Il pattern Command

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

Command

Classificazione

- Scopo: comportamentale
- Raggio d'azione: oggetti

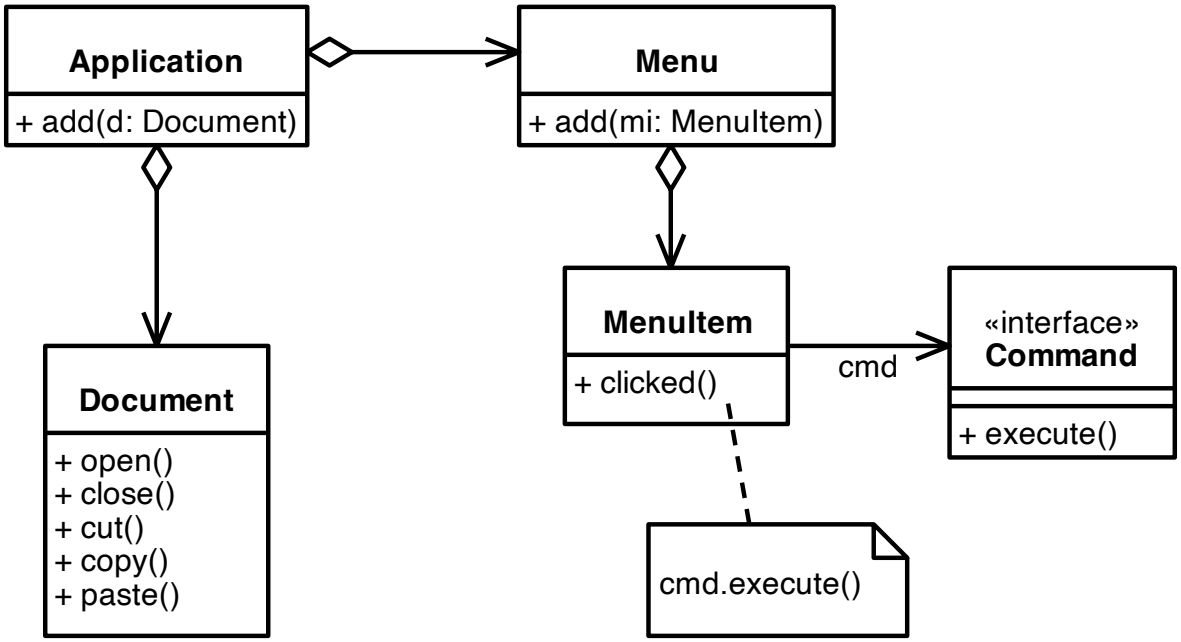
Altri nomi

Action, Transaction

Scopo

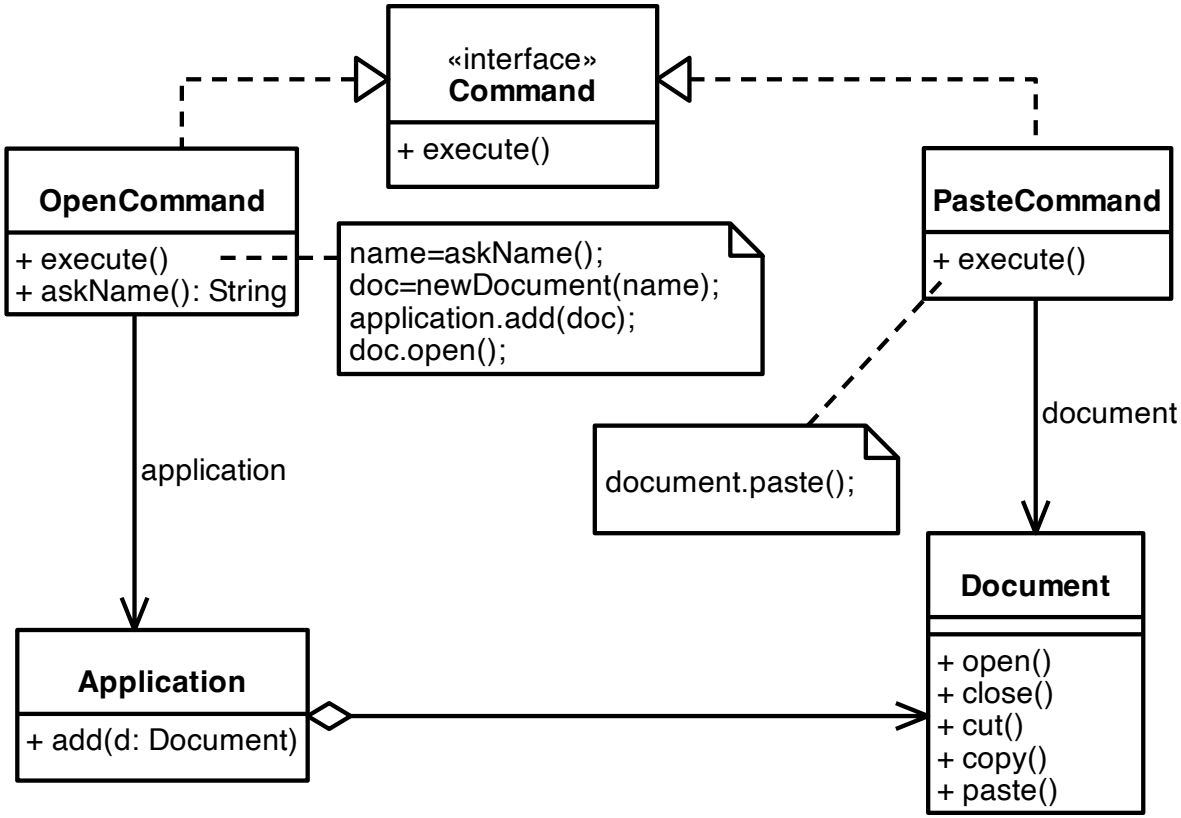
- Incapsula una richiesta in un oggetto, consentendo di parametrizzare i client con richieste diverse, accodare o mantenere uno storico delle richieste e gestire richieste cancellabili.

Motivazione



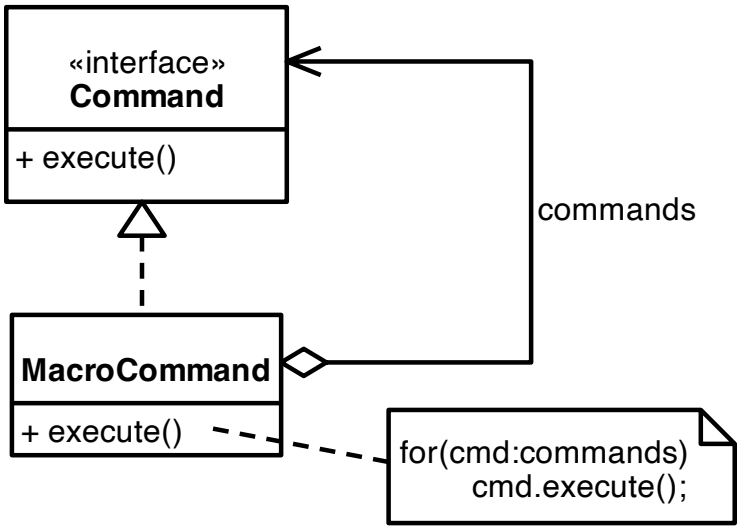
- Talvolta è necessario inoltrare richieste a oggetti senza conoscere nulla dell'operazione richiesta o del destinatario della richiesta.
- Per esempio, i toolkit per la costruzione di interfacce grafiche includono oggetti come pulsanti e menu adibiti a inoltrare le richieste in base alla selezione dell'utente, ma l'ambiente non può implementare la richiesta in modo esplicito nel pulsante o nel menu poiché solo le applicazioni che usano il toolkit conoscono il modo in cui la richiesta deve essere portata a termine.
- Lo sviluppatore del toolkit non può sapere chi sarà il destinatario della richiesta o come la richiesta dovrà essere gestita.
- Il pattern `Command` permette agli oggetti dell'ambiente di inoltrare richieste a oggetti sconosciuti dell'applicazione trasformando la richiesta in un oggetto.
- Questo oggetto può essere salvato e passato come un normale oggetto.

Motivazione



- L'interfaccia `Command` è il tipo base degli oggetti che rappresentano i comandi.
- Le classi concrete che realizzano `Command` specificano un'azione su destinatario di cui mantengono il riferimento implementando il metodo `execute()`. Il destinatario sa come portare a termine la richiesta.
- Nell'esempio ciascun `MenuiItem` è configurato con uno specifico `Command`. Nel caso di `PasteCommand` il destinatario è un oggetto `Document`, nel caso di `OpenCommand` esso crea un nuovo documento e lo aggiunge all'oggetto (destinatario) `Application`.

Motivazione



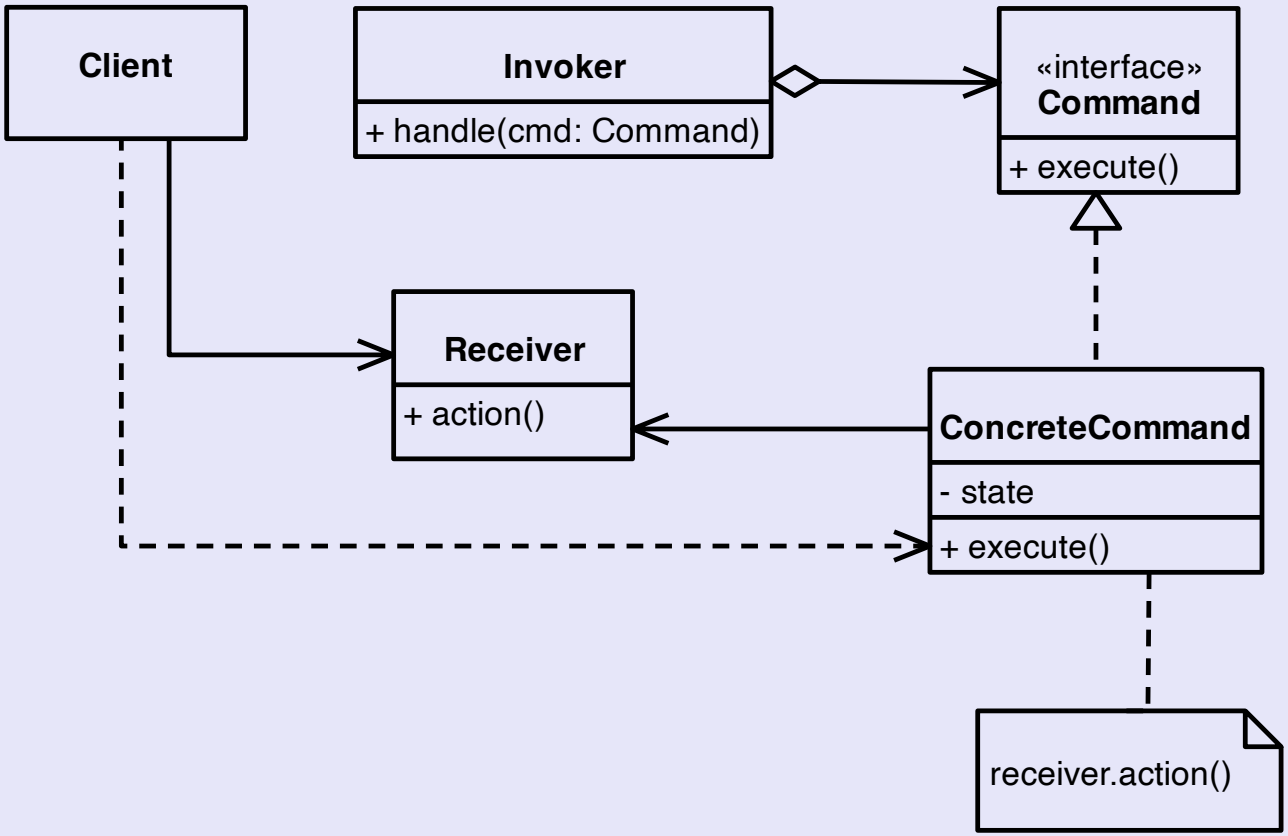
- Un `MenuItem` potrebbe aver bisogno di eseguire una sequenza di comandi. È possibile definire una classe `MacroCommand` che mantiene il riferimento ad un aggregato di oggetti `Command` che corrispondono alla sequenza dei comandi da eseguire.
- `MacroCommand` non ha un destinatario specifico. Ciascun comando della sequenza conosce il proprio destinatario.
- Si noti come il pattern Command disaccoppi l'oggetto che invoca una richiesta dall'oggetto che possiede le informazioni per portarla a termine.
- Un'applicazione può fornire sia un pulsante che una voce di menu per accedere a una funzionalità semplicemente facendo in modo che il menu e il pulsante condividano la stessa istanza di una classe concreta che realizza `Command`.
- È possibile sostituire i comandi dinamicamente, caratteristica utile se si vogliono implementare menu sensibili al contesto.

Applicabilità

È opportuno impiegare il pattern `Command`:

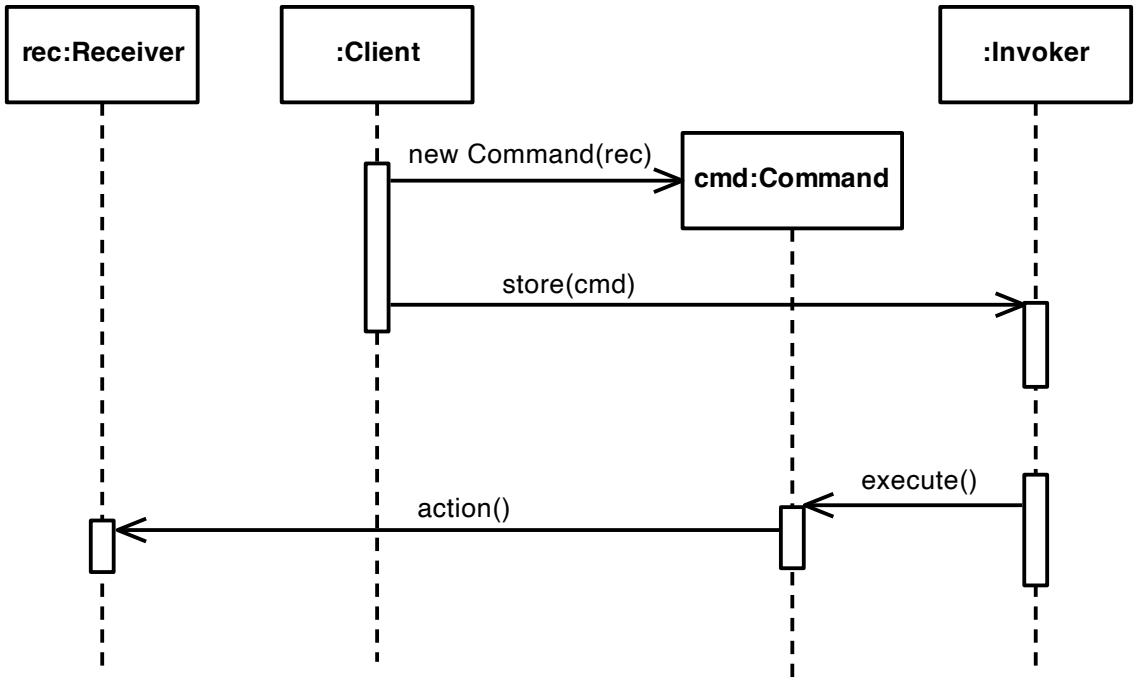
- Per parametrizzare oggetti rispetto a un'azione da eseguire. Gli oggetti `Command` sono un rimpiazzo *ad oggetti* delle funzioni *callback*.
- Per specificare, accodare ed eseguire le richieste in tempi diversi. Un oggetto command può avere un ciclo di vita indipendente dalla richiesta originale. Se i destinatario di una richiesta può essere rappresentato indipendentemente dallo spazio di indirizzi specifico, sarà possibile trasferire un oggetto `Command` in un processo diverso e quindi portare a termine l'operazione in un processo diverso.
- Per consentire l'annullamento di operazioni. Il metodo `execute()` di `Command` può memorizzare lo stato necessario per annullare i suoi effetti nel comando stesso, L'interfaccia `Command` dovrà avere un nuovo metodo `undo()` che annulli gli effetti di una precedente chiamata a `execute()`. I comandi che sono stati eseguiti tramite `execute()` sono memorizzati in uno storico.
- Per organizzare un sistema in operazioni d'alto livello a loro volta basate su operazioni primitive. Un'architettura di questo genere è comune in sistemi per la gestione di transazioni.

Struttura


Client
Invoker
+ handle(cmd: Command)
«interface»
Command
+ execute()
Receiver
+ action()
ConcreteCommand
- state
+ execute()
receiver.action()

Partecipanti

- **Command**: dichiara un'interfaccia per l'esecuzione di un'operazione generica.
- **ConcreteCommand** (`PasteCommand, OpenCommand`): definisce un legame fra un oggetto destinatario e un'azione. Implementa il metodo `execute()` invocando il metodo (i metodi) corrispondente sul `Receiver`.
- **Client**(`Application`): crea un'istanza concreta di `Command` e ne imposta il `Receiver`.
- **Invoker**(`MenuItem`): chiede a `Command` di portare a termine la richiesta
- **Receiver**(`Document, Application`): conosce il modo di svolgere le operazioni associate a una richiesta. Qualsiasi classe può essere vista come `Receiver`.



Conseguenze

- ☺ `Command` disaccoppia l'oggetto che invoca un'operazione da quello che conosce come portarla a termine.
- ☺ Gli oggetti `Command` sono oggetti a tutti gli effetti. Possono essere manipolati ed estesi con un qualsiasi altro oggetto.
- ☺ È possibile comporre più comandi in un comando composito. Un esempio è la classe `MacroCommand` precedentemente descritta. In generale, i comandi compositi sono un' istanza del pattern `Composite`.
- ☺ Risulta facile aggiungere nuovi comandi poiché non è necessario modificare le classi esistenti.

Implementazione

- Ouanto deve essere intelligente un comando? Un `Command` può avere un gran numero di funzionalità: da una parte definisce un accoppiamento tra un destinatario e le azioni necessarie per portare a termine la richiesta. Dall'altra implementa tutto internamente senza delegare alcuna richiesta a uno o più destinatari. Fra questi due estremi ci sono i `Command` capaci di trovare il corretto destinatario in modo dinamico.
- Supporto per l'annullamento e per la riesecuzione dei comandi. Un `Command` concreto potrebbe aver bisogno di salvare informazioni addizionali sullo stato dell'applicazione. Lo stato può essere costituito da:
  - l'oggetto `Receiver`, che svolge effettivamente le operazioni necessarie invocate dal `Command` in risposta a una richiesta;
  - gli argomenti delle operazioni eseguite sul `Receiver`;
  - tutti i valori nel `Receiver` che potrebbero cambiare come conseguenza dello svolgimento della richiesta. Il `Receiver` deve fornire operazioni che permettano al comando di riportarlo a uno stato precedente.

  Per poter gestire un solo livello di annullamento, un'applicazione ha bisogno di memorizzare solo il comando che è stato eseguito per ultimo. Per livelli multipli di annullamento e riesecuzione, l'applicazione ha bisogno di uno storico dei comandi di invocati

Pattern correlati

- Un `Composite` può essere usato per implementare `MacroCommand`.
- Un `Memento` può memorizzare lo stato necessario per implementare funzionalità di annullamento dei comandi.
- Un comando che deve essere copiato prima di essere posto nello storico si comporta come un `Prototype` .