Design pattern – I

Stefano Mizzaro

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

PAOO, Lezione 9
16/4/2004

Riassunto (1/4)

- Introduzione a OO e UML
- I principi della progettazione OO
  - Livelli di incapsulamento
  - Dipendenze
  - Domini
  - Ingombro
  - Legge di Demeter
  - Coesione
  - Spazio degli stati (ed eredità)
  - Transizioni e comportamento (ed eredità)

Riassunto (2/4)

- Asserzioni
- Invarianti di classe
- Precondizioni e postcondizioni delle operazioni (metodi)
- Progetto per contratto
- inv, pre, post ed eredità:
  - Principio di sostituibilità (istanza di sottoclasse dove ci si aspetta istanza della sopraclasse)
    - Conformità di tipo
    - Comportamento chiuso

Riassunto (3/4)

- Per avere la sostituibilità:
- Conformità di tipo
  - inv sottoclasse più forte inv sopraclasse (SdS sottoclasse ha più vincoli sulle dim. della sopraclasse e vincoli nuovi sulle nuove dim.)
  - pre nel metodo della sottoclasse più debole (chiedere di meno, controvarianza)
  - post nel metodo della sottoclasse più forte (garantire di più, covarianza)
- Comportamento chiuso
  - Metodi ereditati dalla sopraclasse devono rispettare l'inv della sottoclasse

Riassunto (4/4)

- I pericoli di ereditarietà e polimorfismo
  - Gerarchie errate
  - Gestione del polimorfismo
- Genericità
- Ancora sull'interfaccia di una classe
  - Anelli di operazioni
  - Tipologie di stati e di comportamento
- Ancora sulla coesione, di operazione/metodo

Dove siamo

- 1/3 del corso
  - Introduzione
  - Criteri per buon OOD
- Prossimi 2/3
  - Design pattern
  - Refactoring
  - OOA (use case, pattern di analisi)
  - "Varie" (agenti, casi di studio, …)

4 Seminari

- JUnit (Mauro Lorenzutti)
- J2EE (Emanuele Rosso + Adolfo Bulfoni)
  - Tomcat, servlet, JSP, UMLxWeb(?)
- eXtreme Programming (Gianluca Demartini)

Scaletta

- Pattern di progetto
  - Cosa sono
  - Perché studiarli
  - Classificazione
- Catalogo dei pattern (oggi i primi 4…)

- Classificazioni dei pattern (di nuovo)
- Cos'è un pattern di progetto (di nuovo)
- Confronti fra pattern

Bibliografia

[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)

[C] J. Cooper, *The design patterns Java Companion*, http://www.patterndepot.com/put/8/JavaPatterns.htm

[E] B. Eckel, *Thinking in Patterns*, www.mindview.net, 2003

Definizione di pattern di progetto

- Schema di soluzione tipica, ricorrente a certi problemi
- Esempi non OO:
  - Due cicli annidati
  - Ricorsione di coda
  - …
- …Lo capiremo meglio alla fine…

Perché studiare i pattern?

- Didatticamente:
  - Esempi di buon software OO
  - Pattern : Programmazione OO = Strutture controllo : Programmazione strutturata
  - "I pattern sono l'equivalente della programmazione strutturata nel caso OO"
- Professionalmente
  - Utili per pensare a un livello più alto
  - Vocabolario, terminologia

Come studiare i pattern?

- Eh, si imparano solo dopo averli usati più volte
- Bisogna pur cominciare…
- Li vediamo tutti, bene o male

Tuttiipattern!

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

Tre categorie

- Strutturali
  - Strutture tipiche
- Comportamentali
  - Gestione di comportamenti
- Creazionali
  - Come creare oggetti in modo controllato
- Discusse, esistono alternative
- …Lo capiremo meglio alla fine…

Classificazione dei pattern

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

1. Adapter (Adattatore)

- Scopo (Intent): Adattare l'interfaccia di una classe già pronta all'interfaccia che il cliente si aspetta
- ("Adapter" da "riduttore" per prese di corrente)

Scopo

- Dati:
  - Un cliente che si aspetta
    - una classe/oggetto che fornisce certi servizi
    - con una certa interfaccia (nome di classe, metodi e loro segnature)
  - Una classe che
    - fornirebbe quei servizi
    - ma ha un'interfaccia diversa da quella attesa
- Adapter adatta l'interfaccia diversa a quella attesa

Esempio (1/3)

- Solita gerarchia…

Esempio (2/3)

- Vogliamo aggiungere **`Cerchio`**
  - Lo implementiamo da zero?
    - Fatica inutile…
  - Usiamo la classe `Circle`?
    - Ha un'interfaccia diversa…
- Creiamo un adattatore
- (Nota: non possiamo modificare `Circle`…)
  - Non abbiamo il sorgente
  - È usato così com'è
  - …

Esempio (3/3)

Considerazioni

- **`Circle`** non potrebbe essere una sottoclasse di **`Figura`**
- **`Cerchio`** deve avere una certa interfaccia (in questo caso "imposta" dall'ereditarietà da **`Figura`**)
- **`Cerchio`** adatta **`Circle`** all'interfaccia attesa

Due tipi di Adapter

- Oggetto adattatore (Object Adapter)
  - Basato su delega/composizione
- Classe adattatore (Class Adapter)
  - Basato su ereditarietà
  - L'adattatore eredita sia dall'interfaccia attesa sia dalla classe adattata
  - No eredità multipla ⇒ l'interfaccia attesa deve essere un'interfaccia, non una classe
- (Due pattern?)

Diagramma Class Adapter

Diagramma Object Adapter

- Nota: Object Adapter non eredita…

© Stefano Mizzaro

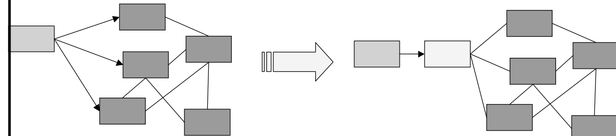
Adapter: riassunto e commenti

- Si usa quando si vuole riusare una classe esistente, magari progettata per il riuso, ma con interfaccia "sbagliata" (diversa)
- Quindi, d'ora in poi, possiamo fregarcene se una classe ha un'interfaccia "sbagliata": ci mettiamo un Adapter e via!
- `Adapter` potrebbe anche modificare leggermente il comportamento di `Adattato` (ma modifiche "pesanti" → altri pattern)
- Un object adapter può adattare più classi…

© S. Mizzaro - Design pattern - 1 25

2. Façade (facciata)

- Scopo
  - Rendere più semplice l'uso di un sistema
  - Fornire un'unica interfaccia per un insieme di funzionalità "sparse" su più interfacce/classi



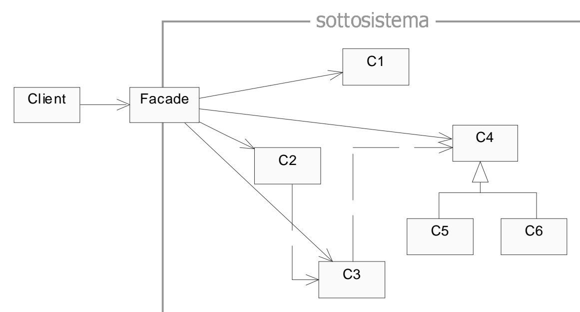
© S. Mizzaro - Design pattern - 1 26

Esempi

- Classi per disegno 3D per disegnare in 2D
- Compilatore
  - Classi `Parser`, `Scanner`, `Token`, `SyntacticTree`, `CodeGenerator`, ecc.… vs.
  - Classe `Compiler` con metodo `compile()`
- Manuale di installazione rapida della stampante
  - Di solito funziona
  - Se non va c'è sempre il manuale dettagliato…

© S. Mizzaro - Design pattern - 1 27

Diagramma Façade



© S. Mizzaro - Design pattern - 1 28

Vantaggi Façade

- Promuove un accoppiamento debole fra cliente e sottosistema
- Nasconde al cliente le componenti del sottosistema
- Il cliente può comunque, se necessario, usare direttamente le classi del sottosistema

© S. Mizzaro - Design pattern - 1 29

Commenti

- È un pattern facile
- Spesso ha solo metodi statici
- Façade, utilità e demo
  - Façade
  - Utilità (utility) in UML
    - Classe con solo metodi statici
  - Demo
    - Classe che mostra come usare una classe o un (sotto)sistema
    - Di solito è un'applicazione (giocattolo) autonoma

© S. Mizzaro - Design pattern - 1 30

Façade vs. Adapter

- Entrambi sono "wrapper" (involucri)
- Entrambi si basano su un'interfaccia, ma:
  - Façade la semplifica
  - Adapter la converte

3. Composite (Composto)

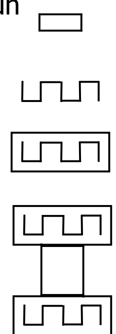
- Scopo
  - Comporre oggetti in strutture ricorsive ad albero per rappresentare gerarchie parte-tutto
  - e consentire ai clienti di trattare in modo uniforme oggetti singoli/semplici e composizioni di oggetti
- È un bel pattern

Esempio (1/5)

- Programma di grafica
- Deve consentire di
  - Creare oggetti semplici (Punto, Linea, Cerchio,…)
  - Raggruppare dinamicamente oggetti semplici in oggetti composti (4 linee in un rettangolo, …)
  - Trattare gli oggetti composti come se fossero oggetti semplici

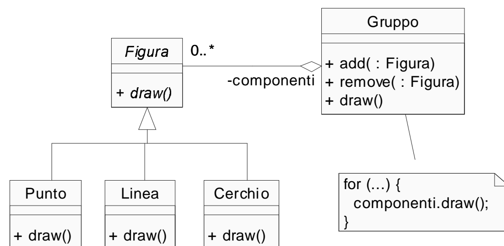
Esempio (2/5)

- Raggruppare quattro linee per fare un rettangolo
- Raggruppare N linee per fare una serpentina
- Raggruppare un rettangolo e una serpentina…
- …
- …Proviamo a fare un diagramma UML…



Esempio (3/5)

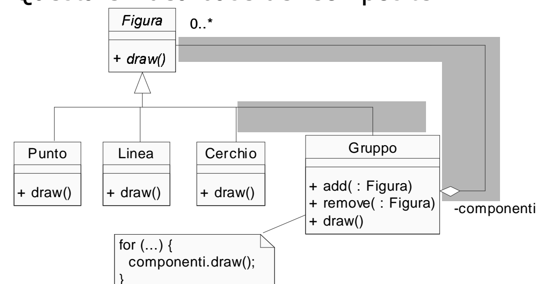
- 1o tentativo: cos'è che non va?



- Risp.: gestione **`Gruppo`** ≠ gestione **`Figura`**…

Esempio (4/5)

- 2o tentativo: **`Gruppo`** sottoclasse di **`Figura`**!
- Questa è l'idea base del Composite

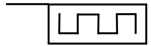


Esempio (5/5)

- Un punto/cerchio/linea è una figura…
- …"Trattare gli oggetti composti come se fossero oggetti semplici"…
- Un gruppo è una figura!
- La ricorsione è in:
  - `Gruppo` contiene `Figura`,
  - e siccome alcune figure sono gruppi (`Gruppo` sottoclasse di `Figura`)
  - un gruppo contiene gruppi…

Diagramma oggetti: albero!

- Per rappresentare il gruppo rettangolo+serpentina+linea:
  - 3 istanze di `Gruppo`
  - 14 istanze di `Linea`



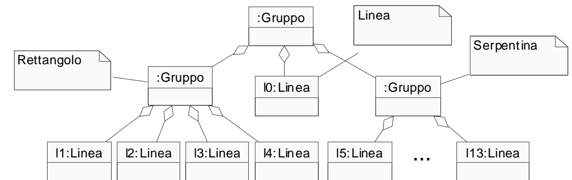
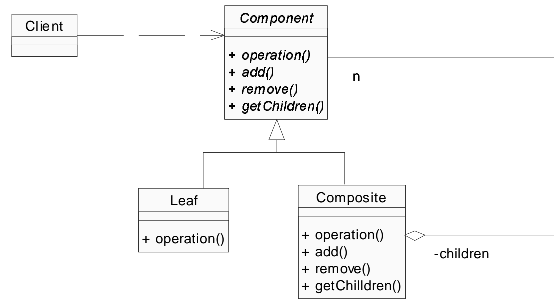


Diagramma Composite



Commenti

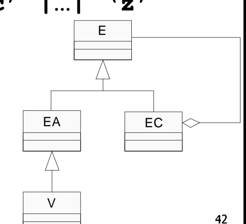
- Semplifica il cliente, che tratta strutture composte e singoli oggetti in modo uniforme
  - In tutti i punti in cui il cliente si aspetta un oggetto primitivo, gli si può passare un oggetto composto
- Interfaccia di `Component`
  - Massimizzazione → flessibilità: il `Client` non distingue oggetti atomici e composti. Però, che implementazione per `add` su una foglia?
  - Minimizzazione → sicurezza: no problem per `add` sulle foglie, però interfacce diverse…

Commenti

- `Leaf` e `Composite` possono:
  - essere astratte
  - avere sottoclassi (ed è facile aggiungerne…)
- Cicli (contenimento di se stesso, mutuo contenimento) possono causare ricorsione ∞ nelle visite
  - Memorizzare i nodi visitati…
- Usato in `java.awt`: `Component` e `Container`
- Può rendere il progetto troppo generico

Commenti: non solo grafica

- Rappresentare espressioni con alberi sintattici
- `E ::= EC | V`
  `EC ::= '(' E '+' E ')'`
  `V ::= 'a' | 'b' | 'c' |…| 'z'`
- `((a + b) + (c + d))`
- Usato in JUnit, vedremo



4. Decorator (Decoratore)

- Scopo
  - Aggiungere dinamicamente a un oggetto responsabilità/funzionalità/operazioni
- Alternativa (più flessibile) alla creazione di sottoclassi per l'estensione di funzionalità
  - (anche la specializzazione può aggiungere responsabilità/funzionalità/operazioni)

Esempio

- Come ottenerlo?

Aggiungere responsabilità staticamente?

- Ereditarietà
  - Creo istanze della classe voluta
  - Contro
    - **`AreaDiTestoConBordoEBarraScorrimento`**? **`AreaDiTestoConBarraScorrimentoEBordo`**? Altre 2 sottoclassi?
    - Statico: non posso aggiungere funzionalità dopo la creazione
    - Altre componenti? (**`Finestra`**) Duplicazione!

Aggiungere responsabilità dinamicamente

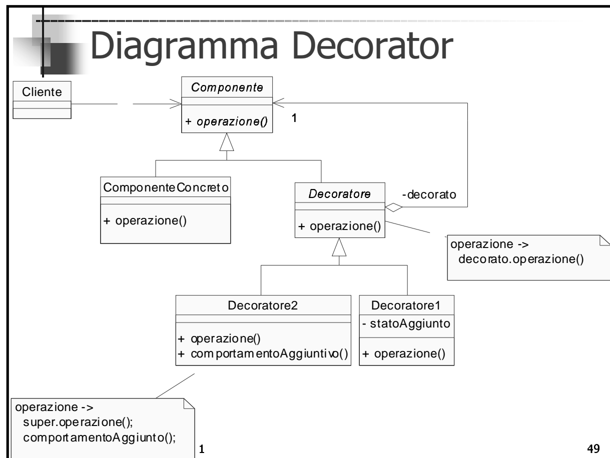
- Racchiudere l'oggetto da "decorare" (l'area di testo) in un altro oggetto, responsabile di gestire la "decorazione" (il bordo, la scrollbar): il "decoratore"
- Il decoratore trasferisce/delega e fa qualcosa in più (prima o dopo)
- Il decoratore ha l'interfaccia conforme all'oggetto decorato, e quindi è trasparente al cliente

Di nuovo l'esempio

Diagramma degli oggetti

- Catena con
  - Varie decorazioni
  - Alla fine, un'istanza di **`AreaDiTesto`**
  - ```
    new DecoratoreBordo(
      new DecoratoreBarraScorrimento(
        new AreaDiTesto()))
    ```
  - ```
    new DecoratoreBarraScorrimento(
      new DecoratoreBordo(
        new AreaDiTesto()))
    ```

Diagramma Decorator



1 49

Decoratore vs. ereditarietà

Dinamico (run time)	Statico (compile time)
Senza vincoli per il cliente (può inventarsi combinazioni non previste)	Con vincoli per il cliente (non può inventarsi combinazioni non previste)
Aggiunge responsabilità a un singolo oggetto	Aggiunge responsabilità a (tutte le istanze di) una classe
Evita esplosione combinatoria	Può causare esplosione combinatoria

© S. Mizzaro - Design pattern - 1 50

Usi tipici del Decorator

- Stream Java
- Interfacce utente grafiche
  - Componente grafica
    - area di testo, finestra, panel, …
  - decorata con
    - bordo, barra di scorrimento, …
    - (eventualmente più di uno!)
  - Non usata nelle Swing di Java
    - `JComponent` ha `setBorder`

© S. Mizzaro - Design pattern - 1 51

Decorator vs. Composite

- Un Decoratore può essere visto come un Composite degenere con un singolo componente
  - (cfr. la molteplicità e il diagramma degli oggetti)
- Però:
  - Un Composite ha lo scopo di aggregare oggetti
  - Un Decoratore ha lo scopo di aggiungere responsabilità/funzionalità a un oggetto
- Spesso usati insieme (?)

© S. Mizzaro - Design pattern - 1 52

Decorator vs. Adapter

- Decorator
  - Arricchisce di funzionalità senza modificare l'interfaccia
  - modifica le responsabilità di un oggetto, non la sua interfaccia
- Adapter modifica l'interfaccia di un oggetto
  - (può anche arricchire, aggiungere responsabilità/funzionalità, ma in modo limitato)

© S. Mizzaro - Design pattern - 1 53

Esercizio

- Chiosco Da Meni
- Vende "cibarie"
  - Pizze, calzoni, toast, tramezzini, piadine, panini, focacce, …
- Con ingredienti a scelta fra:
  - Prosciutto, formaggio, mozzarella, funghi, spinaci, speck, gorgonzola, nutella, marmellata, …
- Eventualmente "doppi" o "tripli" o …
- Non modellate tutta la gestione, solo le classi che consentano la creazione di cibarie

© S. Mizzaro - Design pattern - 1 54

Riassunto

- Inizio pattern di progetto
- Adapter
- Façade
- Composite
- Decorator