Design pattern – VI

Stefano Mizzaro

Dipartimento di matematica e informatica
Università di Udine
http://www.dimi.uniud.it/~mizzaro
mizzaro@dimi.uniud.it

PAOO, Lezione 14
7/5/2004

Riassunto

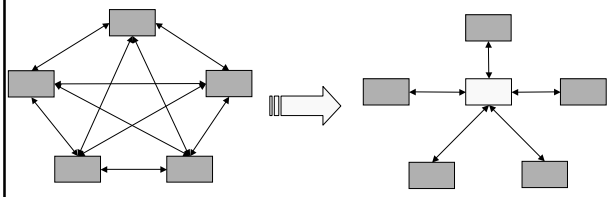
- I 7 pattern strutturali: Adapter, Façade, Composite, Decorator, Bridge, Proxy, Flyweight
- I 5 pattern creazionali: Factory Method, Abstract Factory, Singleton, Prototype, Builder
- I primi 5 pattern comportamentali: Template Method, Strategy, State, Command, Observer

Scaletta

- Gli ultimi 6 pattern comportamentali
  - Mediator
  - Memento
  - Iterator (Iteratore)
  - Visitor (Visitatore)
  - Chain of Responsibility (Catena di responsabilità)
  - Interpreter (Interprete)
- Confronti e commenti

18. Mediator (Mediatore)

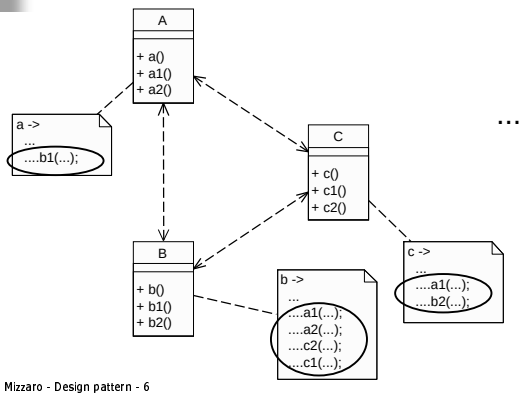
- Scopo
  - Un oggetto incapsula le modalità di interazione fra oggetti…
  - …evitando a tutti gli oggetti di "conoscere" tutti gli altri



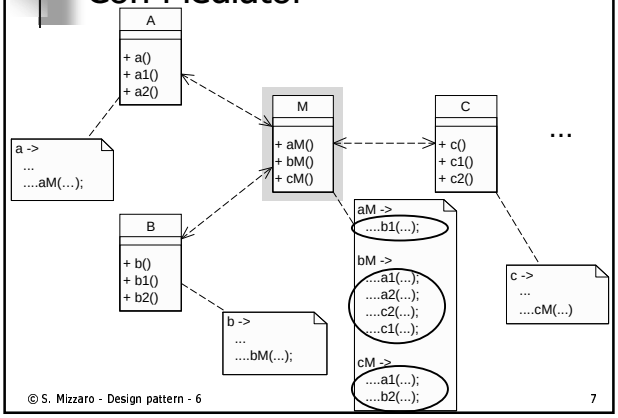
Commenti

- Senza Mediator
  - Tante classi
  - Ogni classe deve conoscere i metodi di molte altre classi
  - Alto accoppiamento
  - Meno comprensibilità, modificabilità, riuso
- Con Mediator
  - Si abbassa l'accoppiamento
  - Riuso…
    - Classi piccole sono più facilmente riusabili…
    - … ma se devono comunicare con molte altre…
  - Consente di variare le modalità di interazione

Senza Mediator



Con Mediator



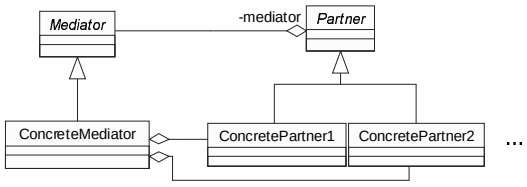
Commenti (1/2)

- Diminuisce l'accoppiamento **A** – **B** – **C**
- + **D**, **E**, **F**, **G**, ...!!
  - Da O(N^2) a O(N)
- È più facile, ad es.:
  - Specializzare una modalità di comunicazione: sottoclasse di **M** e basta
  - sostituire **A** con **A1** e **A2**

Commenti (2/2)

- Il mediatore rappresenta la comunicazione
  - La comunicazione come oggetto separato
  - Posso "ragionarci su" (ad es., subclassing)
- Contro: Mediator può diventare un oggetto complesso, monolitico, di difficile manutenzione
- Esempio
  - GUI con parti interdipendenti

Struttura Mediator



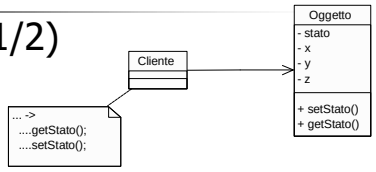
Legami con altri pattern

- Façade
  - Anche lì un oggetto "nasconde" una situazione complicata. Però:
  - Façade: comunicazione unidirezionale
  - Mediator: comunicazione bidirezionale
- Observer
  - Mediator potrebbe essere un Observer,
    - che "propaga" i cambiamenti di stato
  - e i vari partner gli oggetti osservati, che
    - si registrano
    - notificano il mediatore quando cambiano stato

19. Memento (Memento, Ricordo)

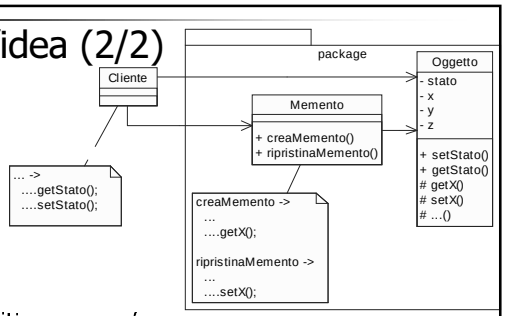
- Scopo
  - Catturare lo stato di un oggetto, in modo da poterlo ripristinare in seguito, senza violare l'incapsulamento
  - Utile, ad es., per undo
    - Anche multiplo, stack di mementi…
- Vediamolo di corsa
  - … pattern un po' "strano"

L'idea (1/2)



- **`stato`** è lo stato "visibile" all'esterno
- **`x`**, **`y`** e **`z`** sono incapsulati
  - ma servono per salvare "tutto" l'oggetto
- L'aggiunta di **`getX/Y/Z`** e **`setX/Y/Z`**
  - aumenterebbe l'accoppiamento
  - violerebbe l'incapsulamento
    - interfaccia a stati illegali

L'idea (2/2)



- Così limitiamo un po'...
  - **`Cliente`** non viola l'incapsulamento
  - **`Cliente`** non può accedere agli attributi privati, ma può salvarne/ripristinarne il valore in un memento

Commenti

- Mah...
- Non è detto che possiamo creare un package...
- Memento potrebbe essere una classe interna
- Confronto
  - Protoype è creazionale, e crea tutta una nuova istanza (copia di un'altra). In Java, **`Object.clone`**
  - Mediator è comportamentale e "salva" solo ciò che serve per ripristinare

20. Iterator (Iteratore)

- Scopo
  - Fornire un modo di accesso sequenziale agli elementi di un oggetto composto...
  - ...senza esporre la struttura dell'oggetto composto
- Es.:
  - Scorrere una lista

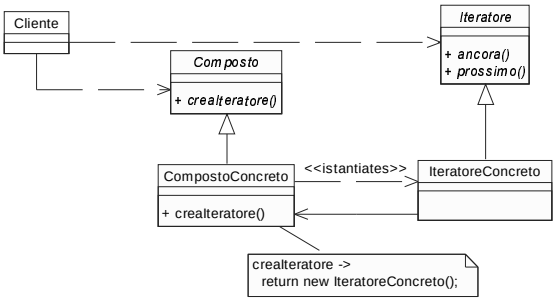
Senza Iterator

- Dovrei mettere la responsabilità dell'accesso nella classe dell'oggetto composto
- E potrei aver bisogno di due attraversamenti
  - Es.: Scorrere gli elementi pari di una lista
  - Magari contemporanei
- L'oggetto composto avrebbe un'interfaccia sovraccarica ("fat interface")

Con Iterator

- Responsabilità di accesso e attraversamento di una lista al di fuori, in una classe Iteratore separata
- La classe Iteratore ha le responsabilità di:
  - Restituire l'elemento "successivo"
  - Mantenere traccia dell'elemento visitato
- La classe dell'oggetto composto ha solo la responsabilità di creare l'iteratore opportuno
  - Si fa con un Factory Method
- Attraversamenti contemporanei: no problem

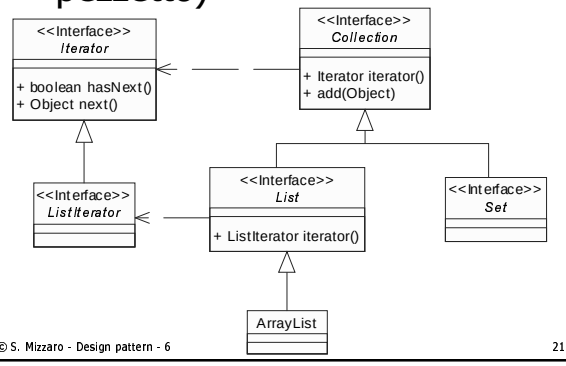
Diagramma Iterator



In Java

- Iterator c'è già
- Interfacce **`Collection`**, **`List`**, **`Set`**
  - **`List`** = collezione ordinata
  - **`Set`** = collezione non ordinata senza duplicati
- Interfaccia **`java.util.Iterator`**
  - Sottointerfaccia **`java.util.ListIterator`**
- **`iterator()`** di **`Collection`** è un Factory Method che restituisce l'iteratore opportuno
- (e poi c'è **`Enumeration`**, deprecated)

Le "collections" di Java (un pezzetto)



Legami con altri pattern

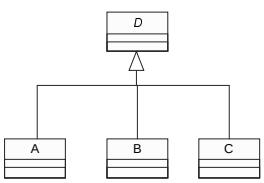
- Factory Method
  - Per generare l'iteratore opportuno
- Iterator e Composite
  - Uso tipico
  - Un po' complicato, deve tenere traccia della posizione su un albero
  - Comodo per implementare visite ≠ (breadth/depth-first, ecc.)

21. Visitor (Visitatore)

- Scopo
  - Permettere di definire una nuova operazione per le classi di una gerarchia senza modificare le classi stesse
- Basato sulla metafora del visitatore
  - Visitatore chiede di essere accettato
  - "Padrone di casa" accetta e ne invoca un metodo

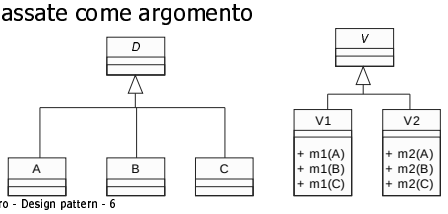
Aggiunta operazioni a gerarchia

- Una gerarchia
  - Comodo aggiungere classi
  - Scomodo aggiungere operazioni a tutte le classi
    - Aggiungere **`m1`**
    - Poi aggiungere **`m2`**
    - …
- C'è un'alternativa…



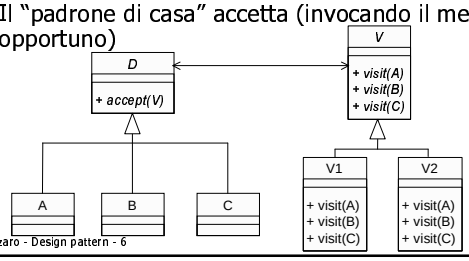
Creare una gerarchia parallela

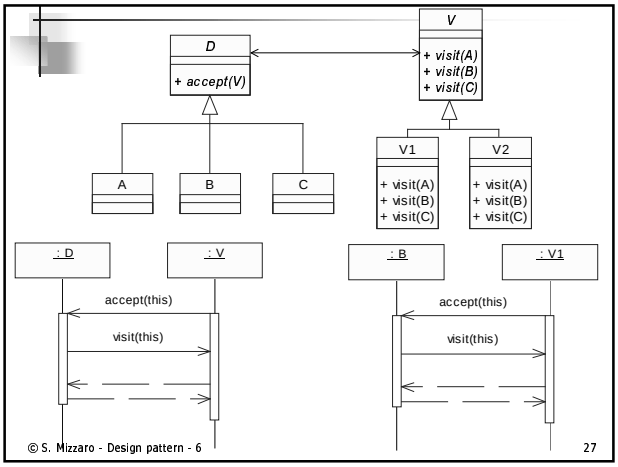
- Una sottoclasse per ogni operazione da aggiungere
- Ogni sottoclasse contiene l'implementazione dell'operazione per tutte le sottoclassi della gerarchia originale
  - Passate come argomento



Il Visitor

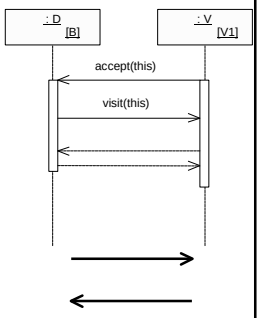
- Ancora un passo: metafora del visitatore
  - Istanza della gerarchia a dx: visitatore
  - Istanza della gerarchia a sx: "padrone di casa"
  - Il visitatore chiede di essere ospitato/accettato
  - Il "padrone di casa" accetta (invocando il metodo opportuno)





Double dispatching

- "Doppio inoltro"
  - **`V`** invoca **`accept`** di **`D`**
  - **`D`** esegue **`accept`**, cioè invoca **`visit`** di **`V`**
  - **`this`** come parametro per scelta polimorfica del codice da eseguire
    - **`D`** (o sottoclasse) potrebbe "non accettare"
    - Il metodo **`visit`** di **`V`** (o sottoclasse) viene scelto sulla base di **`this`**



Commenti

- Visitor è un pattern controverso
- Estendere il comportamento di una gerarchia ne richiede una conoscenza esperta
- Alto accoppiamento fra i visitatori e la gerarchia
- Le alternative sono spesso più robuste

22. Chain of Responsibility (Catena di responsabilità)

- Scopo
  - Evita l'accoppiamento fra mittente di una richiesta e destinatario
- Più oggetti possono soddisfare la richiesta
  - Gli oggetti sono concatenati
  - si passano la richiesta lungo la catena
  - finché un oggetto riesce a soddisfarla

Diagramma Chain of Responsibility

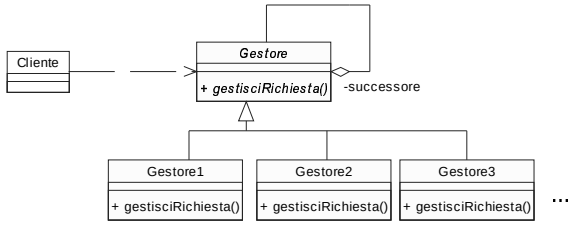
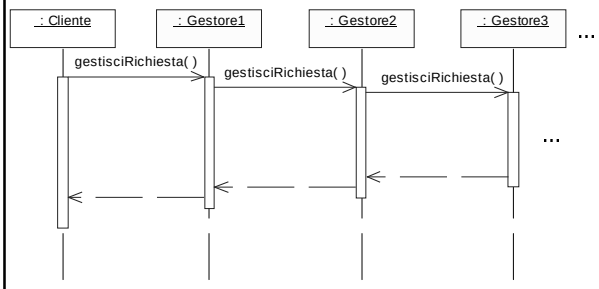


Diagramma di sequenza



Commenti: l'interfaccia comune

- Al solito, il cliente "parla alla classe (astratta) base"
- Il cliente non sa chi gestisce la sua richiesta
- Interfaccia comune per gestione e/o passaggio della richiesta
  - `+gestisciRichiesta()`
  - `-successore`
  - Garantisce che l'oggetto che soddisfa la richiesta rimanga implicito

Commenti: la catena

- Se la richiesta arriva a fine catena, non è gestita
- È possibile cambiare dinamicamente la "catena"
  - Cambiando l'ordine
  - Aggiungendo/eliminando elementi
- La catena potrebbe già esserci (ad es., Composite: uso tipico)
- Eventualmente `java.util.List`

Commenti: la richiesta

- Rappresentazione della richiesta
  - Invocazione di un metodo
    - (eventualmente statica)
  - Parametro con codice richiesta
  - Parametro istanza di una classe `Richiesta` (eventualmente usando Command)

Esempio

- Help contestuale in GUI: pulsante "`?`"
- `Button` contenuto in un `Panel` contenuto in un altro `Panel` contenuto in un `Frame`
  - "`?`" sul Button → richiesta di help al `Button` → richiesta di help al `Panel1` → richiesta di help al `Panel2` → richiesta di help al `Frame`
  - Il primo che riesce a (può) gestirla, lo fa

23. Interpreter (Interprete)

- Scopo
  - Dato un linguaggio
  - definisce una rappresentazione per la sua grammatica
  - e un interprete che usa la rappresentazione per interpretare le frasi del linguaggio
- "Come fare un interprete per un linguaggio semplice"

Esempio

- Grammatica per espressioni booleane

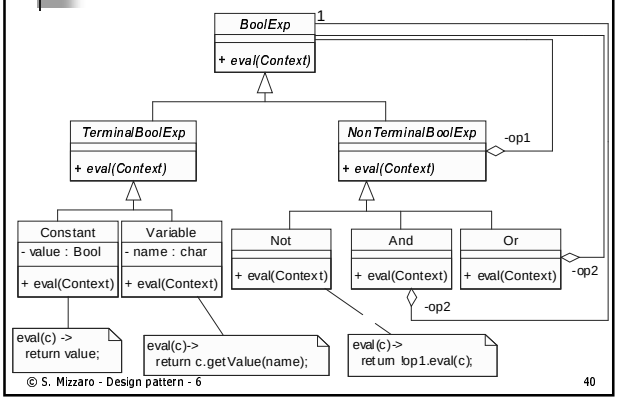
```
BooleanExp ::= VariableExp | Constant | OrExp |
               AndExp | NotExp | '(' BooleanExp ')'
AndExp ::= BooleanExp 'and' BooleanExp
OrExp ::= BooleanExp 'or' BooleanExp
NotExp ::= 'not' BooleanExp
Constant ::= 'true' | 'false'
VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'
```

```
(A and true) or (B or C)
A and (B or C)
...
```


Come fare l'interprete

- Una classe per produzione
- Uso naturale del Composite
- Sottoclassi per terminali e non terminali
- Per valutare:
  - Un metodo **`eval()`** in tutte le classi
  - Una classe **`Context`** che contiene i valori delle variabili

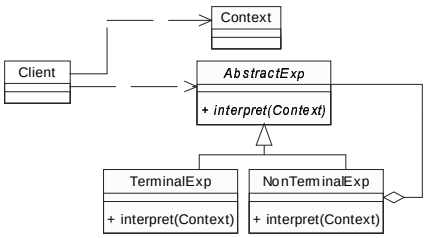
Diagramma



Il pattern Interpreter

- Una classe per ogni produzione
- Con un metodo **`interpret()`** in ogni classe
  - Tipicamente definito ricorsivamente sui simboli non terminali
- Oltre a **`interpret()`** ci possono essere altri metodi
  - Ad es., in un interprete per linguaggio di programmazione: **`typeCheck()`**, ...
- Ok per grammatiche semplici
- N.B. Presuppone l'albero sintattico

Diagramma Interpreter



Pattern correlati

- Composite
  - L'albero sintattico è un Composite
- Flyweight
  - Per condividere i simboli terminali
- Iterator
  - Si può usare per attraversare la struttura
- Visitor
  - `interpret()` e altri metodi possono essere in una gerarchia Visitor separata

Gli 11 pattern comportamentali

13. Template Method (Metodo sagoma)
14. Strategy (Strategia)
15. State (Stato)
16. Command (Comando)
17. Observer (Osservatore)
18. Mediator (Mediatore)
19. Memento (Memento, Ricordo)
20. Iterator (Iteratore)
21. Visitor (Visitatore)
22. Chain of Responsibility (Catena di responsabilità)
23. Interpreter (Interprete)

Pattern comportamentali: temi comuni

- Dal [GoF]
- Spostare responsabilità al di fuori di una classe
  - Ereditarietà
    - Template Method
  - Delega
    - Strategy, State, Command, Mediator, Memento, Iterator, Visitor

Pattern comportamentali: temi comuni

- Incapsulare gli aspetti di un programma che probabilmente cambieranno
  - Strategy → algoritmo
  - State → comportamento dipendente dallo stato
  - Mediator → protocollo di collaborazione
  - Iterator → modalità d'accesso ad aggregato
- Oggetti passati come parametri
  - Visitor: `this` parametro di `accept()` e `visit()`
  - Command
  - Memento
  - ...

Pattern comportamentali: temi comuni

- Disaccoppiare mittenti e destinatari
  - Command, Observer, Mediator, Chain of Responsibility
- Comunicazione centralizzata vs. distribuita
  - Mediator incapsula
  - Observer distribuisce

Classificazione dei pattern [GoF]

Strutturali	Creazionali	Comportamentali
1. Adapter (Adattatore)	8. Factory Method (Metodo fabbrica)	13. Template Method (Metodo sagoma)
2. Façade (Facciata)	9. Abstract Factory (Fabbrica astratta)	14. Strategy (Strategia)
3. Composite (Composto)	10. Singleton (Singoletto)	15. State (Stato)
4. Decorator (Decoratore)	11. Prototype (Prototipo)	16. Command (Comando)
5. Bridge (Ponte)	12. Builder (Costruttore)	17. Observer (Osservatore)
6. Proxy (Proxy)		18. Mediator (Mediatore)
7. Flyweight (Peso piuma)		19. Memento (Memento, Ricordo)
		20. Iterator (Iteratore)
		21. Visitor (Visitatore)
		22. Chain of Responsibility (Catena di responsabilità)
		23. Interpreter (Interprete)

Design pattern: temi comuni

- Uso polimorfismo invece di switch o if
- Parlare alla classe base
  - Disaccoppia dalla specifica classe concreta
- Delega/composizione invece di ereditarietà
  - Estrarre qcosa in un'altra classe
- Incapsulare ciò che varia, ad. es.
  - un algoritmo in uno Strategy (e se un domani l'alg. varia, no problem)
  - un processo di creazione in un pattern creazionale
  - varie tipologie di gruppi in un Composite
- Nuovi requisiti, cambiamenti senza riprogettare

Cosa aspettarsi dai pattern [1/2]

- Programmazione OO
  - Esempi, soluzioni, … "pattern" ben fatti,
  - A più alto livello degli elementi di base OO (oggetti, disaccoppiamento, ereditarietà, polimorfismo, …)
- Progetto nuovi sistemi
  - Concetti/soluzioni utili per passare da modello di analisi a modello d'implementazione
  - Vocabolario comune (comunicazione, documentazione)
- Comprensione software esistente
  - Modelli utili
- Reingegnerizzazione (refactoring)
  - Obiettivi da raggiungere (vedremo)

Cosa aspettarsi dai pattern [2/2]

- [GoF] è vecchio, per C++, pre-Java
  - Es.: Façade in Java non serve, ci sono i package e classi con visibilità di package [E]
- Abbiamo visto solo un'introduzione superficiale…
- … e i design pattern si imparano solo usandoli…
- … e questo spetta a voi…
- Vedremo esempi d'uso

Riassunto

- Gli ultimi pattern comportamentali
  - Mediator
  - Memento
  - Iterator (Iteratore)
  - Visitor (Visitatore)
  - Chain of Responsibility (Catena di responsabilità)
  - Interpreter (Interprete)
- Confronti e commenti (veloci)

Bibliografia [1/2]

- [GoF] E. Gamma, R. Helm, R. Johnson, J. Vlissides (GoF, Gang of Four), *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994 (anche in Italiano)
- [ST] A. Shalloway & J. R. Trott, *Design Patterns Explained*, Addison Wesley, 2002
- [M] S. J. Metsker, *Design Patterns Java Workbook*, Addison wesley, 2002 (anche in italiano)

Bibliografia [2/2]

- [E] B. Eckel, *Thinking in Patterns*, www.mindview.net, 2003
- [C] J. Cooper, *The design patterns Java Companion* http://www.patterndepot.com/put/8/JavaPatterns.htm
- http://www.netobjectives.com/ [ST]
- PLoP: Pattern Languages of Programs Conference http://www.hillside.net/conferences/plop.htm
- http://www.hillside.net/