la responsable de la iteración. Un colección simplemente iterará sobre sus elementos, llamando a la operación Aceptar de cada uno. Un compuesto generalmente se recorrerá a sí mismo haciendo que cada operación Aceptar recorra los hijos del elemento y llame a Aceptar sobre cada uno de ello recursivamente

2. Solución

- Otra solución es usar un iterador para visitar los elementos. En C++, podríamos usar un iterador externo o interno, dependiendo de qué está disponible y qué es más eficiente. Puesto que los iteradores internos son implementados por la estructura de objetos, usar un iterador interno se parece mucho a hacer que sea la estructura de objetos la responsable de la iteración. La principal diferencia estriba en que un iterador interno no provocará un doble-despacho – llamará a una operación del visitante con un elemento como argumento, frente a llamar a una operación del elemento con el visitante como argumento -. Pero resulta sencillo usar el patrón Visitor con un iterador interno si la operación del visitante simplement llama a la operación del elemento sin recursividad

2. Solución

- Incluso se podría poner el algoritmo de recorrido en el visitante, si bien en ese caso acabaríamos duplicando el código del recorrido en cada VisitanteConcreto de cada agregado ElementoConcreto. La principal razón para poner la estrategia de recorrido en el visitante es implementar un recorrido especialmente complejo que dependa de los resultados de las operaciones de las estructura de objetos.

3. Consecuencias

- Los visitantes facilitan añadir nuevas operaciones que dependen de los componentes de objetos complejos. Podemos definir una nueva operación sobre una estructura simplemente añadiendo un nuevo visitante. Si, por el contrario, extendiésemos la funcionalidad sobre muchas clases, habría que cambiar cada clase para definir una nueva operación.
- El comportamiento similar no está desperdigado por las clases que definen la estructura de objetos; está localizado en un visitante. Las partes de comportamiento no relacionadas se dividen en sus propias subclases del visitante. Esto simplifica tanto las clases que definen los elementos como los algoritmos definidos por los visitantes. Cualquier estructura de datos específica de un algoritmo puede estar oculta en el visitante

3. Consecuencias

- El patrón *Visitor* hace que sea complicado añadir nuevas subclases de Elemento. Cada ElementoConcreto nuevo da lugar a una nueva operación abstracta del Visitante y a su correspondiente implementación en cada clase VisitanteConcreto. A veces se puede proporcionar en Visitante una implementación predeterminada que puede ser heredada por la mayoría de los visitantes concretos, pero esto representa una excepción más que una regla
  - Por tanto la cuestión fundamental a considerar a la hora de aplicar el patrón *Visitor* es si es más probable que cambie el algoritmo aplicado sobre una estructura de objetos o las clases de los objetos que componen la estructura. La jerarquía de clases Visitante puede ser difícil de mantener cuando se añaden nuevas clases de ElementConcreto con frecuencia.

3. Consecuencias

- En tales casos, es probablemente más fácil definir las operaciones en las clases que componen la estrucutura. Si la jerarquía de clases de Elemento es estable pero no estamos continuamente añadiendo operaciones o cambiando algoritmos, el patrón *Visitor* nos ayudará a controlar dichos cambios

- Un iterador puede visitar a los objetos de una estructura llamando a sus operaciones a medida que los recorre. Pero un iterador no puede trabajar en varias estructuras de objetos con distintos elementos. Esto implica que todos los elementos que el iterador puede visitar tienen una clase padre común Elemento.
  - El patrón *Visitor* no tiene esta restricción. Puede visitar objetos que no tienen una clase padre común. Se puede añadir cualquier tipo de objeto a la interfaz del Visitante.

3. Consecuencias

- Los visitantes pueden acumular estado a medida que van visitando cada elemento de la estructura de objetos. Sin un visitante, este estado se pasaría como argumentos extra a las operaciones que realizan el recorrido, o quizá como variables globales.
- El enfoque del patrón *Visitor* asume que la interfaz de ElementoConcreto es lo bastante potente como para que los visitantes hagan su trabajo. Como resultado, el patrón suele obligarnos a proporcionar operaciones públicas que acceden al estado interno de un elemento, lo que puede comprometer su encapsulación