Il pattern Visitor

a cura di **Angelo Furfaro**
da “Design Patterns”, Gamma et al.

Dipartimento di Ingegneria Informatica Elettronica Modellistica e Sistemistica
Università della Calabria, 87036 Rende(CS) - Italy
Email: a.furfaro@dimes.unical.it
Web: http://angelo.furfaro.dimes.unical.it

Visitor

Classificazione

- Scopo: comportamentale
- Raggio d'azione: oggetti

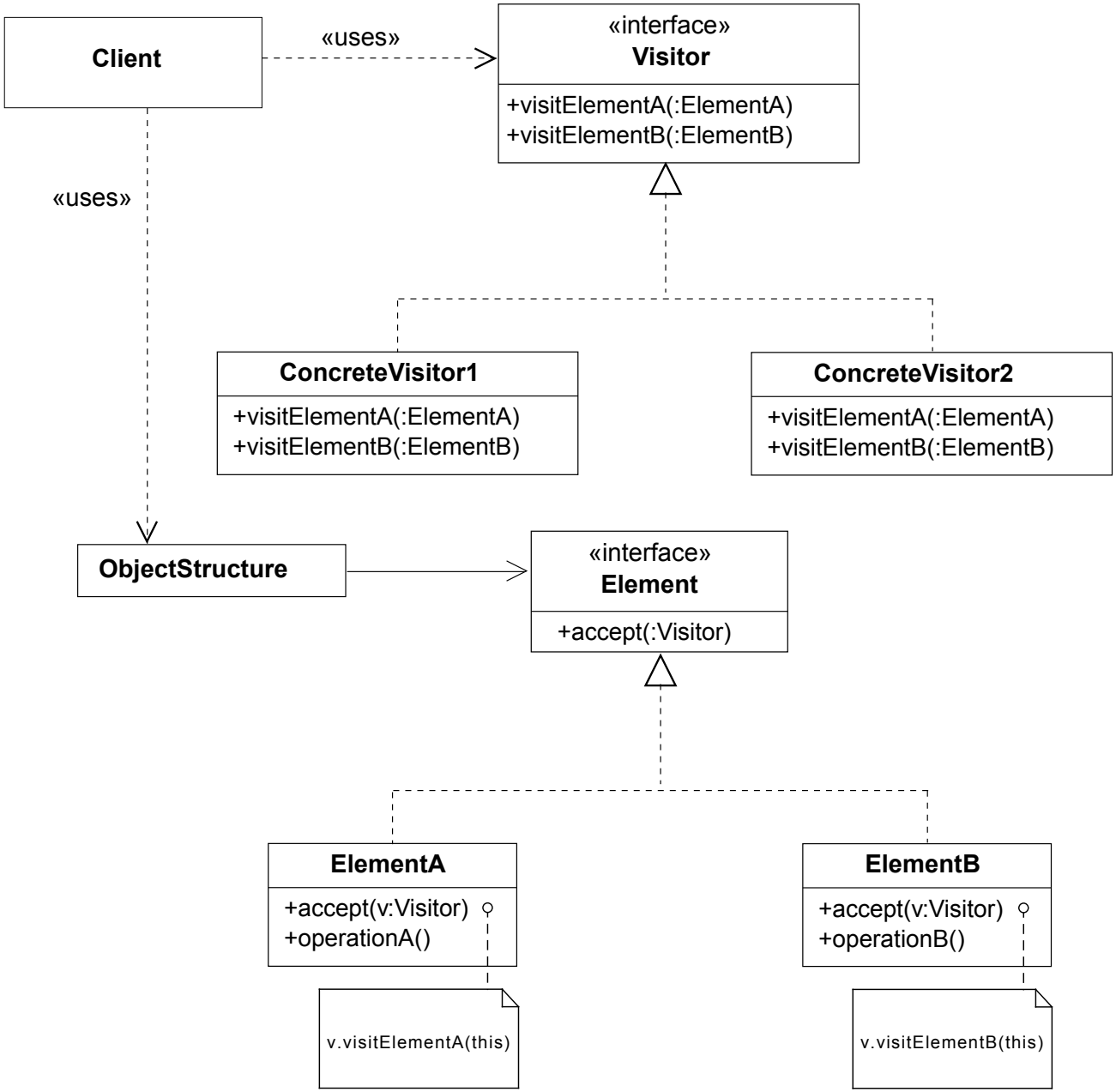
Scopo

- Rappresenta un'operazione da eseguire sugli oggetti di una struttura.
- Visitor consente di definire nuove operazioni senza modificare le classi degli oggetti su cui opera

Applicabilità

- Quando una struttura a oggetti contiene molti oggetti di classi con interfacce diverse e si ha la necessità di svolgere operazioni su questi oggetti dipendenti dalla loro classe concreta.
- Quando è necessario eseguire molte operazioni distinte e scorrelate sugli oggetti di una struttura ma si vuole evitare di modificare le interfacce delle loro classi con tali operazioni.
- Quando più applicazioni condividono la stessa struttura a oggetti, Visitor consente di inserire le operazioni solo nelle applicazioni che ne hanno bisogno.
- Quando le classi che definiscono la struttura di oggetti cambiano raramente, ma si ha il bisogno di definire spesso nuove operazioni sulla struttura.

Struttura

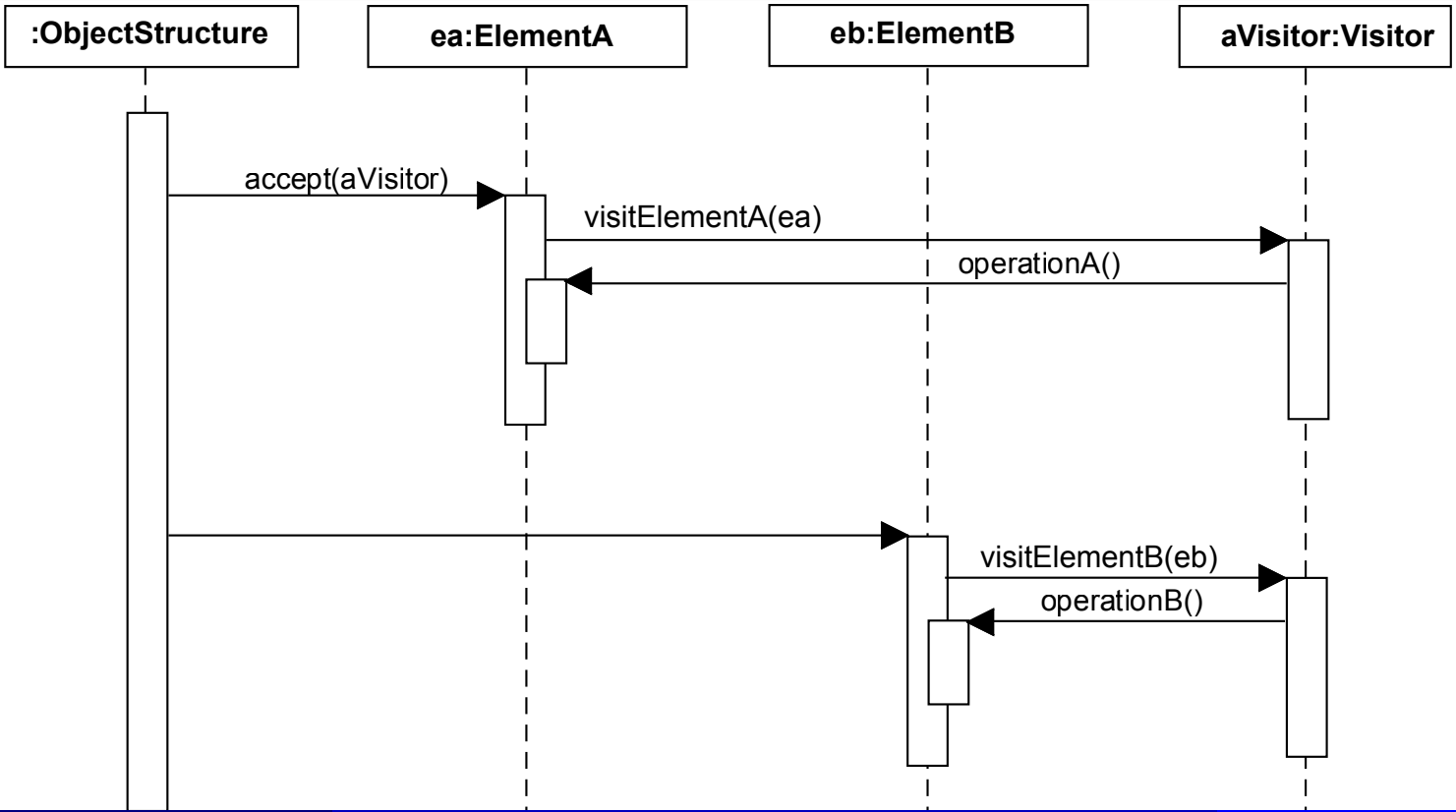

Client
«uses»
«interface»
Visitor
+visitElementA(:ElementA)
+visitElementB(:ElementB)
«uses»
ConcreteVisitor1
+visitElementA(:ElementA)
+visitElementB(:ElementB)
ConcreteVisitor2
+visitElementA(:ElementA)
+visitElementB(:ElementB)
ObjectStructure
«interface»
Element
+accept(:Visitor)
ElementA
+accept(v:Visitor)
+operationA()
ElementB
+accept(v:Visitor)
+operationB()
v.visitElementA(this)
v.visitElementB(this)

Partecipanti

- **Visitor**:
  - dichiara un metodo di visita per ogni classe concreta della struttura.
  - il nome e la firma del metodo identificano la classe che invia la richiesta di visita al visitor in modo che il visitor possa identificare la classe dell'elemento che sta per visitare
- **ConcreteVisitor**:
  - implementa i metodi definiti da `Visitor`.
  - ogni metodo implementa un *frammento* dell'algoritmo definito per la struttura.
  - `ConcreteVisitor` fornisce il contesto per l'algoritmo e memorizza il suo stato locale che accumula durante la visita della struttura.
- **Element**:
  - definisce il metodo `accept()` che riceve un `Visitor`.
- **ConcreteElement**s:
  - implementano il metodo`accept()` che riceve un `Visitor`.
- **ObjectStructure**:
  - può enumerare gli elementi che la costituiscono.
  - può essere realizzata come un *Composite* o come una collezione di elementi.

Collaborazioni

- Un client che usa il pattern Visitor deve creare un `ConcreteVisitor` e quindi attraversare la la object structure, visitando ogni elemento con il visitor.
- Quando un elemento viene visitato, esso invoca il metodo sul `Visitor` che corrisponde alla sua classe
- L'elemento fornisce se stesso come parametro attuale di tale invocazione per consentire al visitor di accedere al suo stato, ove necessario.



Conseguenze

- *Visitor permette di aggiungere facilmente nuove operazioni.* È possibile definire una nuova operazione per una struttura a oggetti semplicemente definendo un nuovo `Visitor`. Al contrario, se le funzionalità sono distribuite su molte classi, per definire una nuova operazione occorrerà modificare ogni classe della struttura di oggetti.
- *Un visitor riunisce le operazioni correlate e separa le operazioni scorrelate.* Le funzionalità correlate non sono distribuite su tutte le classi costituenti la struttura a oggetti; sono localizzate in un `Visitor`. Insiemi di funzionalità scorrelate sono suddivisi nelle sottoclassi di `Visitor`. Questo semplifica sia le classi che definiscono gli elementi sia gli algoritmi definiti nei `Visitor`. Le strutture dati specifiche di un algoritmo possono essere nascoste all'interno del `Visitor` corrispondente.
- *Aggiungere nuove classi* `ConcreteElement` è difficile. Ogni nuovo `ConcreteElement` richiede l'introduzione di un nuovo metodo in `Visitor` e di conseguenze in tutte le classi che realizzano la sua interfaccia. La gerachia di classi di `Visitor` può risultare di difficile manutenzione quando vengano definite frequentemente nuove classi `ConcreteElement`.

Conseguenze

- *Visitare le classi di una gerarchia.* A differenza di `Iterator` che consente di visitare oggetti di una struttura a patto che abbiano un tipo comune, `Visitor` non è soggetto a tale vincolo.
- *Accumulare stato.* `Visitor` può mantenere uno stato mentre attraversa ogni elemento della struttura a oggetti. Senza `Visitor` occorrerebbe passare lo stato ai metodi che implementano l'attraversamento come parametro aggiuntivo, oppure i metodi potrebbero apparire come variabili globali.
- *Rottura dell'incapsulamento.* L'approccio adottato da `Visitor` suppone che l'interfaccia di `ConcreteElement` sia sufficientemente potente da far in modo che il `Visitor` possa svolgere i propri compiti. Ne consegue che il pattern `Visitor` impone che vengano implementati metodi pubblici per esporre lo stato interno di un elemento, il che può compromettere l'incapsulamento dell'elemento stesso.

Implementazione

- *Inoltro doppio*. Il pattern `Visitor` consente in modo efficace di aggiungere operazioni a classi senza cambiare le classi stesse. Il visitor ottiene questo risultato utilizzando una tecnica nota come **inoltro doppio** (*double-dispatch*). L'operazione che viene eseguita dipende sia dalla richiesta che dal tipo di destinatario. "Inoltro doppio" significa semplicemente che l'operazione che verrà eseguita dipende dalla richiesta e dal tipo di due destinatari. Il metodo `accept` è un caso di inoltro doppio. Il suo significato dipende da due tipi: il tipo di `Visitor` e il tipo di `Element`. L'inoltro doppio permette al `Visitor` di eseguire operazioni diverse su ogni classe di elemento.
- *Chi è responsabile dell'attraversamento della struttura a oggetti?* Un visitor deve attraversare ogni elemento della struttura a oggetti. Si può attribuire la responsabilità dell'attraversamento a tre diverse entità: alla struttura a oggetti, al visitor o a un oggetto `Iterator` separato e specifico. Di solito è la struttura a oggetti ad essere responsabile dell'iterazione: un oggetto composito di solito attraverserà se stesso implementando `accept()` in modo tale che chiami ricorsivamente `accept()` sui suoi figli. Potrebbe essere necessario inserire la strategia di attraversamento nel `Visitor` quando è particolarmente complesso, come nel caso in cui dipenda anche dai risultati dei metodi del `Visitor` sulla struttura di oggetti.

Pattern Correlati

- `Composite`: `Visitor` può essere usato per applicare un'operazione su una struttura di oggetti implementata attraverso il pattern `Composite`.
- Interpreter: `Visitor` può essere usato per eseguire l'effettiva interpretazione di un'espressione.