

Composite

- **Intento**
 - Comporre oggetti in strutture ad albero per rappresentare gerarchie di parti o del tutto. Composite permette ai client di trattare oggetti singoli e composizioni di oggetti uniformemente
- **Motivazione**
 - In molti casi è necessario raggruppare elementi semplici tra loro per formare elementi più grandi
 - Se nell'implementazione c'è distinzione tra classi per elementi semplici e classi per contenitori di questi elementi semplici, il codice che usa queste classi deve trattarli in modo differente
 - Questa distinzione rende il codice più complicato
 - Il design pattern Composite permette di descrivere una composizione ricorsiva, in modo che i client non debbano fare distinzione tra tipi di elementi
 - I client tratteranno tutti gli oggetti della struttura uniformemente

Esempi

- Le operazioni su disco permettono la gestione di file (immagini, testo, etc.) e la gestione di cartelle
- I client possono voler rappresentare file e cartelle, senza distinguere tra questi
- Una cartella deve poter contenere al suo interno sia file che altre cartelle, queste a sua volta contengono altri elementi, e così via

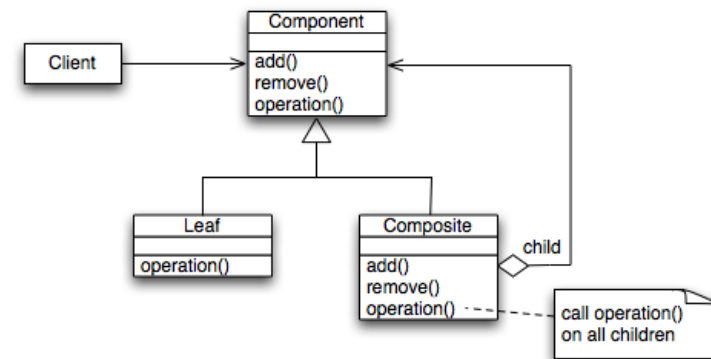
- In un editor di figure ho sia figure semplici, es. Linea, Box, che figure composte che raggruppano figure semplici e composte
- L'utente tratta allo stesso modo sia le figure semplici che quelle composte

Composite

- **Soluzione**
 - L'elemento chiave del design pattern Composite è la classe abstract Component che rappresenta elementi semplici e contenitori
 - La classe Component dichiara le operazioni degli oggetti da comporre
 - Implementa le operazioni comuni alle sottoclassi, in modo appropriato
 - Per gli elementi contenitori dichiara le operazioni per l'accesso e la gestione degli elementi semplici
 - Può definire una operazione per permettere agli elementi di accedere all'oggetto che è il loro padre nella struttura ricorsiva
 - La classe Leaf rappresenta elementi semplici (*child*)
 - E' sottoclasse di Component
 - Implementa il comportamento degli oggetti semplici
 - La classe Composite rappresenta elementi contenitori
 - Definisce il comportamento per l'aggregato di componenti child
 - Immagazzina il riferimento ai componenti child
 - Implementa operazioni per gestire i componenti *child*

Composite

- **Struttura**



Composite

- Collaborazioni
 - I client usano l'interfaccia di Component per interagire con gli oggetti della struttura composita
 - Se il ricevente del messaggio è Leaf, la richiesta è gestita direttamente
 - Se il ricevente è un Composite, questo rimanda la richiesta ai suoi child e possibilmente avvia operazioni aggiuntive prima e dopo
- Conseguenze
 - Oggetti elementari possono essere composti in oggetti più complessi, questi possono essere composti, e così via
 - Un client che si aspetta un oggetto elementare può prendere anche un oggetto composto
 - I clienti sono semplici, possono trattare strutture composte ed oggetti semplici uniformemente. I client non sanno se trattano con un oggetto Leaf o un Composite
 - Nuovi tipi di componenti (Leaf o Composite) possono essere aggiunti e potranno funzionare con la struttura ed i client esistenti
 - Non è possibile a design time vincolare il Composite solo su certi componenti Leaf, dovranno essere invece fatti controlli a runtime

E. Tramontana - Composite - 17 - May 10 5

Composite

- Ulteriori dettagli
 - Per facilitare la navigazione della struttura composta i componenti child possono mantenere il riferimento all'oggetto che li contiene. Il riferimento può essere inserito in Component, mentre Leaf e Composite possono implementare le operazioni per gestirlo
 - Per avere client che non distinguono se stiano trattando Leaf o Composite (è uno degli obiettivi), la classe Component dovrebbe definire quante più operazioni in comune possibile. Le classi Leaf e Composite faranno override delle operazioni
 - Dove dichiarare le operazioni di gestione dei child, add() e remove()?
 - Se dichiarate in Component si ha trasparenza, ma per le classi Leaf tali operazioni non hanno significato. La sicurezza nell'uso è quindi compromessa, poiché i client potrebbero avviarle su Leaf. Se i client avviano una operazione add() su Leaf e si ignora la richiesta, questo potrebbe indicare un bug
 - Se dichiarate in Composite si ha sicurezza, poiché a compile time si verifica che tali operazioni non possono essere chiamate su Leaf, ma si perde la trasparenza

E. Tramontana - Composite - 17 - May 10 6

Composite

- Si può inserire una operazione getComposite() in Component che ritorna null e che è ridefinita in Composite per ritornare il riferimento a se stesso. I client dovrebbero comunque distinguere il tipo di risultato e fare operazioni differenti, niente trasparenza
- La lista che contiene i componenti child dovrebbe essere definita in Composite, altrimenti se definita in Component si spreca spazio, poiché ogni Leaf avrebbe tale variabile anche se non deve usarla mai
- L'ordinamento dei child per un composite potrebbe essere importante e va tenuto in considerazione su certe implementazioni
- Il Composite potrebbe implementare una cache, per ottimizzare le prestazioni, quando gli si richiede un valore che deve ricercare tra tutti i suoi componenti child. I componenti child devono poter accedere ad una operazione che permette di invalidare la cache del Composite

E. Tramontana - Composite - 17 - May 10 7

Esempio codice

```
// Resource plays role Component
public abstract class Resource {
    public abstract void show();
    //add, remove not to be called on Leaf
    public void add(Resource c) {}
    public void remove(Resource c) {}
}

// Image plays role Leaf
public class Image extends Resource {
    private int x, y;
    private String name;
    public Image(String id, int a, int b) {
        name = id; x = a; y = b;
    }
    @Override public void show() {
        // some code here
    }
}

// Folder plays role Composite
import java.util.LinkedList;
public class Folder extends Resource {
    private String name;
    private LinkedList<Resource> r =
        new LinkedList<Resource>();
    public Folder(String f) {
        name = f;
    }
    @Override public void show() {
        for (int i = 0; i<r.size(); i++)
            r.get(i).show();
    }
    @Override public void add(Resource c) {
        r.add(c);
    }
}
```

E. Tramontana - Composite - 17 - May 10 8

Esempio codice

```
// Creator provides some Factory methods
public class Creator {
    public static Resource getFolder() {
        return new Folder("myBooks");
    }
    public static Resource getImage() {
        return new Image("car.jpg",80,60);
    }
}

// TestFile uses Composite design pattern
public class TestFile {
    public static void main(String[] args) {
        // create resources, hold Components
        Resource fold = Creator.getFolder();
        Resource img = Creator.getImage();

        // invoke show on a Leaf
        img.show();

        // insert img into Composite fold
        fold.add(img);

        // invoke show on a Composite
        fold.show();
    }
}
```

Diagramma UML classi per l'esempio

