Design Pattern 4

Introduzione

Obiettivi:

- Applicare propriamente i pattern del modello GoF

Questa sezione rappresenta il naturale conseguimento del documento *Design Pattern 3*, in cui sono elencati ulteriori pattern *comportamentali*, *creazionali* e *strutturali* del *catologo GoF*.

Memento (Behavioral Pattern)

Problema
Come ripristinare un oggetto al suo stato precedente?

Soluzione
Catturare ed esternalizzare lo stato interno dell'oggetto in maniera tale che possa essere restituito e ripristinato successivamente.

Memento promuove l'utilizzo e l'implementazione di azioni di *rollback*, le quali permettono il ripristino di un sistema oppure di un'istanza di classi allo stato precedente a *modifiche*.

In relazione ad un *pattern comportamentale* simile, è progettata un'entità software dedita esclusivamente alla memorizzazione dello *snapshot* creato, la quale non deve sviluppare alcuna logica algoritmica differente; si ricordino i principi del modello *GRASP*, in particolare *high cohesion*, in cui classi software devono mantenere una *coesione funzionale* al loro interno, ossia i metodi sviluppati devono garantire la modellazione di un unico comportamento, affinchè sia concrettizzata una soluzione di *elevata qualità*. Il termine che si occupa di garantire un approccio simile è definito *Memento*.

Pur di evitare incompresioni si attuano definizioni dell'entità coinvolte, in cui si osserva:

- *Originator*, colei che pone e richiede la salvaguardia, all'interno di strutture dati apposite, del proprio stato, affinchè sia ripristinabile
- *Caretaker*, entità affine al *domain layer*, la quale pone la ragione e l'arco temporale in cui il *Originator* debba memorizzare e ripristinare il proprio stato. Per cui riassumendo raffigura il mandante dell'azione di ripristino
- *Memento*, classe software progettata univocamente per la memorizzazione della *rappresentazione opaca* dello stato, in modo tale che sia incomprensibile per *high level classes* ed entità software, al di fuori di *Originator*

Un approccio come quello descritto è attuato per azioni di *serializzazione*, ossia si tenta di rappresentare *stati* di un oggetto attraverso una combinazione di byte, pur di adeguare operazioni di *storage* all'interno di strutture dati persistenti. Infatti, in assenza di *Memento*, sarebbe piuttosto complicato realizzare un *vincolo comportamentale* come quello precedente, poichè non sarebbe possibile adeguare la traduzione in *formato binario*, dato che linguaggi di programmazione legati al paradigma degli oggetti sostengono il principio di *incapsulamento*, secondo cui gli attributi debbano essere privati oppure protetti all'interno di ogni singola classe, causandone la violazione.

Iterator (Behavioral Pattern)

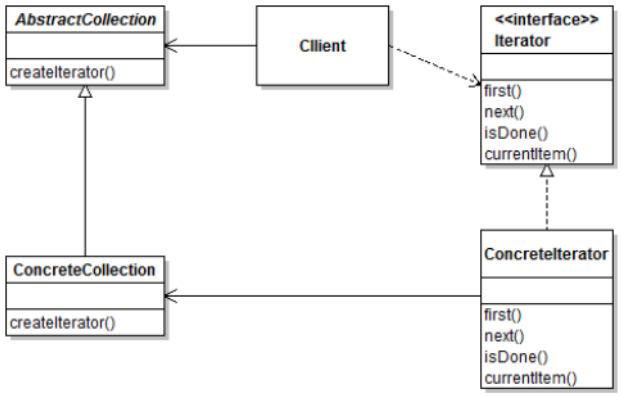
Problema
Come astrarre algoritmi che permettano l'accesso a strutture dati indipendentemente dalla collezione analizzata?

Soluzione
Provvedere un modo per accedere sequenzialmente agli elementi di una collezione in modo tale che la sua rappresentazione non sia vincolante per l'iteratore.

L'idea pone che alla classe *Iterator* sia assegnata la responsabilità di accedere e attraversare l'oggetto aggregato fornendo un protocollo comportamentale standard, a cui è applicata una strategia di *astrazione* che provvede alla separazione della nozione di algoritmo dal concetto di struttura dati. La motivazione consiste nell'intento di garantire una tipologia di programmazione affine al paradigma orientato agli oggetti, riducendo in questa maniera l'ammontare di configurazioni e implementazioni singole per ogni struttura dati.

A titolo semplificativo è attuato un *approccio* che possa favorire l'utilizzo di *algoritmi* indipendentemente dalla *struttura dati*, dove un atteggiamento differente richiederebbe un esteso range di *permutazioni* da sviluppare e mantenere.

Caso di studio



Si analizzano le entità che rappresentano la raffigurazione, in cui:

- *Client*, *high level class* utilizzata da utenti finali, affine al domain model
- *AbstractCollection*, al suo interno è mantenuto lo schema principale dell'algoritmo affinchè sottoclassi possano implementare step mancanti a seconda delle proprie specifiche, si osservi che un approccio del genere è stabilito dal *behavioral pattern template method*
- *ConcreteCollection*, *sottoclasse* derivata dalla *superclasse AbstractCollection*, la quale eredita attributi e metodi su cui sarà certamente adeguato *polimorfismo*. Al suo interno è posta la responsabilità *createIterator()*, ossia una *doing resposability* che consenta la creazione e l'utilizzo di un'istanza della classe *Iterator*
- *InterfaceIterator*, *interfaccia* applicata per soddisfare i principi di qualità del software del modello *SOLID*, soprattutto in relazione al *dipendency inversion principle* in cui

classi legate alla logica algoritmica non debbano dipendere da entità affini al mondo reale; rispetto all'esempio, sarà mantenuta un'istanza del layer astratto all'interno del termine *Client*, affinchè possa usufruire di metodi di *accesso* e di *attraversamento* della *collezione* di oggetti mediante l'*iteratore*

- *ConcreteIterator*, entità software che estende le funzionalità dell'*interfaccia*, in modo tale che siano garantiti comportamenti interpellabili da collezioni di oggetti indipendentemente dalla *struttura dati*

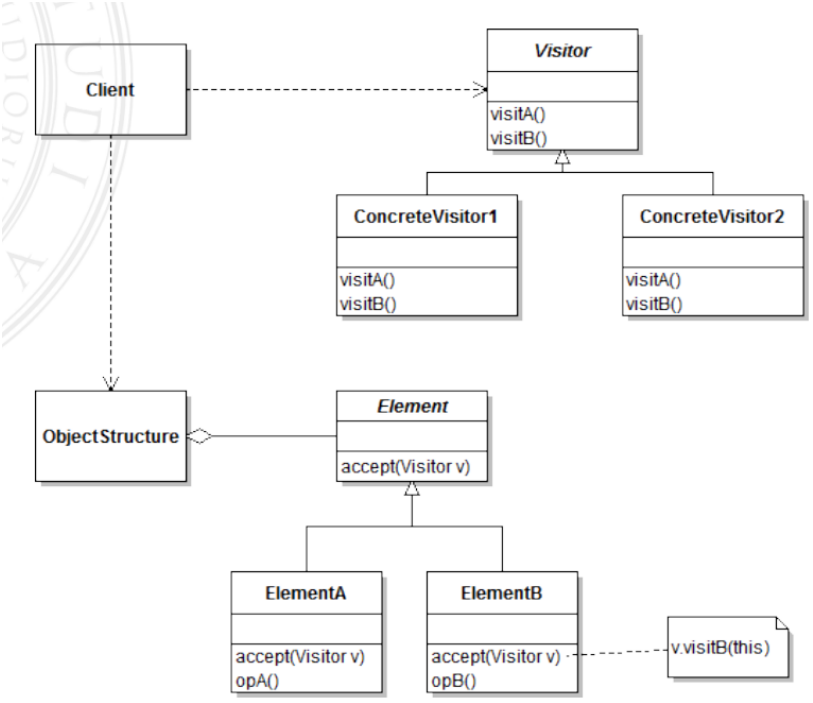
Visitor

Problema
Come sviluppare un'applicazione comune a tutti gli elementi di una collezione, qualora all'interno della struttura dati di riferimento siano memorizzati oggetti eterogenei fra loro?

Soluzione
Attuare una duplice gerarchia affinchè possano essere imbastiti flussi di controllo indipendentemente dalla tipologia di oggetto riscontrato.

Una prima risoluzione potrebbe essere garantita tramite l'impiego di un *Iterator*, il quale garantisce l'accesso e l'attraversamento degli oggetti aggregati a cui poi sarà possibile attuare funzionalità desiderate; oppure automatizzare l'approccio mediante *Visitor*. La volontà di adeguare un pattern comportamentale simile avviene qualora si tenti di aumentare il numero di funzionalità, senza modificare classi concrete riferite agli *Elementi* su cui opera il sistema software.

Caso di studio



Prima di addentrare la raffigurazione su specifiche comportamentali degli elementi che contraddistinguono il modello, si narrano due espressioni fondamentali per concepire al meglio l'intento del pattern, ossia:

- L'obiettivo consiste nella creazione di funzionalità astratte che possano essere applicate ad una gerarchia di elementi, anche se eterogenei
- Si promuove la progettazione di classi affini a singoli *Elementi* che non abbiano un elevato carico lavorativo, rispettando l'intuizione del pattern *high cohesion*, ossia all'interno dell'entità software sono implementati metodi coesi e che mirino a modellare un unico comportamento

L'implementazione procede mediante degli step specifici, suddivisi in:

- Implementare una gerarchia di visitatori, in cui sia posta una superclasse che abbia al suo interno metodi parzialmente definiti, in maniera tale che sottoclassi possano ereditare il comportamento affinchè sia specializzato a seconda delle richieste oppure della struttura dati. In questo contesto *visit()* accetta un solo argomento riferito al puntatore dell'*elemento* della collezione, pur di favorire la creazione del *flusso di controllo*
- Nuovamente, è sviluppata un'ulteriore gerarchia riferita agli elementi aggregati nella collezione, i quali promuovono il metodo *accept()* in cui è posto come parametro l'oggetto della classe *Visitor* in maniera tale che sia concretizzata la referenza
- Infine, se le classi dedite ad utenti finali, come *Client*, richiedano di attuare un'operazione di visita inerente a qualsiasi computazione, provvederà innanzitutto all'istanziazione di *Visitor*, a seconda dell'approccio desiderato e specificato in sottoclassi, in modo tale che sia trasferita al metodo *accept()* di *Element*, risalendo agli oggetti aggregati nella struttura dati; terminata la *call* della funzionalità, il flusso di controllo è delegato alla prima gerarchia di visitatori

Concludendo, è bene attuare il pattern *Visitor* qualora si è certi che gli elementi trattati siano compositi, come all'interno di un *albero*, ed attuare un approccio come quello descritto permette di scorrere in maniera totalmente autonoma la collezione.