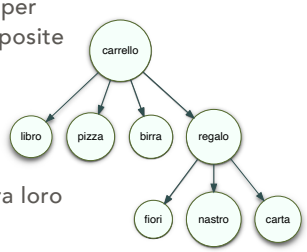
Composite

- **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**
  - E' 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
  - 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

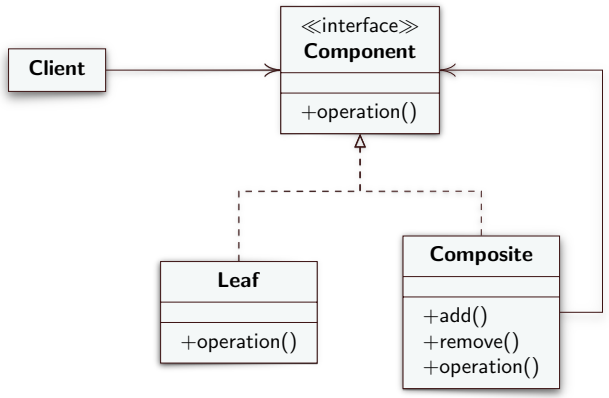
- Esempio 1: File
  - Le operazioni su disco permettono la gestione di file (immagini, testo, etc.) e la gestione di cartelle. Esempi di operazioni: creazione, lettura, scrittura, esecuzione
  - Le classi client devono poter fare operazioni su file e **cartelle**, senza distinguere fra essi
  - Una cartella deve poter contenere al suo interno sia file che altre cartelle, queste a sua volta contengono altri elementi, e così via
- Esempio 2: Figure
  - In un editor di figure ho sia figure semplici, es. Linea, Box, che figure composte, ovvero **raggruppamenti** di figure semplici e composte
  - Le classi client devono poter trattare allo stesso modo sia le figure semplici che quelle composte, quando avviano operazioni come disegna, sposta, etc.

Composite

- **Soluzione**
  - **Component**: interfaccia (o classe abstract) che rappresenta elementi semplici e non; dichiara le operazioni degli oggetti della composizione; implementa le operazioni comuni alle sottoclassi, in modo appropriato; dichiara le operazioni per l'accesso e la gestione degli elementi semplici; può definire un'operazione che permette ad un elemento di accedere all'oggetto padre nella struttura ricorsiva
  - **Leaf**: classe che rappresenta elementi semplici (detti child/ children); è sottoclasse di Component; implementa il comportamento degli oggetti semplici
  - **Composite**: classe che rappresenta elementi contenitori; è sottoclasse di Component; definisce il comportamento per l'aggregato di elementi child; tiene il riferimento a ciascuno degli elementi child; implementa operazioni per gestire elementi child

Composite

- Soluzione



Composite

- Collaborazioni
  - I client usano l'interfaccia di Component per interagire con elementi della struttura composita. Se il ricevente del messaggio è Leaf, la richiesta è gestita direttamente. Se il ricevente è un Composite, questo invia la richiesta ai suoi child e possibilmente avvia operazioni addizionali prima e dopo
- Conseguenze
  - Elementi semplici possono essere composti in elementi più complessi, questi possono essere composti, e così via (ovvero composizione ricorsiva)
  - Un client che si aspetta un elemento semplice può riceverne uno composto
  - I client sono semplici, trattano strutture composte e semplici uniformemente. I client non devono sapere se trattano con un Leaf o un Composite
  - Nuovi tipi di elementi (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 (in base al tipo), invece dovranno essere fatti dei controlli a runtime

Composite

- Ulteriori dettagli
  - Per facilitare la navigazione degli oggetti, gli elementi child mantengono il riferimento all'oggetto che li contiene. Il riferimento è inserito in Component, mentre Leaf e Composite possono implementare le operazioni per gestirlo
  - Per avere client che non distinguono se trattano con 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 di 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 un'operazione add() su Leaf e si ignora la richiesta, questo potrebbe indicare un difetto del programma
    - 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**

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 elementi child deve essere definita in Composite, altrimenti se fosse definita in Component si sprecherebbe spazio, poiché ogni Leaf avrebbe tale variabile anche se non deve usarla mai
- L'ordinamento di 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, è utile quando si implementa un'operazione che deve ricercare tra tutti i suoi componenti child. I componenti child devono poter accedere ad un'operazione che permette di invalidare la cache del Composite

Esempio

- Si vuol gestire un prodotto e un insieme di prodotti (nel carrello) allo stesso modo