



# Object-Oriented Design Patterns

---

## Seconda Parte



## Behavioral patterns

---

- I behavioral pattern riguardano algoritmi e assegnazioni di responsabilità tra diversi oggetti: essi descrivono pattern di comunicazioni tra oggetti, oltre che di relazioni tra gli stessi
- I *behavioral class patterns* utilizzano l'ereditarietà per distribuire comportamenti tra diverse classi
- I *behavioral object patterns* utilizzano composizione piuttosto che ereditarietà
- In generale, si può affermare che si utilizza un behavioral pattern per incapsulare un comportamento



## Creational patterns

---

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor



## Chain of Responsibility

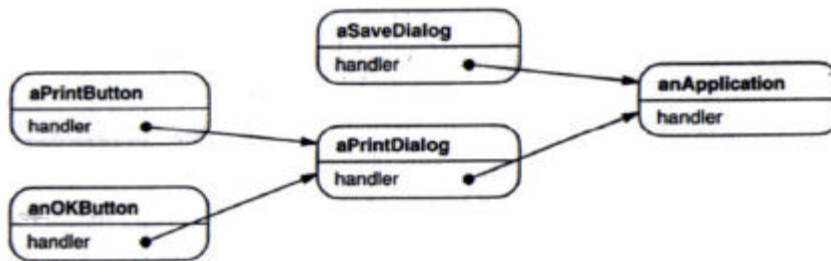
---

(Object Behavioral)

- **Intento:** Evitare l'accoppiamento tra il mittente di una richiesta e il ricevente, dando a più oggetti (collegati tra loro a catena) la possibilità di manipolare la richiesta.
- **Motivazioni:** Supponiamo di voler realizzare un help sensibile al contesto: le richieste proverranno, ad esempio, dal bottone *Ok* della finestra di stampa. Esse potranno essere gestite o dal bottone stesse, o passate alla classe *PrintDialog* che, a sua volta, potrà gestire la richiesta o inoltrarla alla classe *Application*. A seconda di chi gestirà la richiesta, l'help prodotto sarà più o meno generico.

# Chain of Responsibility

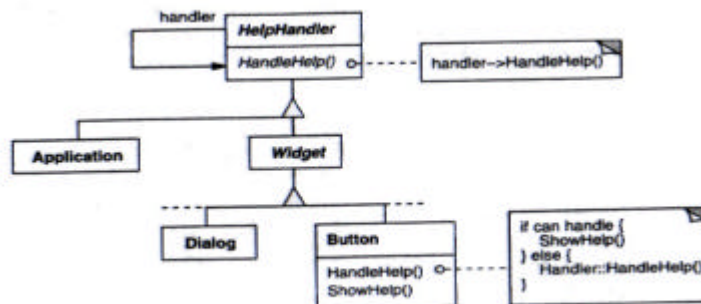
(Object Behavioral)



# Chain of Responsibility

(Object Behavioral)

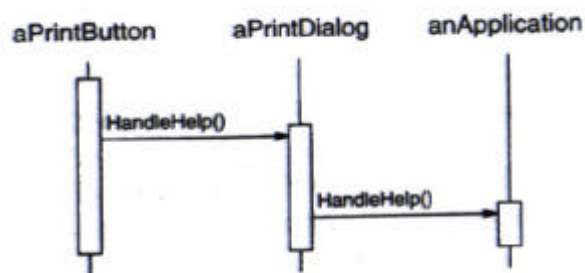
L'oggetto che esegue la richiesta non è a conoscenza di chi la gestirà. La richiesta ha dunque un **ricevente implicito**.



## Chain of Responsibility

(Object Behavioral)

Per inoltrare la richiesta e assicurarsi che il richiedente resti implicito, ciascun oggetto condivide con gli altri un'interfaccia comune per gestire le richieste (nel nostro caso, la superclasse *HelpHandler* e del metodo *HandleHelp()*).



## Chain of Responsibility

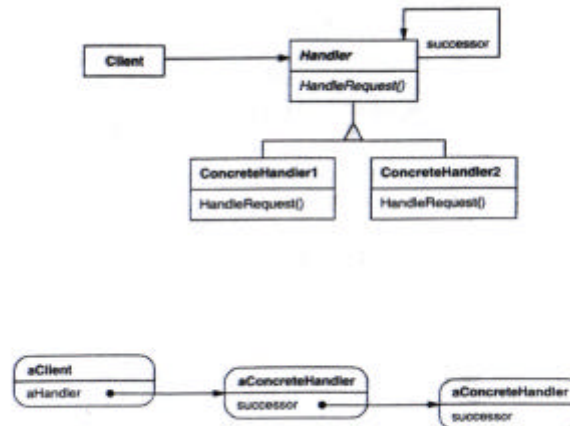
(Object Behavioral)

- **Applicabilità:** questo pattern andrebbe usato se:
  - Una richiesta può essere gestita da più di un oggetto, e il gestore non è noto a priori;
  - Si desidera inviare una richiesta a diversi oggetti senza specificare esplicitamente il ricevente;
  - L'insieme dei gestori della richiesta deve essere specificato dinamicamente

# Chain of Responsibility

(Object Behavioral)

Struttura:



# Chain of Responsibility

(Object Behavioral)

- **Collaboration:** quando un client invia una richiesta, questa è propagata lungo la catena fino al *ConcreteHandler*.
- **Conseguenze:**
  - L'accoppiamento tra il mittente di una richiesta e il gestore è ridotto;
  - Maggiore flessibilità nell'assegnare responsabilità agli oggetti;
  - La ricezione non è però garantita!

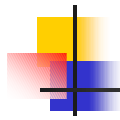


## Chain of Responsibility

(Object Behavioral)

### ■ Conseguenze:

- Isola le classi concrete;
- Facilita la portabilità;
- Aumenta la consistenza tra i prodotti;
- Per contro, inserire nuovi prodotti risulta complicato, in quanto implica cambiamenti all'*Abstract Factory*



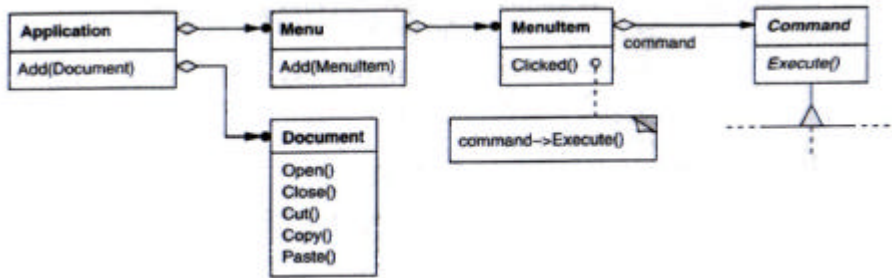
## Command

(Object Behavioral)

- **Intento:** Incapsulare una richiesta in un oggetto, in modo tale da parametrizzare i client rispetto a richieste differenti, consentendo operazioni come accordamento, logging e undo.
- **Nota come:** Action, Transaction
- **Motivazioni:** Oggetti di un toolkit possono effettuare richieste trasformando le richieste stesse in oggetti. La "chiave" del pattern è la classe *Command*, che dichiara un'interfaccia per l'esecuzione delle operazioni (metodo *Execute()*). Le classi *ConcreteCommand* specificano la coppia azione-ricevente implementando tale metodo, e immagazzinando il ricevente in una variabile di istanza.

# Command

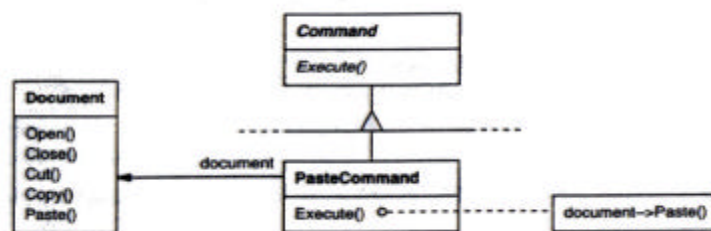
(Object Behavioral)



# Command

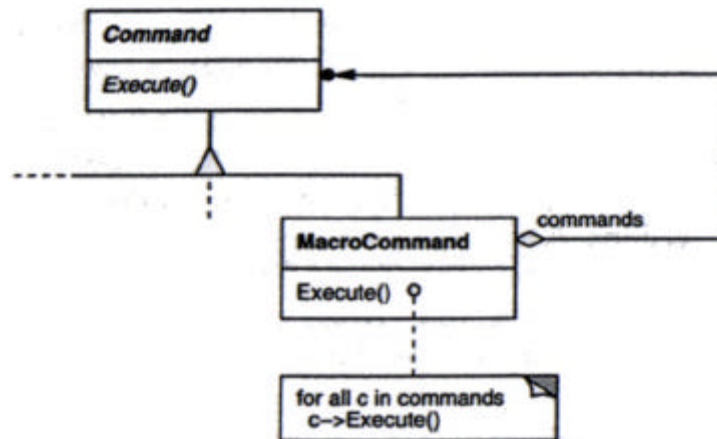
(Object Behavioral)

Come è possibile notare, i menu possono essere implementati agevolmente con tale pattern. Ciascuna scelta del *Menu* è un'istanza della classe *MenuItem*. L'*Application* crea il *Menu* e tiene traccia dei *Documents* aperti. Ciascun *MenuItem* è associato ad un'istanza di una classe *ConcreteCommand*.



# Command

(Object Behavioral)



# Command

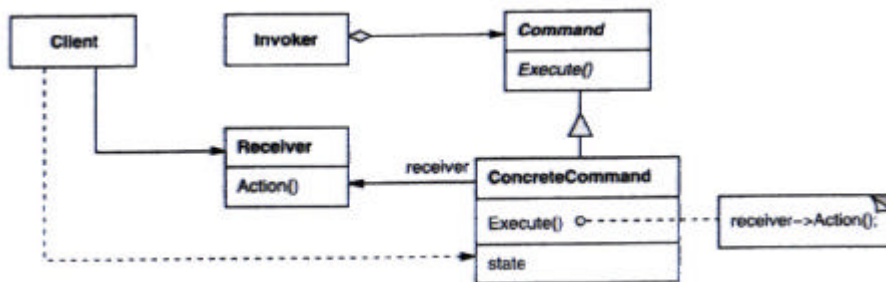
(Object Behavioral)

- **Applicabilità:** Il Command è utile per:
  - Parametrizzare oggetti rispetto ai comandi da eseguire (ciò nei linguaggi procedurali spesso avviene per mezzo di una **callback**, ovvero una funzione registrata in un certo punto per poi essere richiamata successivamente);
  - Specificare, accodare ed eseguire richieste in tempi diversi;
  - Supportare l'undo: l'operazione di *Execute* può mantenere lo stato per annullare il proprio effetto;
  - Supportare il logging dei comandi in modo da consentire il redo in caso di crash del sistema;
  - Eseguire transazioni atomiche.

# Command

(Object Behavioral)

## Struttura:



# Command

(Object Behavioral)

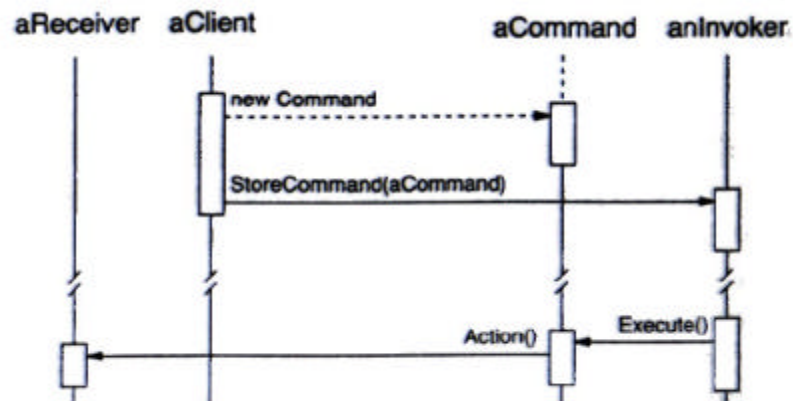
## ■ Collaborations:

- Il client crea un *ConcreteCommand* e specifica il proprio ricevente;
- L'*Invoker* immagazzina il *ConcreteCommand*;
- Il *ConcreteCommand* invoca operazioni sul ricevente per eseguire le richieste.

# Command

(Object Behavioral)

## Collaborations:

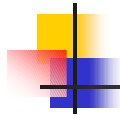


# Command

(Object Behavioral)

## ■ Conseguenze:

- Il *Command* disaccoppia l'oggetto che invoca l'operazione da quello che la esegue;
- Un *Command* può essere manipolato o esteso come qualsiasi altro oggetto;
- E' possibile assemblare *Commands* creando dei comandi composti;
- Aggiungere nuovi *Command* risulta semplice, e non richiede modifiche alle classi esistenti.



# Interpreter

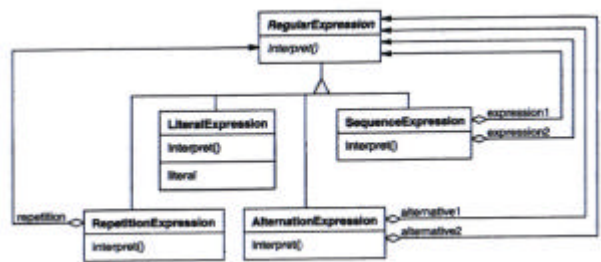
(Class Behavioral)

- **Intento:** Dato un linguaggio, definire una rappresentazione per la grammatica di tale linguaggio e un interprete del linguaggio stesso.
- **Motivazioni:** es. dovendo realizzare un generico strumento di ricerca, è possibile definire espressioni regolari in maniera tale che lo strumento ricerchi tutte le stringhe di un documento soddisfacenti tale espressione. L'*Interpreter* utilizza una classe per rappresentare ciascuna regola della grammatica. I simboli sul lato destro della regola sono variabili di istanza delle classi.



# Interpreter

(Class Behavioral)

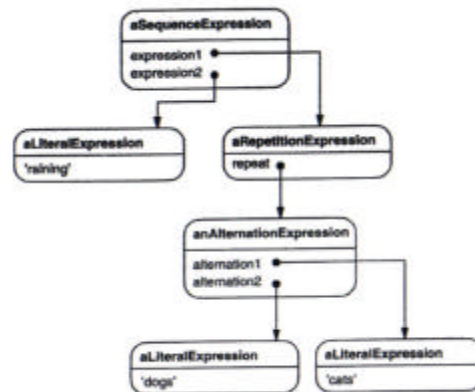




## Interpreter

(Class Behavioral)

- Ogni espressione regolare definita dalla grammatica è rappresentata da un albero sintattico composto dai istanze delle classi:



Rappresenta l'espressione **raining & (dog | cats)\***



## Interpreter

(Class Behavioral)

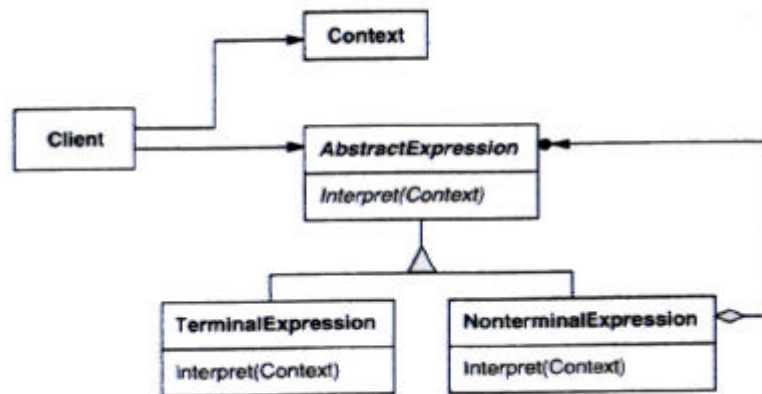
- **Applicabilità:** L'Interpreter è applicabile quando c'è un linguaggio da interpretare, i cui statements possono essere rappresentati da nodi dell'albero sintattico. Il pattern si rivela efficace se:
  - La grammatica è semplice (altrimenti la gerarchia di classi diventa complessa e difficile da gestire)
  - L'efficienza non è essenziale (interpreti basati su espressioni regolari sono più efficienti se realizzati mediante macchine a stati)



# Interpreter

(Class Behavioral)

## Struttura:

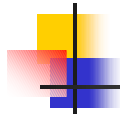


# Interpreter

(Class Behavioral)

## ■ Collaborations:

- Il client crea l'albero sintattico, composto da istanze di *NonterminalExpression* e *TerminalExpression*
- Ciascuna istanza di una espressione non terminale definisce un *Interpret* in termini di interprete di una sottoespressione. L'interprete dell'espressione terminale è il caso base del meccanismo ricorsivo
- L'operazione *Interpret* a ciascun nodo usa il contesto per immagazzinare lo stato dell'*Interpreter*.

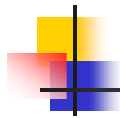


## Interpreter

(Class Behavioral)

### ■ Conseguenze:

- E' semplice modificare ed estendere la grammatica;
- Implementare una grammatica risulta, allo stesso modo, abbastanza agevole;
- Tuttavia, è difficile gestire grammatiche complesse (che implicano gerarchie complesse di classi)
- E' possibile aggiungere nuovi modi di interpretare la medesima espressione, definendo una nuove operazioni nelle classi relative alle espressioni stesse.



## Iterator

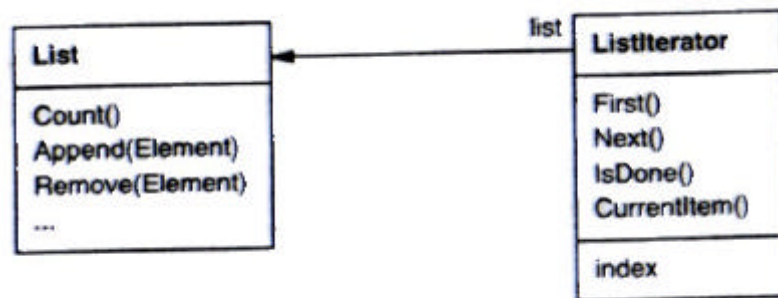
(Object Behavioral)

- **Intento:** fornire un meccanismo per accedere ad elementi di un oggetto aggregato in maniera sequenziale, nascondendone la rappresentazione
- **Nota come:** Cursor
- **Motivazioni:** Supponiamo di voler accedere sequenzialmente agli elementi di una lista. L'*Iterator* tiene traccia della posizione corrente e fornisce un'interfaccia per tale accesso mediante le seguenti operazioni:
  - **First:** Posiziona il cursore sul primo elemento della lista;
  - **Next:** Avanza di un elemento;
  - **IsDone:** Restituisce *true* se la visita è terminata;
  - **CurrentItem:** Restituisce l'elemento corrente.



## Iterator

(Object Behavioral)



## Iterator

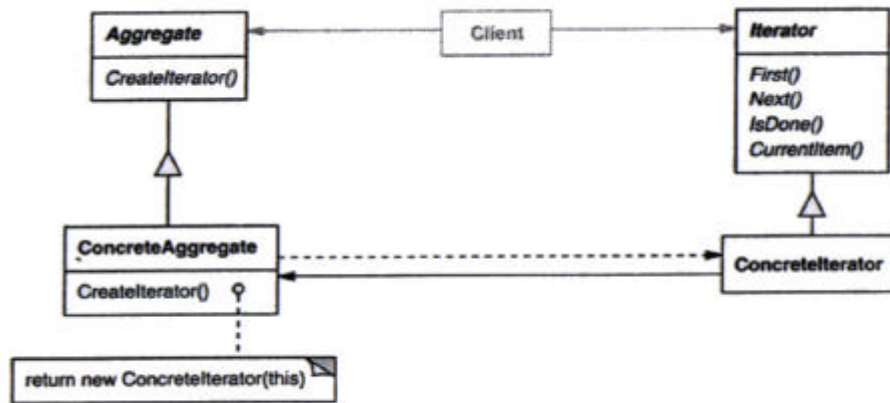
(Object Behavioral)

- E' possibile definire Iterator per diverse politiche di attraversamento (es. *FilteredIterator*, *ReverseIterator*, ecc.);
- Il problema della creazione dell'Iterator è risolto inserendo nella struttura un'operazione *CreateIterator* che restituisce un Iterator per tale struttura;
- Il client deve essere a conoscenza della struttura da attraversare, per cui potrebbe essere ragionevole generalizzare l'iterator in maniera tale da evitare modifiche del client quando la struttura dati cambia.



# Iterator

(Object Behavioral)



# Iterator

(Object Behavioral)

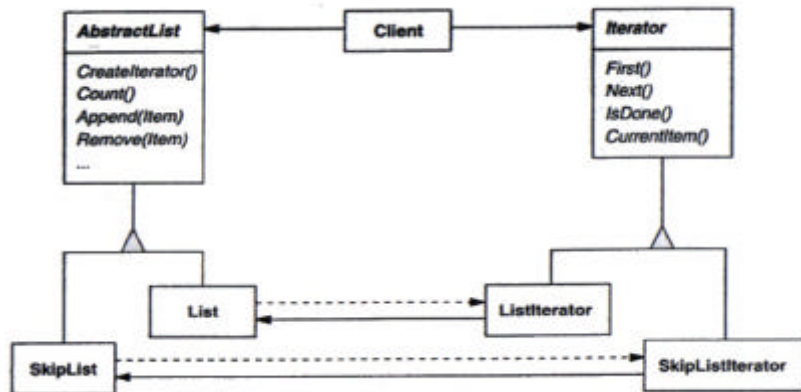
- **Applicabilità:** L'Iterator è utilizzato per:
  - Accedere al contenuto di oggetti aggregati senza esporne la rappresentazione interna;
  - Supportare modi di attraversamento multipli;
  - Fornire un'interfaccia multipla per attraversare diverse strutture (mediante il polimorfismo degli iterator)



# Iterator

(Object Behavioral)

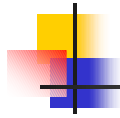
## Struttura:



# Iterator

(Object Behavioral)

- **Collaborations:** Il *ConcreteIterator* tiene traccia dell'oggetto concreto oggetto dell'attraversamento ed è in grado di determinare quale sarà il successivo
- **Conseguenze:**
  - E' possibile supportare diverse politiche di attraversamento;
  - Gli Iterator semplificano l'interfaccia dell'oggetto aggregato;
  - E' possibile eseguire contemporaneamente più di un attraversamento sullo stesso oggetto aggregato.



## Mediator

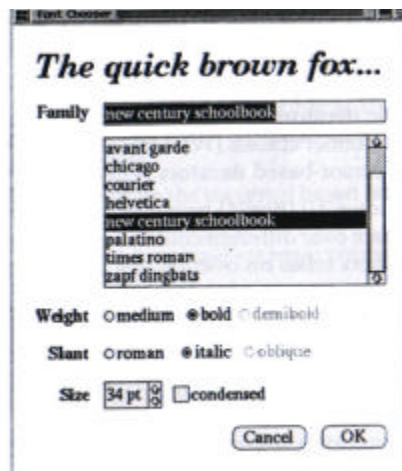
(Object Behavioral)

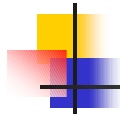
- **Intento:** Definire un oggetto che incapsula il modo in cui diversi oggetti interagiscono tra loro, evitando che tali oggetti possano riferirsi a vicenda in maniera esplicita, e consentendo di cambiare il meccanismo di interazione in maniera agevole.
- **Motivazioni:** Consideriamo es. la finestra di selezione font nella pagina successiva. Selezionando un font dalla listbox lo stato degli altri widgets della finestra cambierà. Si vuole evitare che ciò avvenga tramite scambio di messaggi direttamente tra i widgets.



## Mediator

(Object Behavioral)

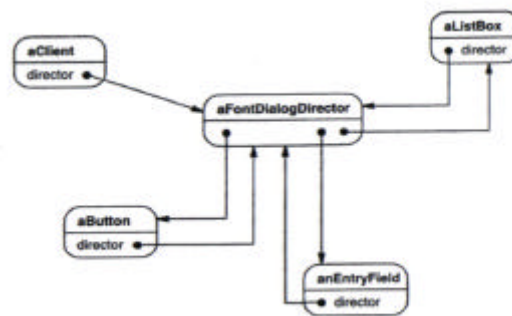




# Mediator

(Object Behavioral)

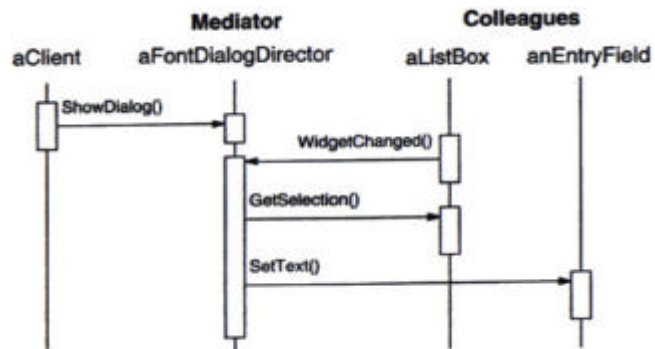
Soluzione: inserire un oggetto *Mediator*, responsabile di controllare e coordinare le interazioni di gruppi di oggetti; ciascun oggetto ha conoscenza soltanto del Mediator.



# Mediator

(Object Behavioral)

Successione di eventi mediante i quali una selezione nella list box provoca la modifica nell'entry field.

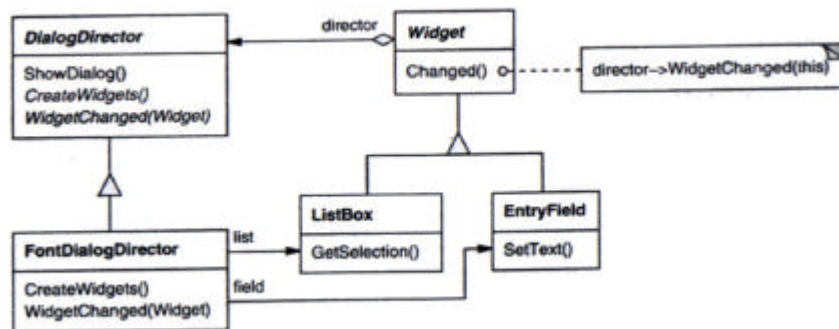




# Mediator

(Object Behavioral)

Ecco invece come integrare il tutto in una gerarchia di classi:



# Mediator

(Object Behavioral)

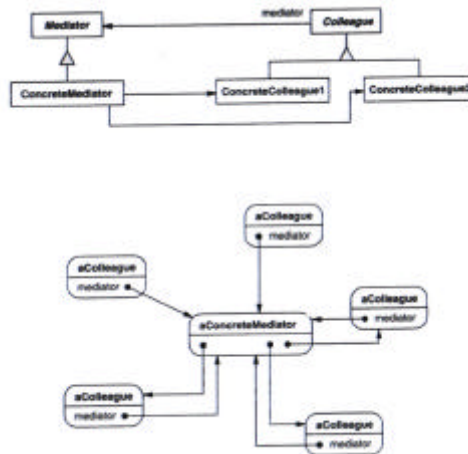
- **Applicabilità:** Il Mediator è utilizzabile se:
  - Un insieme di oggetti comunicano in maniera ben definita (ma complessa) e le interdipendenze risultando difficili da comprendere;
  - Il riuso di un oggetto è difficoltoso in quanto questo comunica con molti oggetti;
  - Un behavior distribuito tra molte classi è customizzabile senza effettuare molto subclassing.



# Mediator

(Object Behavioral)

## Struttura:



# Mediator

(Object Behavioral)

- **Collaborations:** gli oggetti inviano e ricevono richieste tramite il Mediator, che si occupa di fare routing.
- **Conseguenze:**
  - Subclassing limitato;
  - Disaccoppiamento tra oggetti;
  - Semplificazione del protocollo di comunicazione tra oggetti;
  - Astrae come gli oggetti cooperano;
  - Centralizza il controllo.



## Memento

(Object Behavioral)

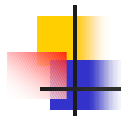
- **Intento:** senza violare l'incapsulamento, cattura ed esternalizza lo stato interno di un oggetto in modo che possa essere ripristinato in futuro
- **Nota come:** Token
- **Motivazioni:** Supponiamo di voler realizzare un editor grafico dotato di funzionalità di *undo*. A questo punto è possibile mettere su una struttura in cui un oggetto, il *Memento*, conserva lo stato di un'altro oggetto, l'*Originator*. Il *Memento* è visibile soltanto dall'*Originator*.



## Memento

(Object Behavioral)

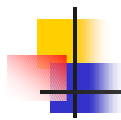
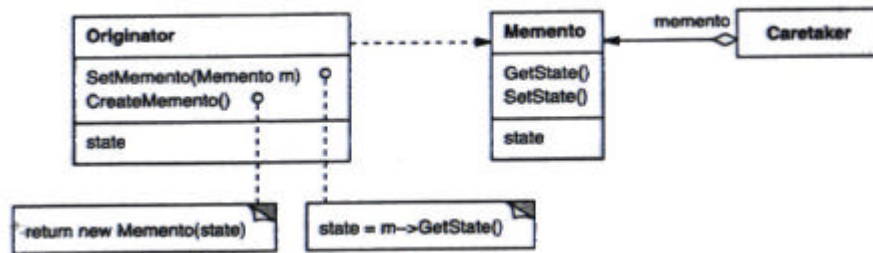
- **Applicabilità:** Il Memento può essere usato se:
  - Occorre salvare lo stato di un oggetto (o una parte di esso) in modo da poterlo ripristinare successivamente;
  - Un'interfaccia diretta per ottenere tale stato potrebbe esporre dettagli implementativi (venendo meno così al concetto di incapsulamento)



# Memento

(Object Behavioral)

## Struttura:

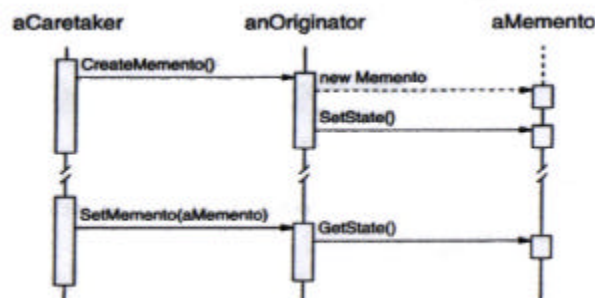


# Memento

(Object Behavioral)

## ■ Collaborations:

- Un *caretaker* richiede un *memento* da un *originator*, lo conserva, e lo ritorna successivamente allo stesso *originator*.
- Il *memento* è in effetti passivo, in quanto il suo stato è gestito dall'*originator*.





## Memento

(Object Behavioral)

### ■ Conseguenze:

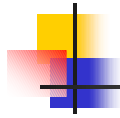
- Preservare l'incapsulamento;
- Semplificare l'*Originator*
- Utilizzare un *Memento* potrebbe tuttavia essere costoso (se copiare l'intero stato dovesse risultare dispendioso; è possibile applicare un discorso di incrementalità, operando in maniera differenziale);
- In alcuni linguaggi è difficile limitare al solo *Originator* l'accesso al *Memento*;
- Il costo del *Memento* non è noto al *caretaker* (che quindi non conosce la dimensione dello stato immagazzinato).



## Observer

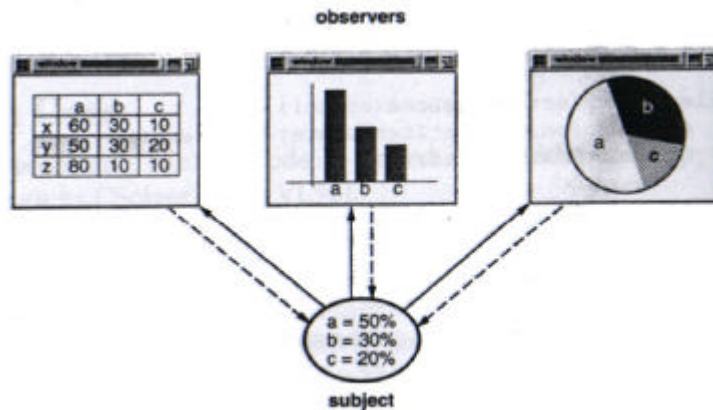
(Object Behavioral)

- **Intento:** Definire una relazione uno a molti tra oggetti, in modo che quando un oggetto cambia stato, ciò è notificato a tutti gli oggetti dipendenti
- **Nota come:** Dependant, Publish-Subscribe
- **Motivazioni:** Es. consideriamo i grafici prodotti da uno spreadsheet. Cambiando i dati di origine, i grafici si aggiornano automaticamente. L'interazione avviene tra un *subject* e un *observer*, il quale può avere un numero qualsiasi di *observer* dipendenti, ai quali giunge una notifica quando cambia lo stato di un *subject*.



## Observer

(Object Behavioral)



## Observer

(Object Behavioral)

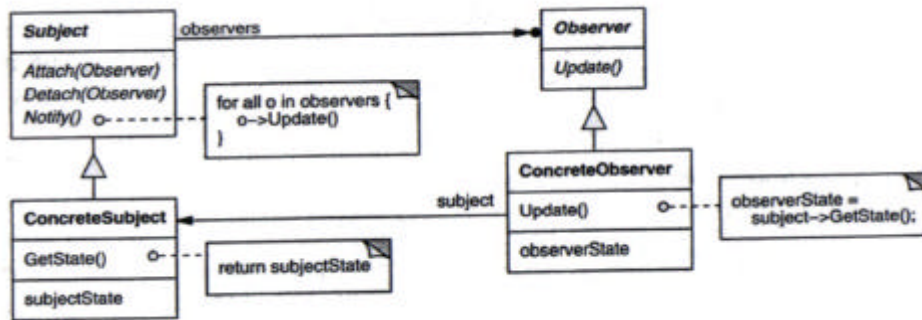
- **Applicabilità:** l'Observer andrebbe usato quando:
  - Un'astrazione ha due aspetti, una dipendente dall'altra, e si desidera incapsulare le due astrazioni in oggetti separati;
  - Se cambiamenti ad un oggetto richiedono di cambiarne altri, senza sapere quali;
  - Quando un oggetto deve essere in grado di notificare qualcosa ad un'altro oggetto non noto a priori.



# Observer

(Object Behavioral)

## Struttura:



# Observer

(Object Behavioral)

## ■ Collaborations:

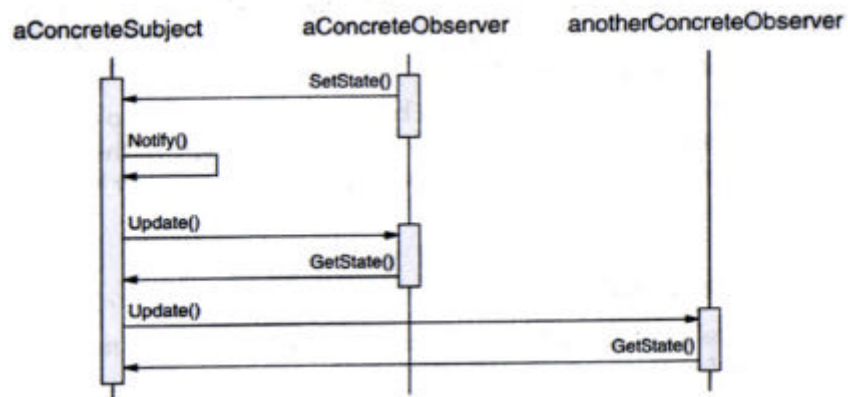
- Il *ConcreteSubject* notifica ai propri *Observers* quando ha luogo un cambiamento che porterebbe l'*Observer* in uno stato inconsistente.
- Dopo essere stato informato del cambiamento, il *ConcreteObserver* effettua una query sul *Subject* per richiedere informazioni, e le utilizza poi per riallineare il proprio stato con quello del *Subject* stesso.



# Observer

(Object Behavioral)

## Collaborations:



# Observer

(Object Behavioral)

## ■ Conseguenze:

- Accoppiamento astratto tra Subject e Observer: un Subject sa che ha una lista di Observer, conformi ad un'interfaccia astratta (AbstractObserver), ma non è a conoscenza delle classi concrete degli stessi;
- Supporto alla comunicazione di tipo broadcast;
- Aggiornamenti inattesi: se le interdipendenze non sono ben formate, possono verificarsi aggiornamenti a cascata indesiderati.



## State

(Object Behavioral)

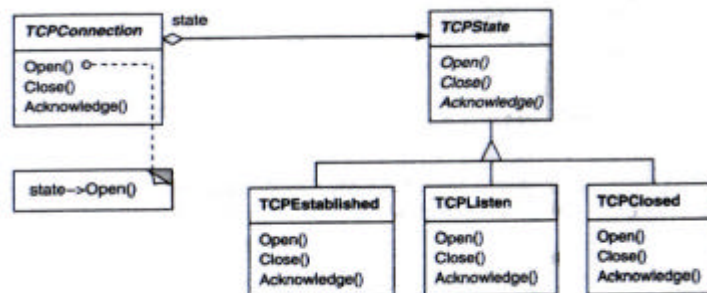
- **Intento:** Consentire ad un oggetto di modificare il proprio comportamento quando il suo stato interno cambia
- **Nota come:** Objects o States
- **Motivazioni:** Consideriamo una connessione TCP, che può trovarsi nei seguenti stati: *Established*, *Listening*, *Closed*. Quando un oggetto *TCPConnection* riceve richieste, si comporta in maniera diversa a seconda dello stato.



## State

(Object Behavioral)

L'idea è di creare una classe astratta *TCPState*, avente interfaccia comune da specializzare per i diversi stati. L'oggetto cambia ogni qualvolta cambia lo stato della connessione.



# State

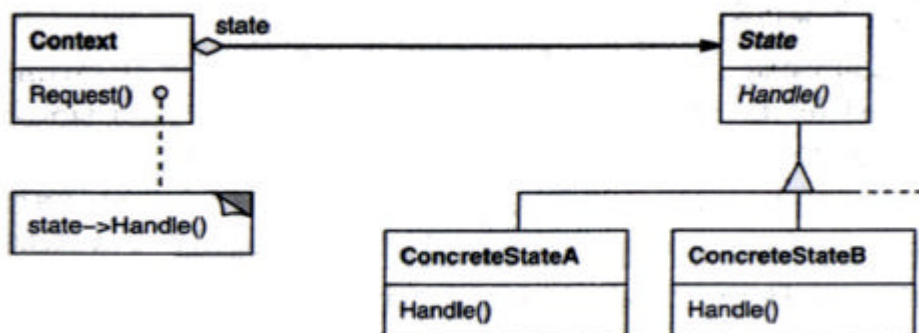
(Object Behavioral)

- **Applicabilità:** utilizzare lo State nei seguenti casi:
  - Il comportamento di un oggetto dipende dal suo stato, e deve modificare il proprio comportamento a run-time in base alla variazione dello stato;
  - Le operazioni sono implementate mediante largo uso di statement condizionali dipendenti dai valori assunti dalle variabili di stato: lo State pone ciascun branch in una classe separata.

# State

(Object Behavioral)

Struttura:





## State

(Object Behavioral)

### ■ Collaborations:

- Il *Context* delega richieste state-specific all'oggetto *ConcreteState*;
- Un *Context* può passare se stesso come argomento ad un oggetto *State* che gestisce la richiesta
- Il *Context* è l'interfaccia verso i *Client*
- Le sottoclassi di *Context* e *ConcreteState* decidono quale sarà il nuovo stato e sotto quali condizioni.



## State

(Object Behavioral)

### ■ Conseguenze:

- Partiziona i comportamenti state-specific;
- Rende esplicite le transizioni tra stati;
- Protegge il *Context* dalla possibilità di passare in stati inconsistenti;
- Possibilità di condividere (es. mediante un Flyweight) degli *State* objects.



# Strategy

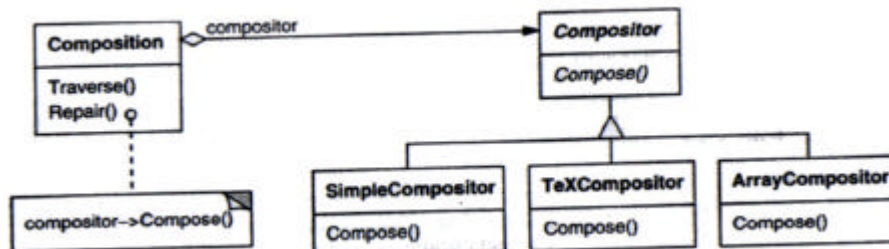
(Object Behavioral)

- **Intento:** Definire una famiglia di algoritmi, incapsularli e renderli intercambiabili in maniera trasparente rispetto all'uso da parte del client
- **Nota come:** Policy
- **Motivazioni:** supponiamo di voler suddividere in righe del testo utilizzando strategie diverse:
  - **SimpleCompositor:** una riga per volta;
  - **TeXCompositor:** un paragrafo per volta;
  - **ArrayCompositor:** ogni riga ha un numero fisso di caratteri.



# Strategy

(Object Behavioral)





# Strategy

(Object Behavioral)

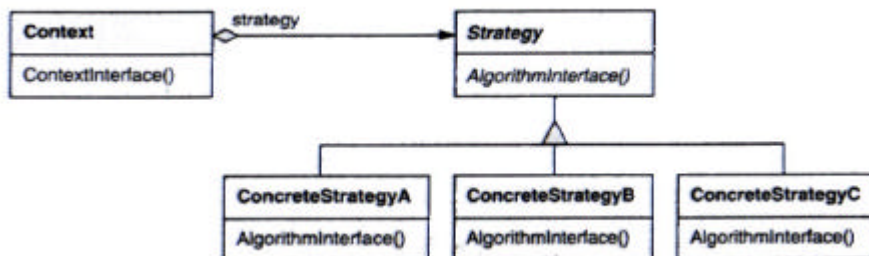
- **Applicabilità:** Lo Strategy è utile se:
  - Classi collegate differiscono solo per il loro comportamento (es. algoritmi di compressione che minimizzano lo spazio/minimizzano il tempo);
  - Gli algoritmi utilizzano dati non a conoscenza del client;
  - Una classe definisce differenti comportamenti, espressi come statement condizionali multipli nelle operazioni (il problema si risolve in maniera simile al caso del pattern *State*)



# Strategy

(Object Behavioral)

## Struttura:





## Strategy

(Object Behavioral)

### ■ Collaborations:

- *Strategy* e *Context* interagiscono per implementare l'algoritmo scelto; il *Context* può passare tutti i dati richiesti dall'algoritmo allo *Strategy* alla chiamata dell'algoritmo o, alternativamente, passare se stesso alle operazioni dello *Strategy* ;
- Un *Context* inoltra le richieste del client all'opportuno *Strategy*. Di solito i client passano un oggetto *ConcreteStrategy* al *Context*.



## Strategy

(Object Behavioral)

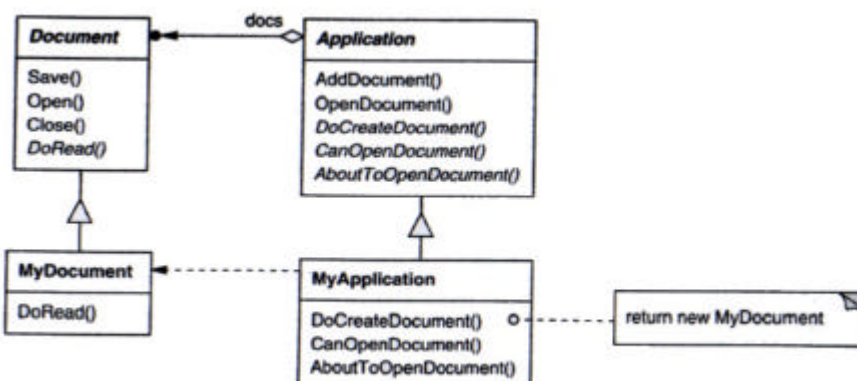
### ■ Conseguenze:

- Creazione di famiglie di algoritmi collegati;
- E' un'alternativa al subclassing;
- Eliminazione degli statement condizionali;
- Possibilità di scelta a run-time di una implementazione;
- Il client deve essere a conoscenza delle differenti strategie prima di selezionarne una;
- Possibilità di overhead di comunicazione tra *Strategy* e *Context*;
- Il numero di oggetti cresce.

## Template Method (Class Behavioral)

- **Intento:** Definire lo scheletro di un algoritmo in un'operazione, delegando alcuni substeps alle sottoclassi, le quali possono modificarli senza impattare la struttura dell'algoritmo.
- **Motivazioni:** consideriamo il framework di applicazioni che aprono documenti diversi. Un'operazione *OpenDocument* dell'*Application* può essere considerato un Template Method, che definisce un algoritmo in base ad operazioni astratte le cui sottoclassi ne definiscono il comportamento.

## Template Method (Class Behavioral)





## Template Method (Class Behavioral)

```
void Application::OpenDocument (const char* name) {  
    if (!CanOpenDocument(name)) {  
        // cannot handle this document  
        return;  
    }  
    Document* doc = DoCreateDocument();  
    if (doc) {  
        _docs->AddDocument(doc);  
        AboutToOpenDocument(doc);  
        doc->Open();  
        doc->DoRead();  
    }  
}
```

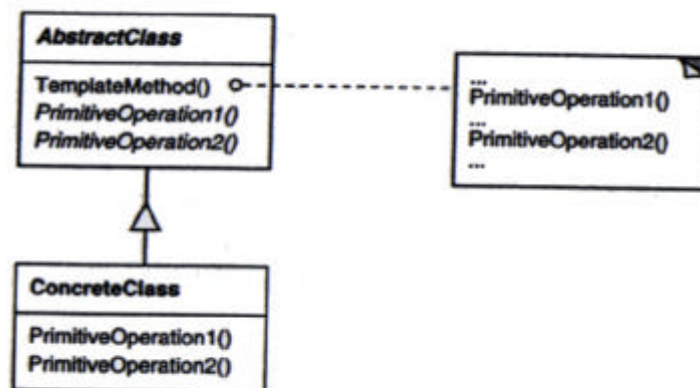


## Template Method (Class Behavioral)

- **Applicabilità:** Il template method andrebbe usato:
  - Per implementare parti invarianti di un algoritmo e lasciare alle sottoclassi l'implementazione dei comportamenti variabili;
  - Quando risulta opportuno clusterizzare comportamenti comuni tra sottoclassi per evitare duplicazione di codice;
  - Per controllare estensioni di sottoclassi.

## Template Method (Class Behavioral)

### Struttura:



## Template Method (Class Behavioral)

- **Collaborations:** La *ConcreteClass* si affida all'*AbstractClass* per implementare le parti invarianti dell'algoritmo.
- **Conseguenze:** Il Template Method è fondamentale per effettuare riuso, particolarmente nella realizzazione di librerie di classi, in quanto costituisce un metodo per fattorizzare comportamenti comuni.



## Visitor

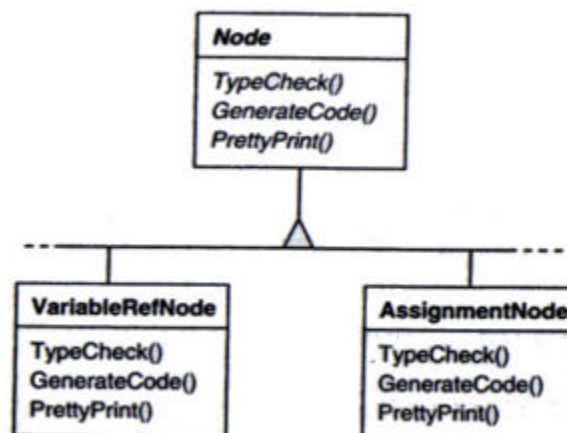
(Object Behavioral)

- **Intento:** Rappresentare un'operazione da eseguire sugli elementi di una struttura. Il Visitor consente di definire nuove operazioni senza modificare le classi degli elementi su cui operare.
- **Motivazioni:** Consideriamo la rappresentazione mediante albero sintattico di un programma. Tale rappresentazione sarà utilizzata dal compilatore per generare binario, ma può servire ad altri scopi (es. pretty-printing). E' auspicabile separare la struttura dalle operazioni da compiere su di essa.



## Visitor

(Object Behavioral)





## Visitor

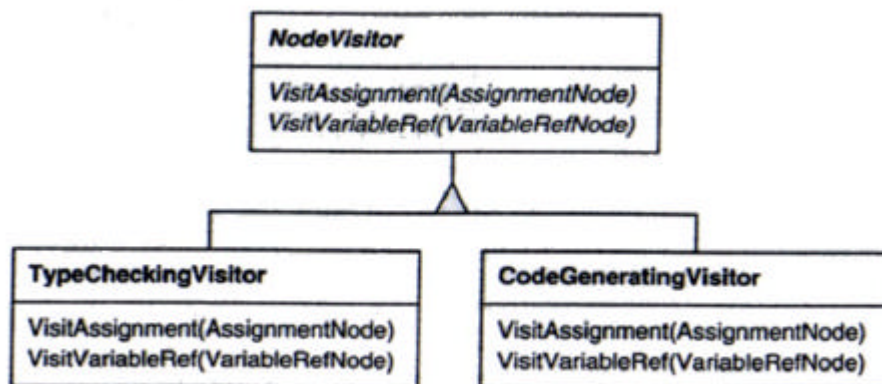
(Object Behavioral)

L'idea è di creare una gerarchia di classi separata da quella della struttura da visitare. Le istanze degli oggetti di tale gerarchia sono passate agli oggetti della struttura dati che, quindi, "**accetta**" il *visitor*. La *superclasse* *NodeVisitor* contiene operazioni astratte per ogni nodo della struttura dati. Le sottoclassi implementano le operazioni da utilizzare nei diversi casi.



## Visitor

(Object Behavioral)

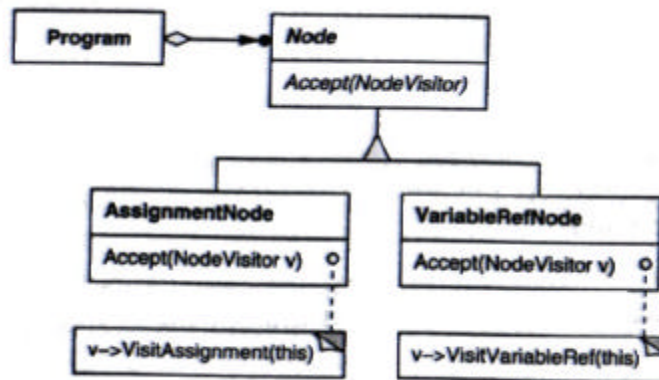




## Visitor

(Object Behavioral)

La struttura dati deve essere così modificata, in modo da poter accettare i visitor:



## Visitor

(Object Behavioral)

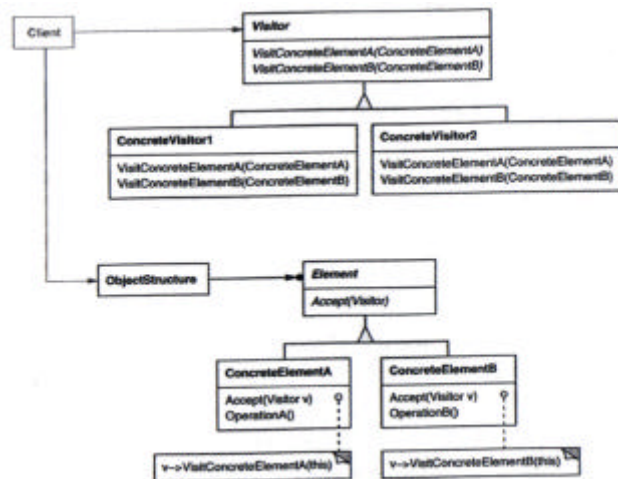
- **Applicabilità:** Il visitor andrebbe usato quando:
  - Una struttura di oggetti contiene diverse classi con diverse interfacce, e si desidera eseguire le operazioni in base alle classi concrete;
  - Occorre eseguire sulla struttura diverse operazioni distinte e scorrelate, e si desidera evitare di complicare l'interfaccia delle classi della struttura, aggiungendo le diverse operazioni;
  - **IMPORTANTE:** le classi della struttura cambiano raramente, ma occorre spesso invece definire nuove operazioni. Modifiche alla struttura richiederebbero la ridefinizione dell'interfaccia di tutti i visitor.



# Visitor

(Object Behavioral)

## Struttura:



# Visitor

(Object Behavioral)

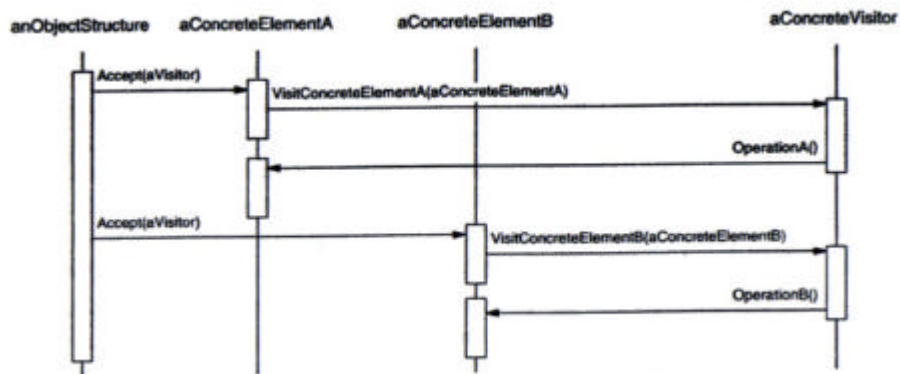
## ■ Collaborations:

- Un client che utilizza un *Visitor* deve creare un *ConcreteVisitor* e poi attraversare la struttura, visitando ciascun elemento tramite il visitor;
- Quando un elemento è visitato, esso richiama il *Visitor* che corrisponde alla propria classe, passandosi come argomento.

# Visitor

(Object Behavioral)

## ■ Collaborations:



# Visitor

(Object Behavioral)

## ■ Conseguenze:

- L'aggiunta di nuove operazioni è agevole;
- Le operazioni correlate sono messe assieme, quelle scorrelate sono separate;
- E' difficile aggiungere nuove classi di *ConcreteElement*;
- I visitor consentono di "accumulare" stato;
- Il visitor assume che il *ConcreteElement* abbia un'interfaccia in grado di consentirgli di svolgere il proprio compito. Ciò forza la pubblicazione di alcune operazioni del *ConcreteElement*, in contrasto con i meccanismi di incapsulamento.