Patrón de diseño Visitor

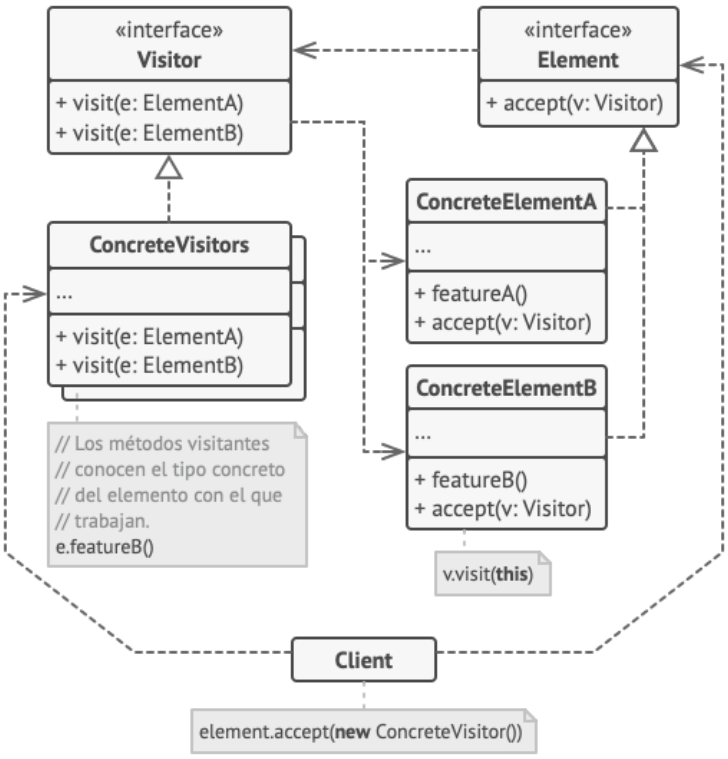
Patrón de diseño de comportamiento

Objetivo

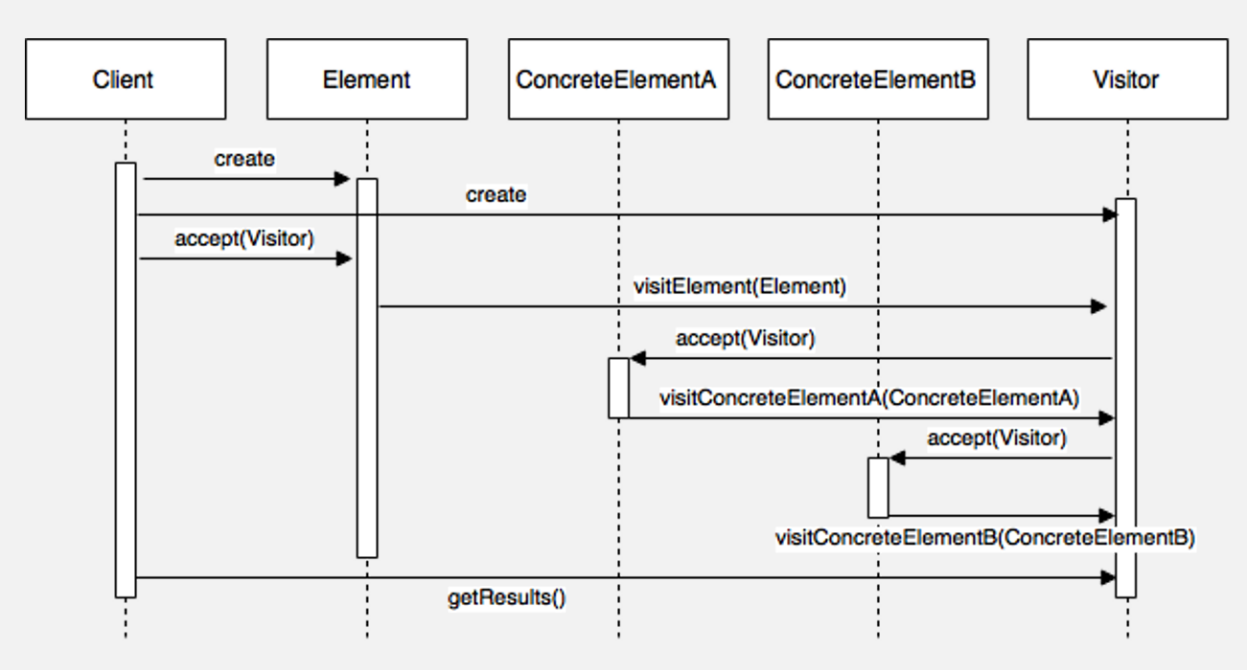
Permite separar algoritmos de los objetos sobre los que operan, es útil cuando necesitamos realizar operaciones distintas y no relacionadas sobre una estructura de objetos. Añade funcionalidades a una clase sin tener que modificarla.

Descripción

Este patrón establece que el objeto **Visitor** se define por separado y tiene el fin de implementar una operación que se realiza en uno o más elementos de la estructura del objeto. Los clientes que acceden a la estructura del objeto llaman entonces a la operación de envió *accept(visitor)* en el elemento en cuestión, lo que delega la petición en el objeto visitante aceptado. Así, el objeto **Visitor** puede realizar la operación necesaria.



- **Cliente**: Componente que interactúa con la estructura (element) y con el Visitante, éste es responsable de crear los visitantes y enviarlos al elemento para su procesamiento.
- **Element**: Representa la raíz de la estructura, en forma de árbol, sobre la que utilizaremos el Visitante. Este objeto por lo general es una interface que define el método accept y deberán implementar todos los objetos de la estructura.
- **ConcreteElement**: Representa un hijo de la estructura compuesta, la estructura completa puede estar compuesta de un gran número de estos objetos y cada uno deberá implementar el método accept.
- **Visitor**: Interface que define la estructura del visitante, la interface deberá tener un método por cada objeto que se requiera analizar de la estructura (element).
- **ConcreteVisitor**: Representa una implementación del visitante, esta implementación puede realizar una operación sobre el element. Es posible tener todos los ConcreteVisitor necesarios para realizar las operaciones que necesitemos.



Comó implementarlo

1. Declara la interfaz visitante con un grupo de métodos “visitantes”, uno por cada clase de elemento concreto existente en el programa.

2. Declara la interfaz de elemento. Si estás trabajando con una jerarquía de clases de elemento existente, añade el método abstracto de “aceptación” a la clase base de la jerarquía. Este método debe aceptar un objeto visitante como argumento.
3. Implementa los métodos de aceptación en todas las clases de elemento concreto. Estos métodos simplemente deben redirigir la llamada a un método visitante en el objeto visitante entrante que coincida con la clase del elemento actual.
4. Las clases de elemento sólo deben funcionar con visitantes a través de la interfaz visitante. Los visitantes, sin embargo, deben conocer todas las clases de elemento concreto, referenciadas como tipos de parámetro de los métodos de visita.
5. Por cada comportamiento que no pueda implementarse dentro de la jerarquía de elementos, crea una nueva clase concreta visitante e implementa todos los métodos visitantes.
   Puede que te encuentres una situación en la que el visitante necesite acceso a algunos miembros privados de la clase elemento. En este caso, puedes hacer estos campos o métodos públicos, violando la encapsulación del elemento, o anidar la clase visitante en la clase elemento. Esto último sólo es posible si tienes la suerte de trabajar con un lenguaje de programación que soporte clases anidadas.
6. El cliente debe crear objetos visitantes y pasarlos dentro de elementos a través de métodos de “aceptación”.

Aplicaciones

- Utiliza el patrón Visitor cuando necesites realizar una operación sobre todos los elementos de una compleja estructura de objetos.
  - El patrón Visitor te permite ejecutar una operación sobre un grupo de objetos con diferentes clases, haciendo que un objeto visitante implemente distintas variantes de la misma operación que correspondan a todas las clases objetivo.
- Utiliza el patrón Visitor para limpiar la lógica de negocio de comportamientos auxiliares.
  - El patrón te permite hacer que las clases primarias de tu aplicación estén más centradas en sus trabajos principales extrayendo el resto de los comportamientos y poniéndolos dentro de un grupo de clases visitantes.
- Utiliza el patrón cuando un comportamiento solo tenga sentido en algunas clases de una jerarquía de clases, pero no en otras.
  - Puedes extraer este comportamiento y ponerlo en una clase visitante separada e implementar únicamente aquellos métodos visitantes que acepten objetos de clases relevantes, dejando el resto vacíos.

Consecuencias

- Facilita la definición de nuevas operaciones
- Agrupa operaciones relacionadas
- Permite visitar objetos que no están relacionados por un padre común
- El visitor puede acumular el estado de una operación al visitar la estructura de objetos, en vez de pasarlo como argumento o usar variables globales

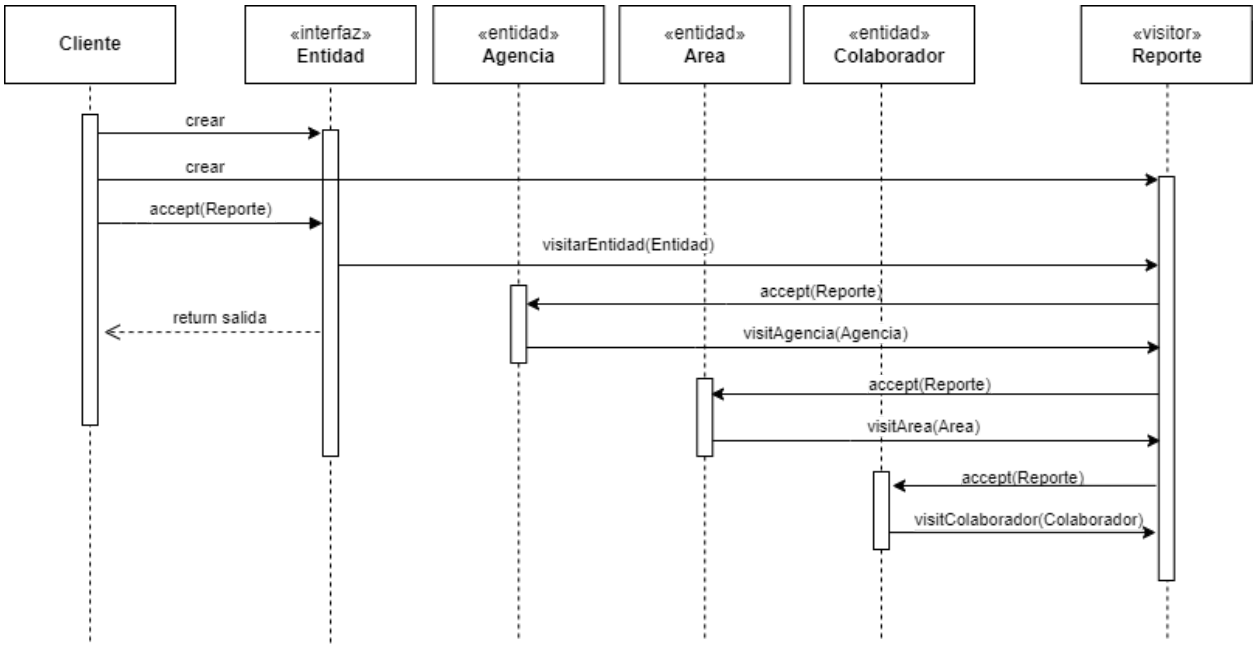
Desventajas

- Debes actualizar todos los visitantes cada vez que una clase se añada o elimine de la jerarquía de elementos.
- Los visitantes pueden carecer del acceso necesario a los campos y métodos privados de los elementos con los que se supone que deben trabajar.
- Añadir nuevas clases ConcreteElement es costoso
- Rompe la encapsulación

Relación con otros patrones

- Puedes tratar a **Visitor** como una versión potente del patrón **Command**. Sus objetos pueden ejecutar operaciones sobre varios objetos de distintas clases.
- Puedes utilizar el patrón **Visitor** para ejecutar una operación sobre un árbol **Composite** entero.
- Puedes utilizar **Visitor** junto con **Iterator** para recorrer una estructura de datos compleja y ejecutar alguna operación sobre sus elementos, incluso aunque todos tengan clases distintas.

Ejemplo


Cliente
«interfaz»
Entidad
«entidad»
Agencia
«entidad»
Area
«entidad»
Colaborador
«visitor»
Reporte
crear
crear
accept(Reporte)
visitarEntidad(Entidad)
accept(Reporte)
return salida
visitAgencia(Agencia)
accept(Reporte)
visitArea(Area)
accept(Reporte)
visitColaborador(Colaborador)