Il pattern Composite

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@dimes.unical.it
Web: http://angelo.furfaro.dimes.unical.it

Composite

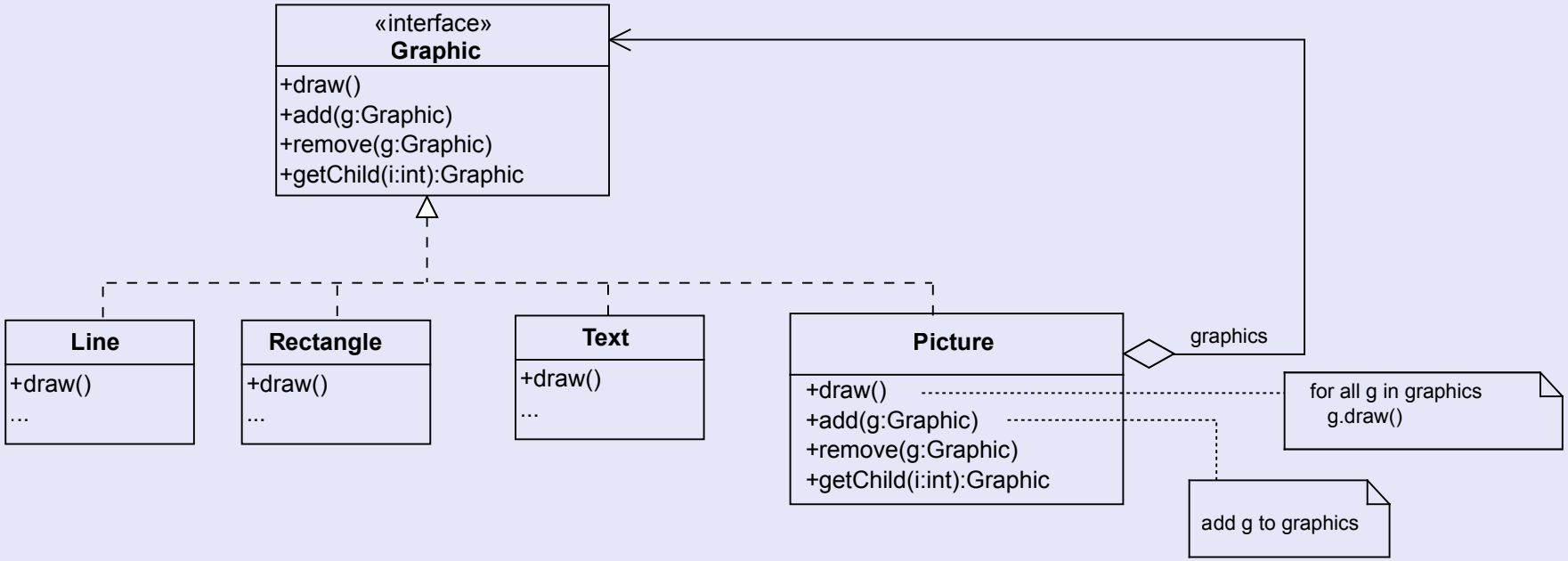
Classificazione

- Scopo: strutturale
- Raggio d'azione: basato su oggetti

Scopo

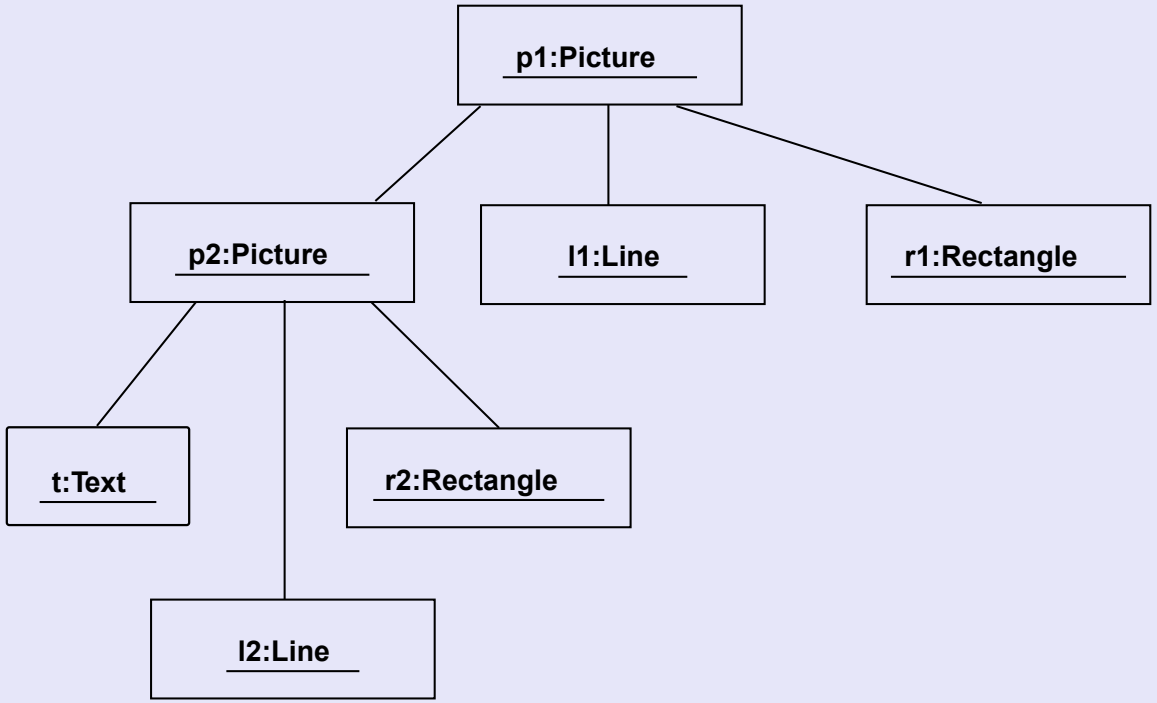
- Comporre oggetti in strutture ad albero per rappresentare gerarchie parte-tutto e consentire ai client di trattare oggetti singoli e composizioni in modo uniforme.

Motivazione



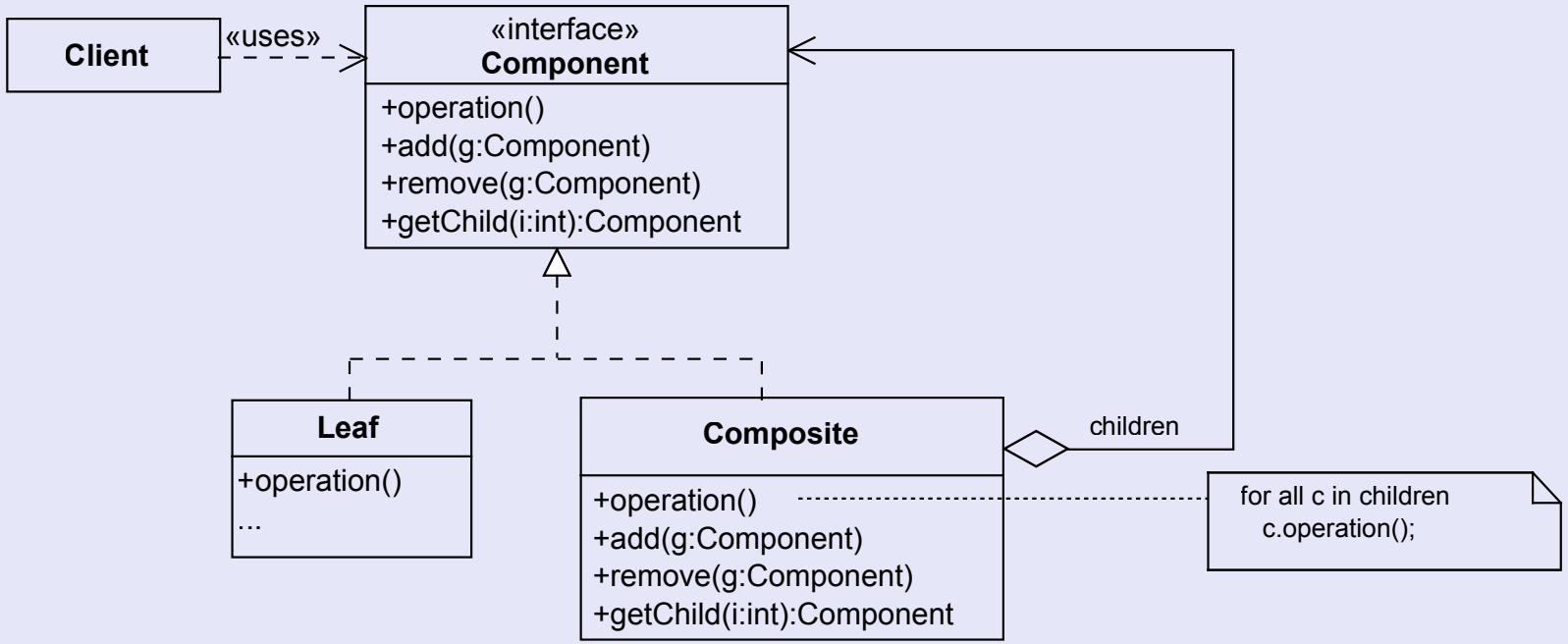
- Applicazioni quali editor grafici vettoriali o ambienti per la progettazione di circuiti consentono agli utenti di costruire diagrammi complessi a partire da semplici componenti.
- Componenti semplici possono essere raggruppati per costruire componenti più complessi che possono, a loro volta, essere utilizzati come parti di componenti ancor più complessi.
- Solitamente si introducono alcune classi per modellare gli oggetti semplici ed altre per rappresentare gli oggetti ottenuti per composizione.
- Il pattern Composite introduce un'interfaccia comune per gli oggetti semplici e per quelli compositi in modo che il codice cliente li possa trattare uniformemente.

Motivazione



- Nell'esempio, le classi `Line`, `Rectangle` e `Text` definiscono gli oggetti primitivi.
- Il metodo `draw()` implementa l'operazione di disegno. Poiché gli oggetti primitivi non hanno *figli*, le operazioni per la gestione dei componenti non sono implementate.
- La classe `Picture` è un aggregato di oggetti `Graphic` ed implementa `draw()` invocandolo a sua volta sugli oggetti da cui è composto.
- L'object diagram riportato illustra una tipica struttura di oggetti `Graphic` ricorsivamente composti.

Struttura



Partecipanti

- **Component** (`Graphic`): dichiara l'interfaccia per gli oggetti *componibili*. Può essere una classe astratta che fornisce un'implementazione di default per i metodi di gestione dei componenti *figli*. Introduce (opzionale) un metodo per accedere al componente genitore.
- **Leaf** (`Rectangle`,`Line`, etc.): rappresenta gli oggetti *primitivi* i quali non hanno figli.
- **Composite**(`Picture`): definisce il comportamento dei componenti che hanno figli. Memorizza i componenti figli e implementa i relativi metodi introdotti dall'interfaccia `Component`.

Conseguenze

- ☺ Definisce gerarchie di classi costituite da oggetti primitivi ed oggetti compositi. Gli oggetti primitivi possono essere composti per dare origine ad oggetti più complessi. Il codice cliente tratta oggetti compositi e primiviti in modo indifferente.
- ☺ Il codice cliente non sa (e normalmente non è interessato a saperlo) se sta utilizzando un oggetto primitivo o uno composito.
- ☺ Rende semplice l'aggiunta di nuovi tipi di componenti seza influenzare il codice cliente.
- ☹ L'interfaccia `Component` potrebbe essere *troppo* generale. Può essere necessario vincolare un particolare tipo di oggetto composito ad avere solo determinati tipi di componenti. Il compilatore non può garantire tale vincolo, per cui è necessario ricorrere a verifiche a tempo di esecuzione.

Implementazione

- *Riferimenti al genitore*
  Fare in modo che ciascun componente conservi il riferimento al genitore può semplificare le operazioni di gestione e di *visita* della struttura composita. È importante fare in modo che tutti i figli di un oggetto composito riferiscano esso stesso come genitore. Il modo più semplice per garantire ciò consiste nel modificare il riferimento al genitore contestualmente alle operazioni di aggiunta e rimozione dei figli.
- *Massimizzazione dell'interfaccia* `Component`
  Al fine di rendere uniforme l'interfaccia dei componenti, `Component` potrebbe include alcune operazioni che non hanno senso per i componenti primitivi. Ciò è in contrasto con il principio secondo cui una classe astratta dovrebbe includere solo operazioni che hanno senso per le sottoclassi.
- *Dichiarazione di metodi di gestione dei figli*
  Una scelta importante nell'implementazione del pattern riguarda il punto in cui sono definite le operazioni di gestione dei figli. Nell'esempio riportato esse sono dichiarate da `Component`, un'alternativa sarebbe quella di introdurle nella classe `Composite`.
  - La prima scelta garantisce l'uniformità di interfaccia ma, di contro, se invocate su componenti primitivi tali operazioni falliscono o non hanno effetto.
  - Nel secondo caso ci sono maggiori garanzie a tempo di compilazione ma si perde in termini di trasparenza.

TextDocExample: rappresentazione del documento in memoria centrale