Il pattern Memento

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@unical.it
Web: http://angelo.furfaro.dimes.unical.it

Memento

Classificazione

- Scopo: comportamentale
- Raggio d'azione: oggetti

Altri nomi

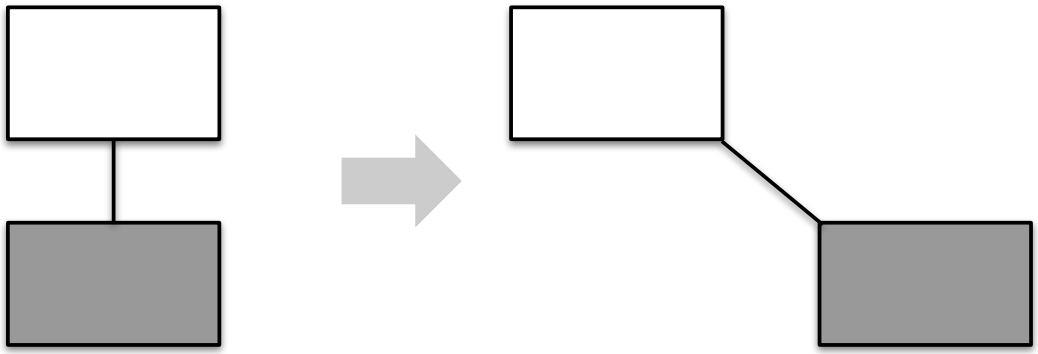
Token

Scopo

- Catturare ed esternare lo stato interno di un oggetto, senza violare l'incapsulamento, in modo tale che, in un secondo momento, sia possibile ripristinare un oggetto nello stato esportato.

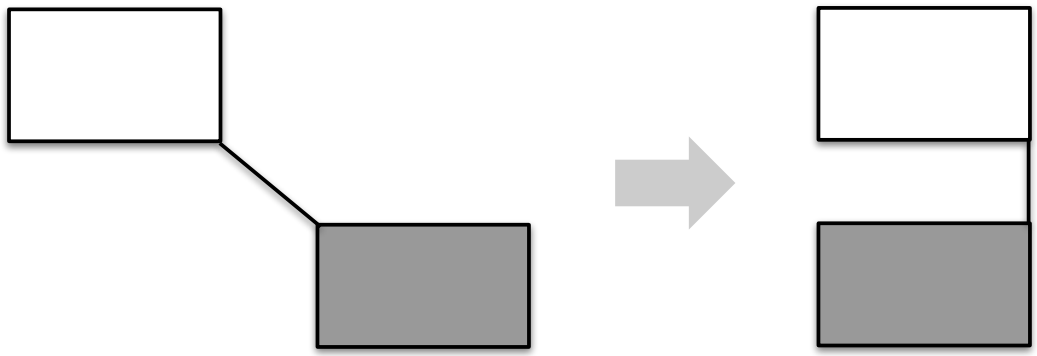
Motivazione

- Talvolta è necessario memorizzare lo stato interno di un oggetto.
- Ciò può accadere quando si devono implementare meccanismi di ripristino che per mettano all'utente di tornare indietro in seguito a prove di operazioni, o di ripristinare uno stato affidabile dopo che si sono verificate delle condizioni di errore.
- Si deve poter salvare lo stato così che sia possibile riportare gli oggetti a un loro stato precedente. Un oggetto però di solito ha il proprio stato parzialmente o totalmente incapsulato, inaccessibile dagli altri oggetti e impossibile da salvarsi esternamente.
- Rendere visibile questo stato violerebbe l'incapsulamento e comprometterebbe l'affidabilità ed estendibilità dell'applicazione.
- Si consideri, per esempio, un editor grafico che permette di connettere gli oggetti fra loro.
- Un utente può connettere due rettangoli attraverso una linea, e i rettangoli mantengono questa connessione a prescindere da come vengono mossi. L'editor garantisce che la linea si modificherà per mantenere la connessione.



Motivazione

- Un modo noto per mantenere le connessioni fra oggetti è l'uso di un sistema di vincoli. I siamo incapsulare questa informazione in un oggetto `ConstraintSolver` che memorizza le connessioni non appena vengono create e genera le funzioni matematiche tele descrivono.
- Queste equazioni sono calcolate e risolte ogni volta che l'utente crea una nuova connessione o comunque modifica il diagramma. `ConstraintSolver` usa i risultati di questi calcoli per riposizionare gli elementi grafici così che vengano preservarne le connessioni.
- In un'applicazione di questo tipo la gestione dell'*undo* non è facile come a prima vista potrebbe sembrare. Un modo banale di implementare il ripristino di un precedente stato, a seguito di un'operazione di spostamento, consta nel memorizzare la distanza originaria di movimento e riposizionare l'elemento spostato nella sua vecchia posizione, ma questo metodo non garantisce che gli oggetti abbiano la stessa rappresentazione a video.



Motivazione

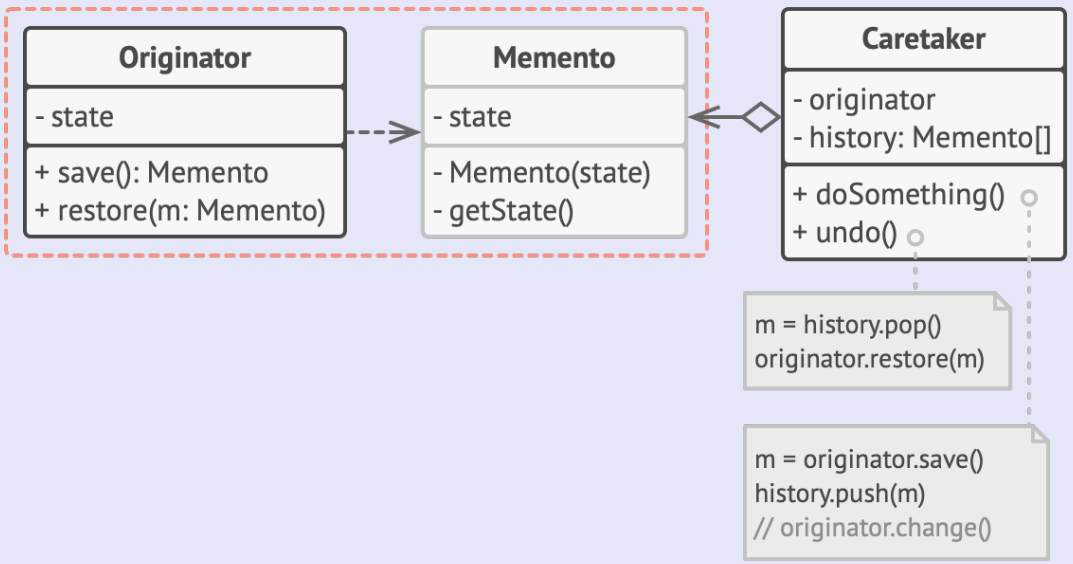
- È possibile risolvere questo problema tramite il pattern `Memento`.
- Il memento è un oggetto che memorizza un'istantanea dello stato interno di un altro oggetto, l'originatore del memento.
- Il meccanismo di ripristino richiede un memento dall'originatore quando ha bisogno di memorizzare un particolare stato dell'originatore.
- L'originatore inizializza il memento con le informazioni necessarie al ripristino del suo stato attuale.
- Solo l'originatore può memorizzare e recuperare dati in un memento, che è opaco per ogni altro oggetto.

Applicabilità

È opportuno usare il pattern `Memento` quando:

- si deve memorizzare un'istantanea (totale o parziale) dello stato di un oggetto così da poterla ripristinare in un secondo tempo;
- un'interfaccia diretta per accedere allo stato esporrebbe dettagli implementativi e violerebbe l'incapsulamento dell'oggetto.

Struttura



Partecipanti

- **Memento**
  - Memorizza lo stato interno dell'oggetto `Originator`. `Memento` può memorizzare quel poco che basta affinché l'`Originator` possa ripristinare il suo stato interno.
  - Non permette l'accesso alla sua struttura dati se non all'`Originator`. I `Memento` hanno effettivamente due interfacce. Il `Caretaker` vede un'interfaccia ridotta di `Memento`, può solo passare il `Memento` ad altri oggetti. `Originator` al contrario vede un'interfaccia estesa, tramite la quale è possibile accedere a tutti i dati necessari per ripristinare il suo stato precedente. Idealmente solo l'`Originator` che ha prodotto il `Memento` ha il permesso di accedere allo stato interno del `Memento`.
- **Originator**
  - Crea un `Memento` contenente un'istantanea del proprio stato interno corrente.
  - Usa un `Memento` per ripristinare il proprio stato interno.
- **Caretaker**
  - È responsabile di memorizzare i `Memento`.
  - Non invoca operazioni né esamina i contenuti di un `Memento`.

Conseguenze

L'uso del pattern `Memento` ha svariate conseguenze.

- Preservare i confini tracciati da un corretto incapsulamento. Il pattern `Memento` non espone le informazioni che solo un `Originator` dovrebbe gestire, ma che devono comunque essere memorizzate esternamente all'`Originator`.
- Semplifica l'`Originator`. In altre architetture che salvaguardino l'incapsulamento, `Originator` mantiene al suo interno tutto lo storico degli stati richiesti dal client appesantendo l'`Originator`.
- L'uso dei memento potrebbe essere costoso. L'uso dei memento potrebbe essere oneroso se `Originator` deve copiare grosse quantità di dati nel `Memento` o se i client richiedono molti memento e li ripristinano frequentemente.
- Definizione di interfacce ridotte ed estese. In alcuni linguaggi di programmazione potrebbe essere difficile far sì che solo l'`Originator` possa accedere allo stato di un memento.
- Costi nascosti nella gestione dei memento. Il `Caretaker` è responsabile di cancellare i `Memento` che ha in gestione. Ma il `Caretaker` non conosce la quantità di dati presente in un `Memento`. Quindi un `Caretaker`, altrimenti poco oneroso, potrebbe incappare in costi elevati di gestione e memorizzazione quando gestisce dei memento.

Implementazione

Pattern correlati

- `Command` può usare `Memento` per memorizzare lo stato per operazioni che consentano l'annullamento..