Pattern Composite

Ex presentazione realizzata
dallo studente Matteo Biancardi
nell'a.a. 2006/2007

Composite

- **Strutturale**

  I pattern strutturali si occupano delle modalità di composizione di classi e oggetti per formare strutture complesse.

- **Basato su oggetti**

  I pattern basati su oggetti descrivono modalità di composizione di oggetti.

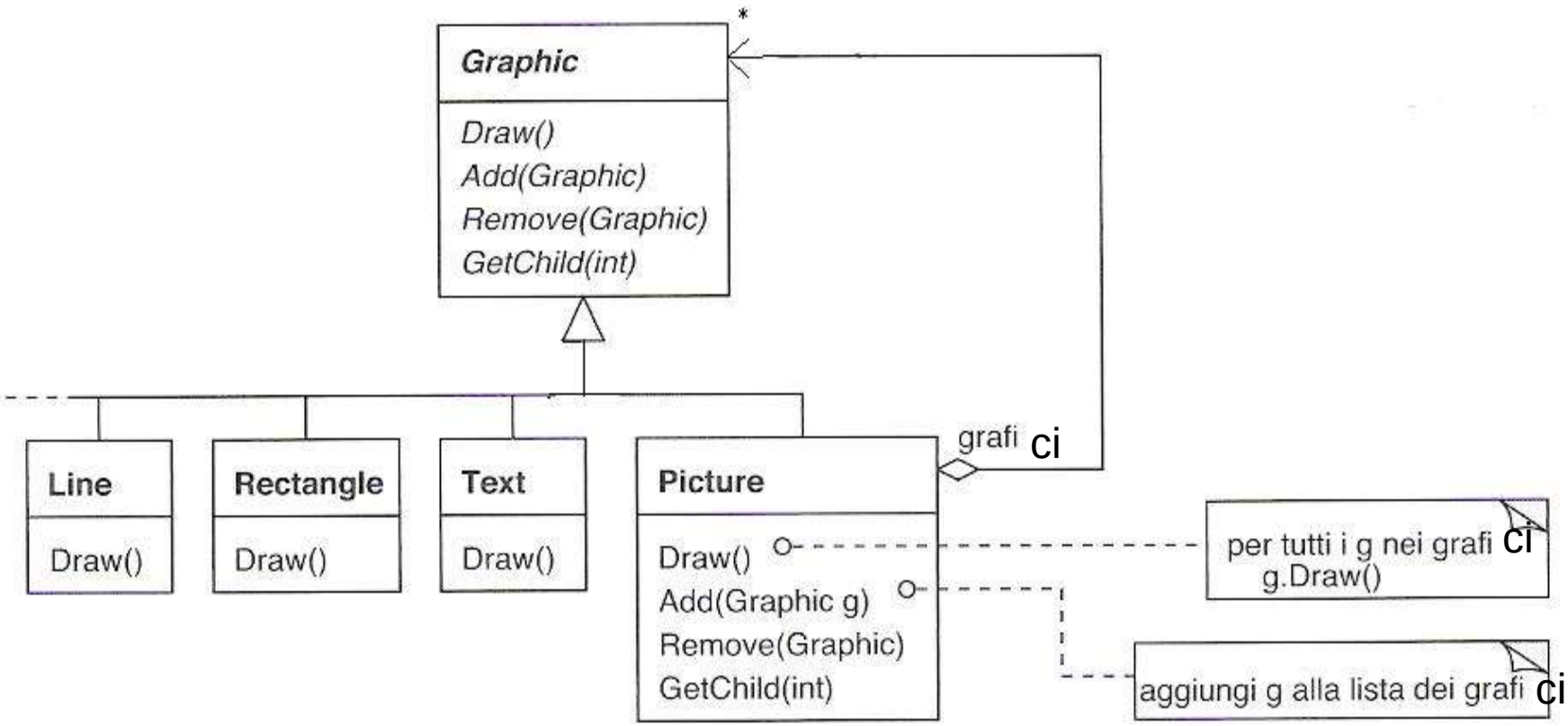
  L'utilizzo della composizione tra oggetti fornisce flessibilità che deriva dalla possibilità di cambiare la composizione durante l'esecuzione, fatto impossibile se si utilizza la composizione statica fra classi.

- **Scopo**

  Comporre oggetti in strutture ad albero rappresentanti gerarchie parte – tutto e consentire ai client di trattare oggetti singoli e composizioni in modo uniforme.

Composite

Motivazione



Applicabilità

Il pattern Composite può essere utilizzato quando:

- si vogliono rappresentare oggetti strutturati in gerarchie parte – tutto;
- si vuole fare in modo che i client possano ignorare la differenza fra oggetti singoli e composizioni di oggetti.

Struttura

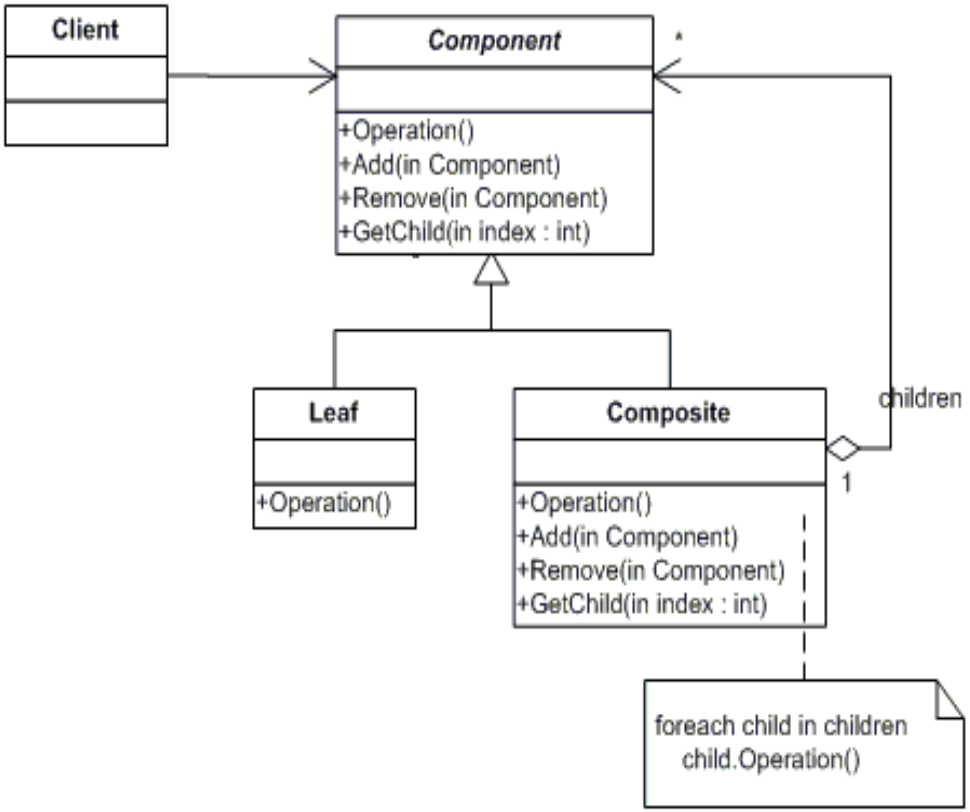
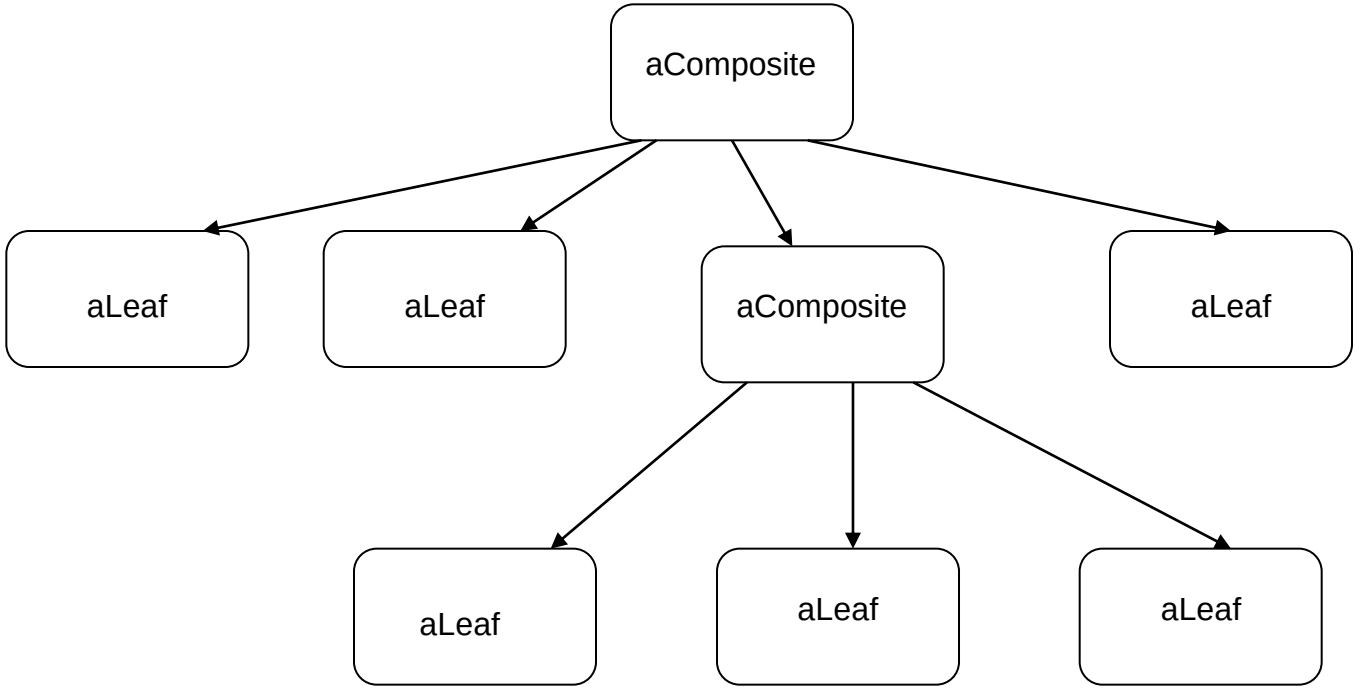

Client
Component
*
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
Leaf
+Operation()
Composite
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
children
1
foreach child in children
child.Operation()

Diagramma degli oggetti

Una tipica struttura di oggetti Composite potrebbe essere:



Partecipanti

- **Component (Graphic)**

  - Dichiara l'interfaccia per gli oggetti che fanno parte della composizione.
  - Implementa il comportamento standard comune a tutte le classi,
  - Dichiara un'interfaccia per l'accesso e la gestione dei suoi componenti figli.
  - (opzionale) Definisce un'interfaccia per accedere al componente padre nella struttura ricorsiva.

- **Leaf (Rectangle, Line, Text, ecc.)**

  - Rappresenta gli oggetti foglia nella composizione.
  - Definisce il comportamento degli oggetti primitivi nella composizione.

- **Composite (Picture)**

  - Definisce il comportamento per i componenti che hanno figli.
  - Memorizza i componenti figli.
  - Implementa le operazioni correlate ai figli definite nell'interfaccia Component.

- **Client**

  - Manipola gli oggetti della composizione utilizzando l'interfaccia Component.

Collaborazioni

I client utilizzano l'interfaccia della classe Component per interagire con gli oggetti della struttura composita.

- Se l'oggetto destinatario è una foglia:
  la richiesta viene gestita direttamente.

- Se l'oggetto è Composite:

  la richiesta è solitamente trasferita ai suoi oggetti figli.

Conseguenze

Il pattern Composite:

- Definisce gerarchie costituite da oggetti primitivi e compositi.

  Gli oggetti primitivi possono essere composti per formare oggetti più complessi, che, a loro volta, potranno essere composti ricorsivamente.

- Semplifica il (codice del) client.

  I client possono trattare strutture composite e oggetti singoli in modo uniforme. I client non sanno se stanno operando su una foglia o su un componente composito.

Conseguenze

- Rende più semplice l'aggiunta di nuove tipologie di componenti.

  Nuove sottoclassi Leaf o Composite potranno essere utilizzate automaticamente nelle strutture esistenti e operare con il codice del client. Il codice del client non dovrà essere modificato qualora venissero aggiunte sottoclassi di Component.

- Può rendere il progetto troppo generico.

  Lo svantaggio di rendere più facile l'aggiunta di nuovi componenti è che ciò rende difficile limitare i tipi dei componenti che fanno parte di una specifica struttura composita. Se si vuole introdurre questo limite, è necessario che il codice effettui dei controlli durante l'esecuzione.

Implementazione

- Riferimenti espliciti ai padri
  - Semplificazione dell'attraversamento e della gestione (di una struttura composita).
  - Riferimento al padre nella classe *Component*.
  - Mantenimento invariante padre-figli e viceversa (tutti i figli di un elemento composito devono avere come padre tale elemento e questo deve aver come figli tutti gli oggetti che lo referenziano come padre).
  - Modificare il padre di un componente soltanto quando il componente stesso viene aggiunto o rimosso dalla struttura composita (inserire questa verifica nei metodi **Add (Component)** e **Remove(Component)** della classe Composite).

Implementazione

- Condivisione di componenti
  - Memorizzare i riferimenti ai padri nei figli crea ambiguità.
  - Pattern Flyweight per evitare la memorizzazione dei riferimenti ai padri (laddove i figli possono fare a meno di inviare richieste al padre, portando al loro esterno – in parte o in toto – il proprio stato)

Implementazione

- Massimizzazione dell'interfaccia di *Component*
  - La classe *Component* dovrebbe definire il maggior numero possibile di operazioni comuni alle classi Leaf e Composite e fornirne un'implementazione di default.
  - Può scontrarsi con il principio di sostituzione di Liskov.
  - Implementazione di default delle operazioni che sembrano avere significato solo per Composite: un po' di creatività può aiutare! Ad es. operazione per accedere ai figli: per default non restituisce alcun figlio (va bene per le classi Leaf, viene sovrascritta in Composite)

Implementazione

- Dove dichiarare le operazioni di gestione dei figli, in *Component* o in Composite?
  - Scelta tra sicurezza e trasparenza.

Sicurezza

- Dichiarare un metodo **getComposite ()** nella classe *Component*. Questo metodo, nell'implementazione fornita dalla classe *Component*, restituisce un puntatore **nullo**, mentre, nell'implementazione della classe Composite, restituisce un riferimento all'oggetto su cui è stato invocato, cioè ritorna **this**.
- **getComposite ()** consente di interrogare un componente per verificare se è di tipo Composite. Se l'oggetto è composto, è possibile eseguire in modo sicuro i metodi **Add (Component)** e **Remove (Component)**.
- Ma in questo modo non si possono trattare i componenti in modo uniforme …

Trasparenza

- Definire i metodi **Add (Component)** e **Remove (Component)** con un'implementazione standard in *Component*.
- Fare in modo che i due metodi abbiano un comportamento base che porta al fallimento dell'operazione (cioè si solleva un'eccezione) se il componente su cui viene invocata l'operazione **Add** o **Remove** non può avere figli.

Implementazione

- Ordinamento dei figli
  - È spesso utile progettare interfacce per l'accesso e la gestione dei figli in modo da garantire l'ordinamento dei figli di un oggetto composto (es. albero sintattico). A tal fine si può sfruttare il pattern Iterator.

Implementazione

- Utilizzo di cache per migliorare le prestazioni
  - La classe Composite può operare come memoria cache per memorizzazione informazioni utili per semplificare la ricerca e l'attraversamento dei figli (ad es. un Picture può memorizzare la visibilità dei figli).
  - I cambiamenti in un componente figlio richiedono di invalidare la cache del padre $\rightarrow$ necessità di un'interfaccia di notifica all'oggetto composito.

Implementazione

- Chi cancella i componenti?
  - *Garbage collection* oppure responsabilità della classe Composite.
  - Eccezione nel caso di oggetti foglia immutabili e condivisi.

Implementazione

- Qual è la migliore struttura dati per la memorizzazione dei componenti (entro la classe Composite)?
  - Varietà di strutture dati per la memorizzazione dei figli (liste concatenate, array, alberi, tabelle hash).
  - Scelta da effettuarsi di caso in caso in base all'efficienza.

Pattern correlati

- Decorator
  - Pattern spesso utilizzato con Composite.
  - Se oggetti Decoratori e Compositi sono usati insieme, solitamente hanno una superclasse comune e in questo caso i decoratori devono supportare l'interfaccia *Component* con metodi quali Add(Component), Remove(Component), GetChild(int).

Pattern correlati

- Iterator
  - Fornisce un accesso sequenziale agli oggetti presenti in una collezione, senza esporre la rappresentazione interna della stessa.
  - Può essere utilizzato per attraversare le strutture composite.

Pattern correlati

- Visitor
  - Consente la definizione di nuove operazioni senza modificare le classi degli elementi sui quali opera.
  - Localizza operazioni e comportamento che altrimenti dovrebbero essere distribuiti fra le classi Composite e Leaf.

Pattern correlati

- Flyweight

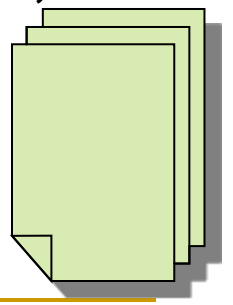
  Scopo

  Utilizzare la condivisione per supportare in modo efficiente un gran numero di oggetti a granularità fine.

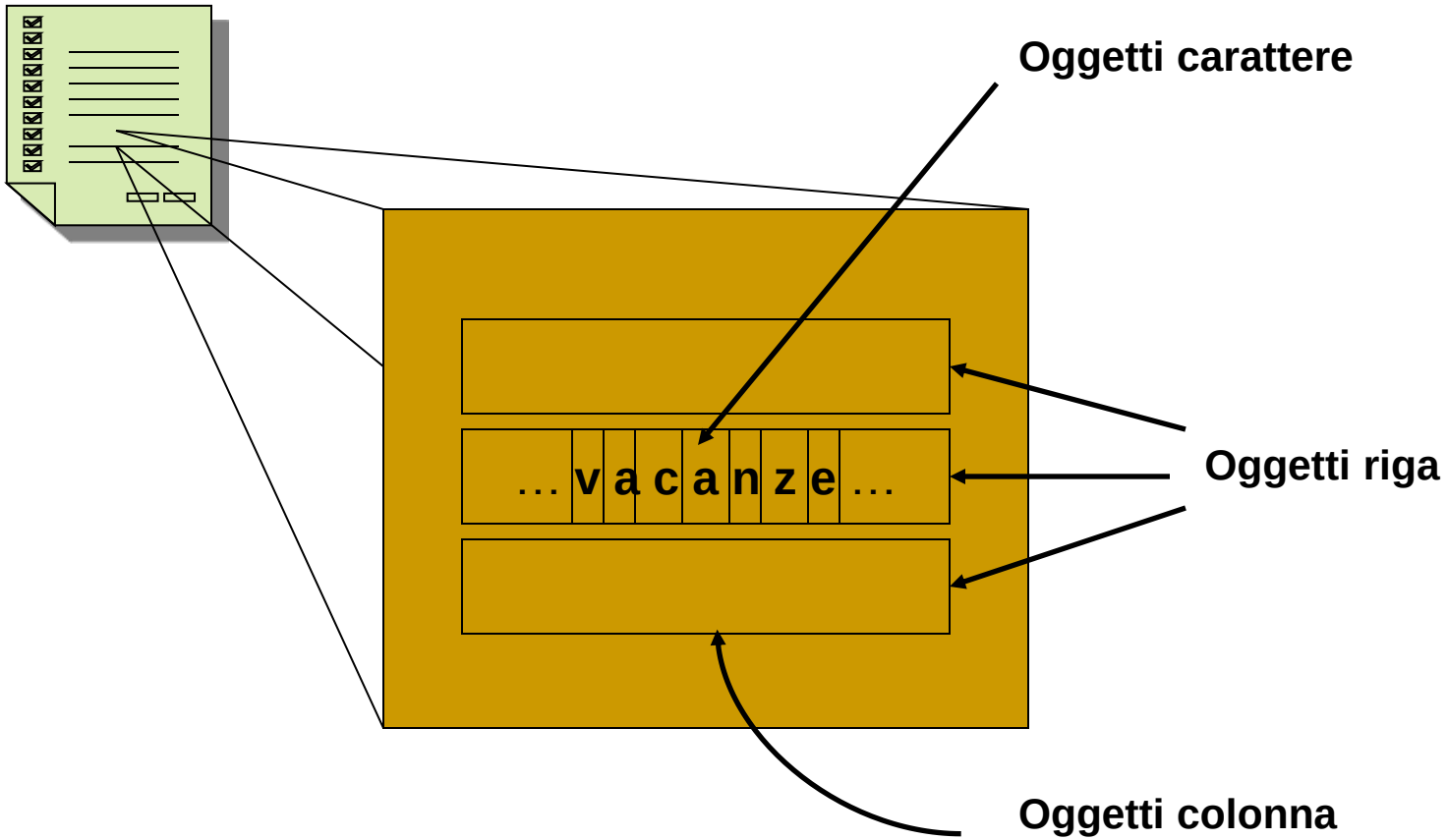
  Motivazione

  Alcune applicazioni potrebbero trarre beneficio dall'approccio a oggetti fin dalla progettazione, ma un'implementazione ingenua risulterebbe assai onerosa.

  Esempio: editor di documenti

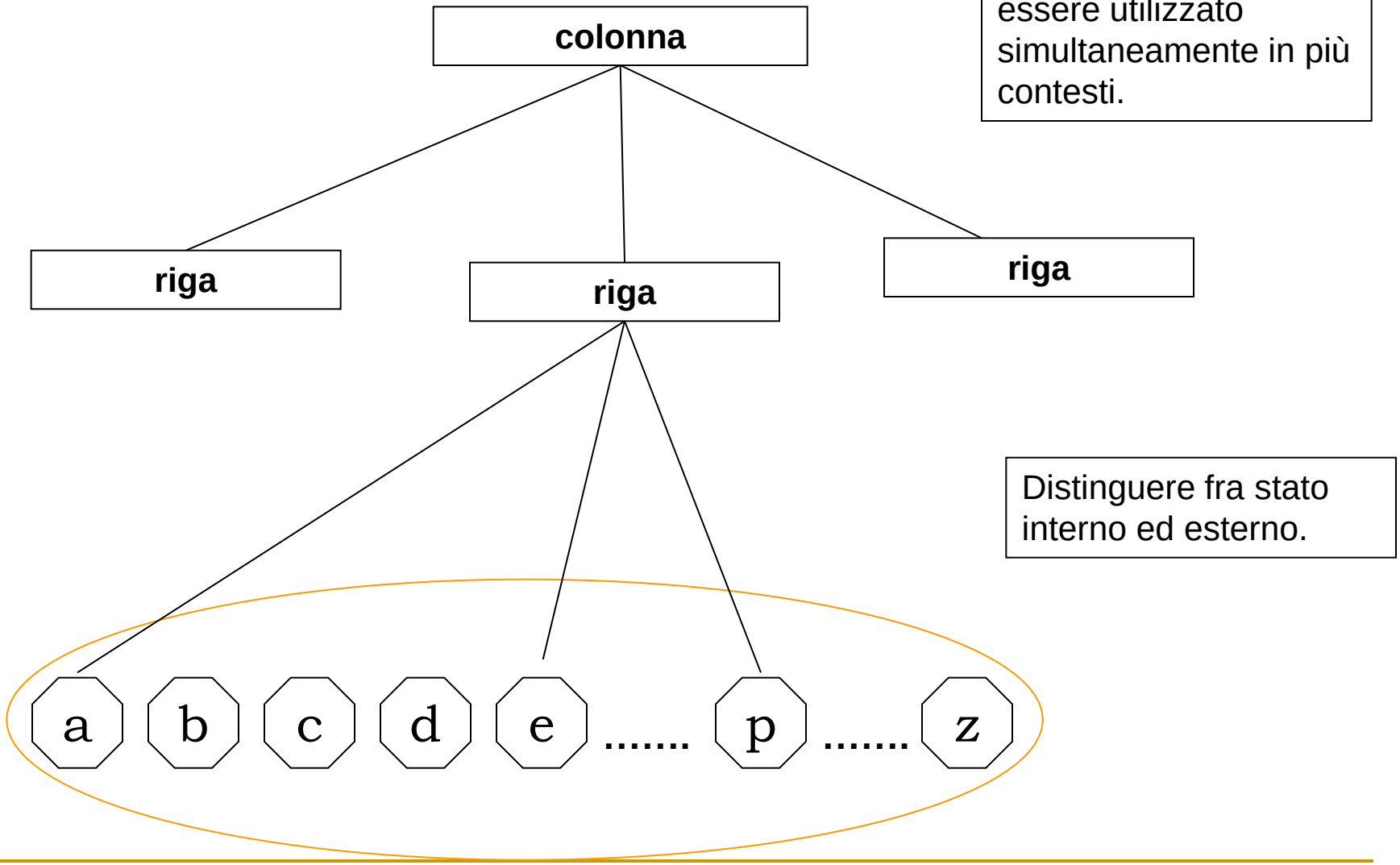


Pattern correlati



Pattern correlati

Flyweight: oggetto condiviso che può essere utilizzato simultaneamente in più contesti.



Distinguere fra stato interno ed esterno.

Riepilogo

Il pattern Composite racchiude due concetti potenti e correlati:

- Il primo è che un gruppo può contenere sia elementi singoli sia altri gruppi.
- Collegato a questo concetto, l'idea che sia i gruppi sia gli oggetti singoli possano condividere un'interfaccia comune.