11.
I pattern Composite e Decorator

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli “Federico II”

Corso di Linguaggi di Programmazione II

- Riesaminiamo il meccanismo che permette di aggiungere componenti grafici (ad es., pulsanti) ad un pannello Swing/AWT
- Consideriamo l'esempio seguente, che posiziona tre pulsanti in una finestra
- Il risultato è mostrato nella figura a destra

```
// un contenitore
Container c1 = new Container();
c1.setLayout(new FlowLayout());

// aggiungo due pulsanti al contenitore
c1.add(new Jbutton(“qui”));
c1.add(new Jbutton(“qua”));

// la finestra
JFrame f = new JFrame();
Container c2 = f.getContentPane();
c2.setLayout(new FlowLayout());

c2.add(c1); // c1 si comporta come un componente!
c2.add(new JButton(“quo”));
```

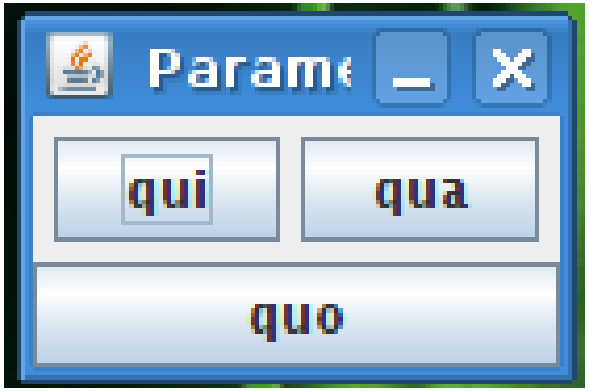


Figura 1: La finestra che si ottiene eseguendo il codice a sinistra.

Contenitori e componenti

- Cominciamo col creare un contenitore c1, nel quale inseriamo i due pulsanti “qui” e “qua”
  - il *FlowLayout* ci assicura che, finché c'è abbastanza spazio, i due pulsanti saranno posizionati uno di fianco all'altro
- Poi, creiamo una finestra e otteniamo un riferimento c2 al suo pannello del contenuto (contentPane)
  - come c1, anche il pannello del contenuto è di tipo *Container*
- Per ottenere la disposizione voluta, **aggiungiamo il contenitore c1 al contenitore c2**, per poi aggiungere a quest'ultimo anche il pulsante “quo”
  - per aggiungere c1 a c2, abbiamo trattato il contenitore c1 come un normale componente
  - difatti, il tipo accettato dal metodo add è *Component*
  - sia *Container* che *JButton* estendono *Component*
    - in realtà, *JButton* estende *Container*, che a sua volta estende *Component*
- Questo meccanismo corrisponde al pattern **COMPOSITE**

In breve: aggregazione ricorsiva di oggetti

Contesto:

1) Oggetti primitivi possono essere combinati in un oggetto composito
2) I clienti possono trattare un oggetto composito come primitivo

Soluzione:

1) Definire un'interfaccia (*Primitive*) che rappresenti un'astrazione dell'oggetto primitivo
2) Un oggetto composito contiene una collezione di oggetti primitivi
3) Sia gli oggetti primitivi che quelli compositi implementano l'interfaccia Primitive
4) Nel realizzare un metodo dell'interfaccia Primitive, un oggetto composito applica il metodo corrispondente a tutti i propri oggetti primitivi, e poi combina i risultati ottenuti

Diagramma del pattern COMPOSITE

- Il diagramma a destra illustra le principali classi coinvolte nel pattern COMPOSITE, e le relazioni tra di loro

- **Primitive** è l'interfaccia che rappresenta un oggetto primitivo
- Sia i veri oggetti primitivi (qui chiamati **Leaf**, "foglia") sia gli oggetti compositi implementano l'interfaccia Primitive
  - non è detto che Primitive sia sempre un'interfaccia, come nel caso di Component, illustrato nella prossima slide

- La relazione di **aggregazione** tra Composite e Primitive indica che un oggetto di tipo Composite contiene un insieme di riferimenti ad oggetti primitivi

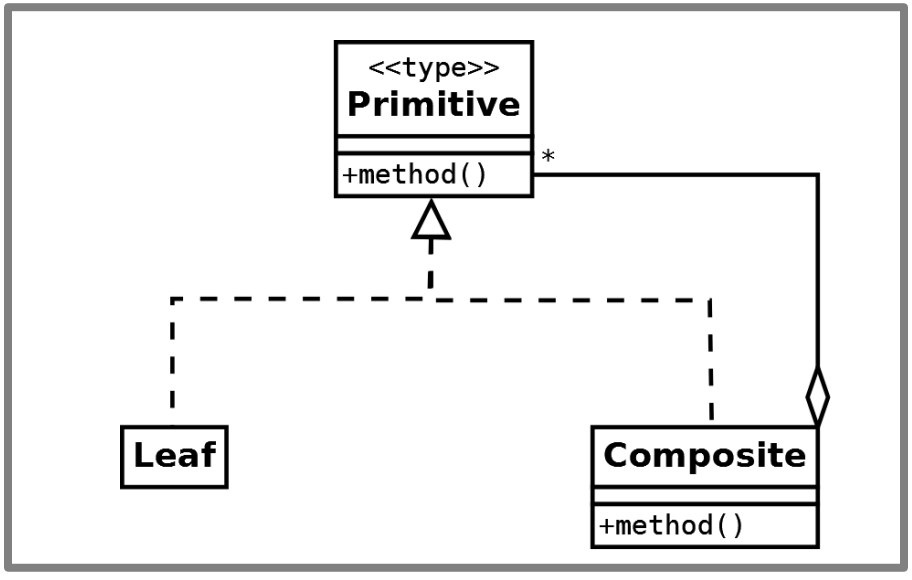


Figura 2: Diagramma UML tipico del pattern COMPOSITE.

Applicazione del pattern COMPOSITE

- Il diagramma a destra illustra come il pattern COMPOSITE è stato applicato nel caso dei contenitori e dei componenti grafici Swing/AWT
- Il ruolo dell'interfaccia Primitive è ricoperto dalla classe Component
  - getPreferredSize è uno dei tanti metodi di quella classe
  - restituisce le dimensioni "ideali" di questo componente
- Sia JButton che Container estendono Component
- Il metodo add di Container permette di aggiungere un altro Component all'insieme

- In accordo col pattern, quando un contenitore deve rispondere ad una chiamata a getPreferredSize, esso chiamerà il metodo omonimo su tutti i componenti contenuti, e poi sommerà le dimensioni ottenute, sulla base del suo layout, per restituire le sue dimensioni ideali

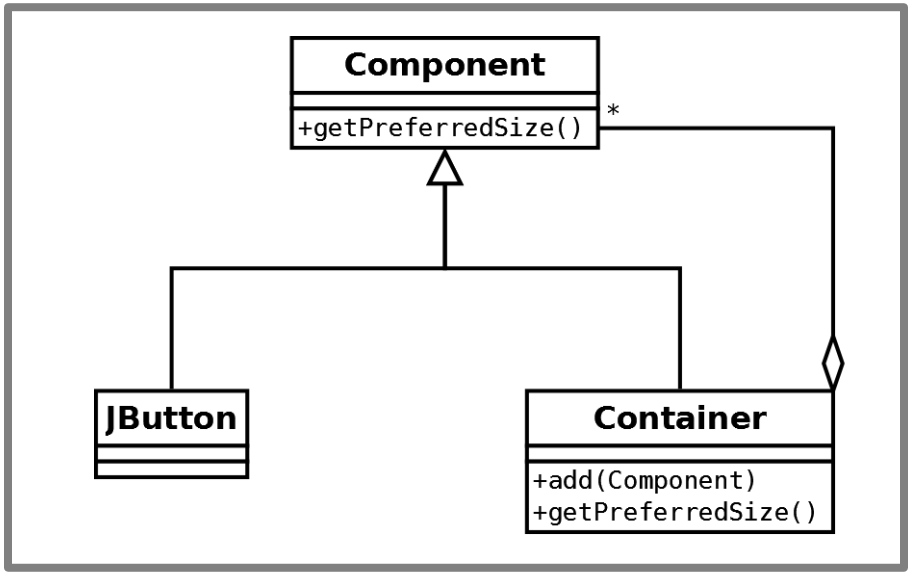


Figura 3: Applicazione del pattern COMPOSITE a componenti e contenitori della libreria grafica Swing/AWT.

Pattern DECORATOR

Barre di scorrimento

- Per introdurre un nuovo pattern, esaminiamo il meccanismo per **aggiungere delle barre di scorrimento** ad un componente Swing/AWT
  - ad esempio, ad un campo di testo di più righe

```
/* un'area di testo con 10 righe e 25 colonne */
Component area = new JTextArea(10, 25);
```

Come progettereste l'aggiunta di una barra di scorrimento?

Barre di scorrimento

- Per introdurre un nuovo pattern, esaminiamo il meccanismo per **aggiungere delle barre di scorrimento** ad un componente Swing/AWT
  - ad esempio, ad un campo di testo di più righe

```
/* un'area di testo con 10 righe e 25 colonne */
Component area = new JTextArea(10, 25);

/* aggiungiamo le barre di scorrimento */
Component scrollArea = new JScrollPane(area);
```

- Notiamo che si deve creare **un nuovo oggetto**, di tipo effettivo *JScrollPane*, a partire dall'oggetto *JTextArea* esistente
- Inoltre, l'oggetto così creato è a sua volta sottotipo di *Component*

- Questo meccanismo viene codificato dal pattern **DECORATOR**, presentato nella prossima slide

Il pattern DECORATOR

In breve: aggiungere funzionalità senza coinvolgere la classe

Contesto:

1) Si vuole decorare (ovvero *migliorare*, ovvero *aggiungere funzionalità a*) una classe componente
2) Un componente decorato può essere utilizzato nello stesso modo di uno normale
3) La classe componente non vuole assumersi la responsabilità della decorazione
4) L'insieme delle decorazioni possibili non è limitato

In breve: aggiungere funzionalità senza coinvolgere la classe

Soluzione:

1) Definire un'interfaccia (*Component*) che rappresenti un'astrazione di un componente

2) Le classi concrete che definiscono componenti implementano l'interfaccia *Component*

3) Definire una classe (*Decorator*) che rappresenta la decorazione

4) Un oggetto decoratore contiene e gestisce l'oggetto che decora

5) Un oggetto decoratore implementa l'interfaccia *Component*

6) Nel realizzare un metodo di *Component*, un oggetto decoratore applica il metodo corrispondente all'oggetto decorato e ne combina il risultato con l'effetto della decorazione

Diagramma del pattern DECORATOR

- Il diagramma a destra illustra le principali classi coinvolte nel pattern DECORATOR, e le relazioni tra di loro

- L'interfaccia **Component** rappresenta un componente generico
- Sia i veri componenti base (ConcreteComponent) che le loro decorazioni (Decorator) implementano l'interfaccia, in modo che il client li possa usare nello stesso modo

- La relazione di **aggregazione** indica che un oggetto decoratore contiene un riferimento al componente che sta decorando
  - presumibilmente, tale riferimento viene inizializzato tramite un **costruttore** o metodo di Decorator

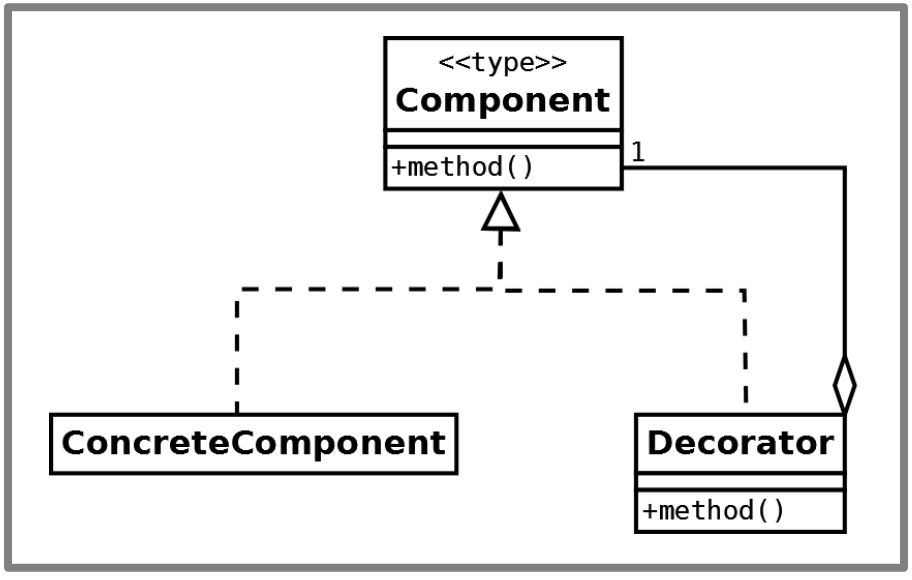


Figura 4: Diagramma UML tipico del pattern DECORATOR.

Applicazione del pattern DECORATOR

- Il diagramma a destra illustra come il pattern DECORATOR è stato applicato nella libreria Swing/AWT

- Sia il campo di testo (JTextArea) che le barre di scorrimento (JScrollPane) estendono la classe Component
- Il costruttore di JScrollPane prende come argomento il componente a cui applicare la decorazione
- Quando un client chiama **paint** (il metodo che chiede ad un componente di disegnare il suo contenuto) di un oggetto JScrollPane, quest'ultimo chiama il metodo omonimo dell'oggetto decorato e poi aggiunge il disegno delle barre di scorrimento

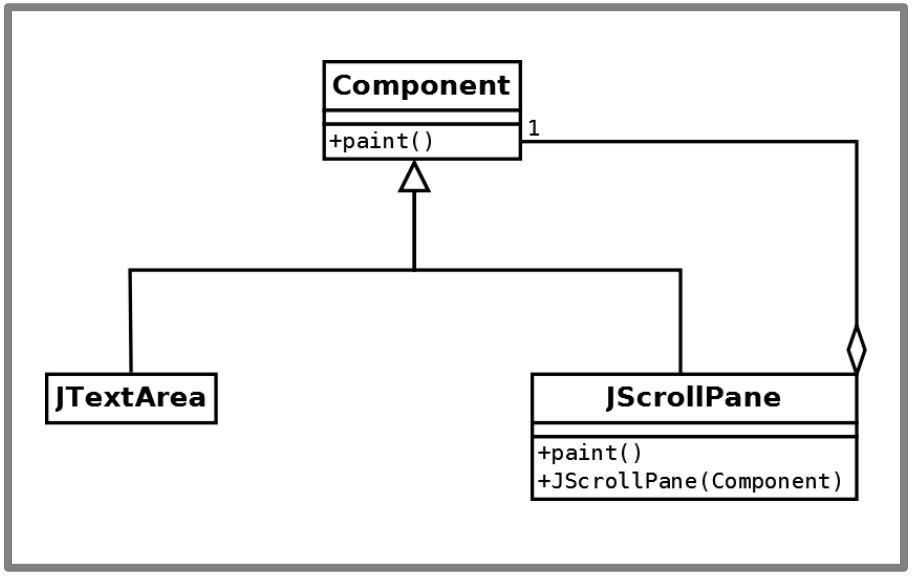
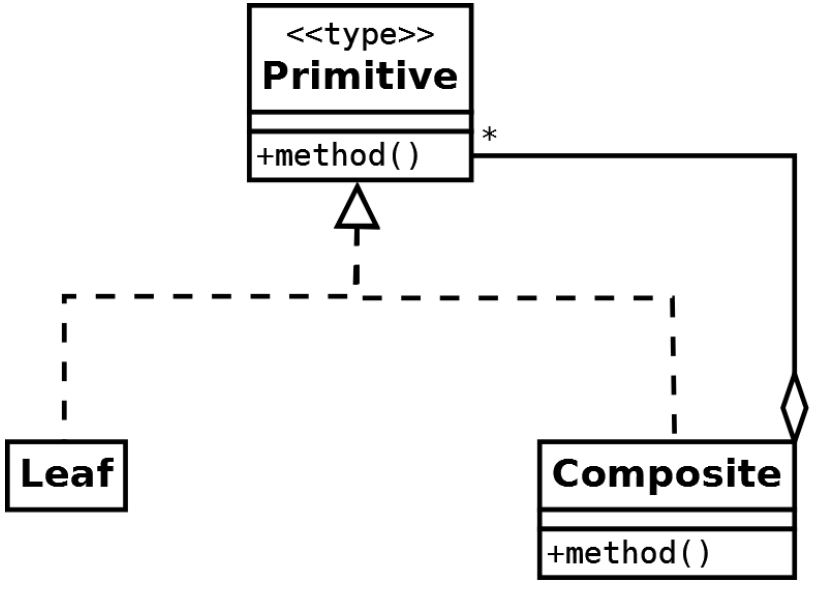
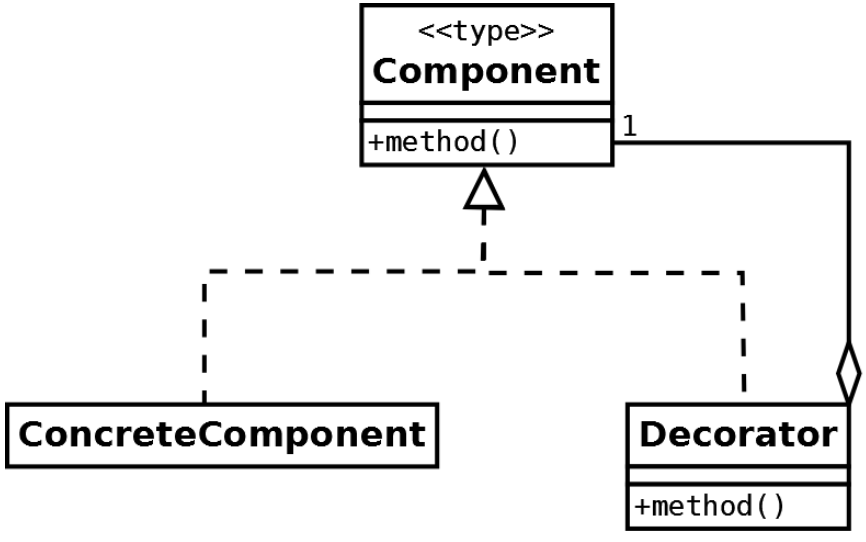


Figura 5: Diagramma dell'applicazione del pattern DECORATOR all'aggiunta di barre di scorrimento ad un campo di testo multi-rigo.

Confronto tra COMPOSITE e DECORATOR





Confronto tra COMPOSITE e DECORATOR

- Confrontando le descrizioni e i diagrammi dei pattern COMPOSITE e DECORATOR, notiamo diverse somiglianze
- Entrambi prevedono una distinzione tra oggetti di base (chiamati *primitivi* in un caso e *componenti* nell'altro) e oggetti compositi
- Entrambi prevedono che gli oggetti delle due categorie implementino un'interfaccia comune

- Tuttavia, i contesti di applicazione sono molto diversi:

- **COMPOSITE** si applica quando bisogna **aggregare più oggetti** di base in un unico oggetto, che può a sua volta essere aggregato con altri
  - si ottiene così una gerarchia di oggetti disposti in un **albero**

- **DECORATOR** si applica quando bisogna **aggiungere funzionalità**, illimitate a priori, ad una classe, senza coinvolgere la classe stessa