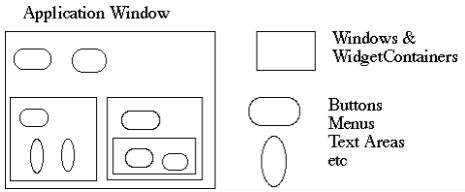
Composite

Pattern strutturale

COMPOSITE

- Intento: manipolazione uniforme di strutture ad oggetti legate da relazioni del tipo tutto-parte: composizione/aggregazione.
- ad es. le interfacce utente



- Ho oggetti «semplici» (foglie) ed oggetti «compositi», contenitori di altri oggetti
- Voglio poter invocare l'operazione che ridisegna (update()) complessivamente tutta l'interfaccia utente come una singola operazione su un unico oggetto

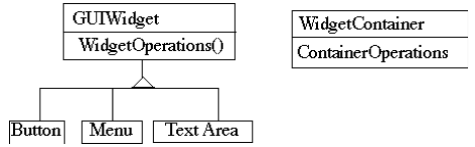

```
class Window {
    Buttons[] myButtons;
    Menus[] myMenus;
    TextAreas[] myTextAreas;
    WidgetContainer[] myContainers;

    public void update() {
        if (myButtons != null)
            for (int k = 0; k < myButtons.length(); k++) myButtons[k].refresh();
        if (myMenus != null)
            for (int k = 0; k < myMenus.length(); k++) myMenus[k].display();
        if (myTextAreas != null)
            for (int k = 0; k < myButtons.length(); k++) myTextAreas[k].refresh();
        if (myContainers != null)
            for (int k = 0; k < myContainers.length(); k++)  myContainers[k].updateElements();
        // etc.
    }

    public void fooOperation() {
        if (myButtons != null);
        // etc.
    }
    //etc...
}
```

Ogni operazione avra' la stessa struttura duplicata: n cicli, uno per tipologia di oggetto.

Uso direttamente le interfacce degli oggetti: la classe dipenderà da tutti i tipi esistenti

Idea: uso il polimorfismo

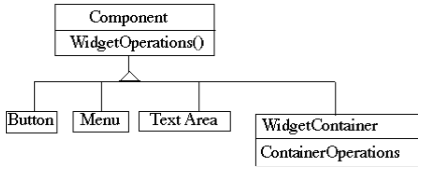
- Con una interfaccia rappresento in astratto tutti i tipi semplici (foglie)
- un solo tipo li rappresenta tutti
- Distinguo ancora tra due categorie: oggetti contenitori e semplici
- Devo comunque iterare esplicitamente gli oggetti delle due tipologie
- Idea migliore: vedere il contenitore come una foglia "speciale"




```
class Window {
    GUIWidgets[] myWidgets;
    WidgetContainer[] myContainers;

    public void update() {
        if (myWidgets != null)
            for (int k = 0; k < myWidgets.length(); k++)
                myWidgets[k].update();
        if (myContainers != null)
            for (int k = 0; k < myContainers.length(); k++)
                myContainers[k].updateElements();
        // etc.
    }
}
```

Composite pattern

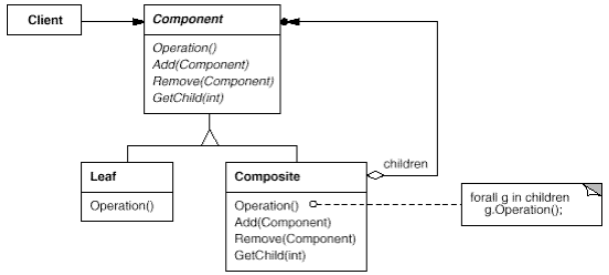
- **Una sola interfaccia** (**Component**) definisce in modo uniforme come interagire con tutti i tipi di oggetto: sia semplici che composti
- può anche implementare dei comportamenti di default
- Gli oggetti semplici (**Leaf**) fanno override delle operazioni di component per specializzarle
- Il **Composite** (widgetContainer) referenzia i suoi molteplici oggetti come istanze tutte di tipo Component
- In Composite specializzo ogni operazione in Component con un semplice ciclo, che invoca l'operazione su tutti i suoi componenti




```
class WidgetContainer extends Component {
    Component[] myComponents;

   //ricorsione object oriented
    public void update() {
        if (myComponents != null)
            for (int k = 0; k < myComponents.length();
k++)
                myComponents[k].update();
    }

   // add container specific operations here
}
```


Struttura del pattern Composite

- L'interfaccia **Component** rappresenta in modo uniforme le **operazioni applicabili all'intera struttura**, ovvero, comuni a tutti gli oggetti semplici
- Il codice client è indipendente dai tipi **Leaf**: Leaf1, Leaf2…LeafN usati nella struttura
- Solo il Composite implementa dei metodi in più per aggiungere o togliere oggetti al suo interno (e similari)



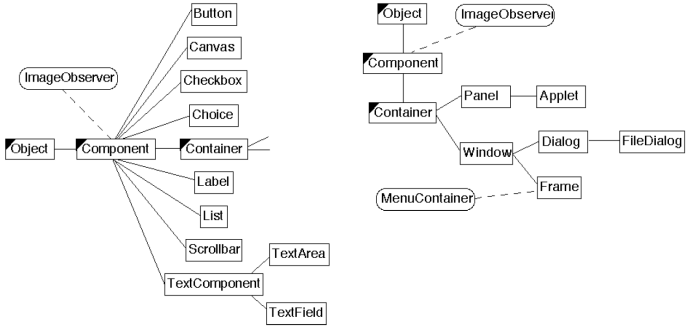
- Composite implementa le operazioni di applicandole **ricorsivamente** agli oggetti al suo interno;

Composite

- **Motivazione**
  - Necessità di **raggruppare** elementi semplici tra loro per formare elementi composti
  - Non **distinguere** tra classi per elementi semplici e classi per contenitori nei client
- **Collaborazioni**
  - I client usano **l'interfaccia 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

Usato in Java

- Lungamente impiegato per le librerie AWT in Java fin dalla versione 1.0



Conseguenze

- Un client che si aspetta un elemento semplice può riceverne uno composto
- I client sono semplici, trattando strutture composte e semplici uniformemente, non sapendo se usano un Leaf o un Composite (polimorfici)
- Nuovi tipi di elementi (Leaf o Composite) possono essere aggiunti e potranno funzionare con la struttura ed i client esistenti (riusabili)
- Supporto qualsiasi tipo di Leaf che implementi l'interfaccia usata dal client (Component), non serve che siano parenti
- Non è possibile a design time vincolare il Composite all'uso solo alcuni tipi di Leaf: posso solo bloccarne l'uso facendo dei controlli a runtime

Implementazione

- **Il Composite può essere anche un ulteriore tipo di Leaf** (ad es. la classe Window ha un titolo, posizione, bordo, dimensioni, etc…) con suoi dati e operazioni
  - Di base è solo un raccoglitore che implementa la gestione dei suoi children (**add-remove)**
- In fase di costruzione della struttura, so quali oggetti sono Composite e possono ricevere messaggi add/remove
- Se dichiarassi add/remove nel Component?
  - Vantaggi: **trasparenza.** Tutti gli oggetti hanno la stessa interfaccia e sono quindi indistinguibili per il client: lo scopo del pattern
  - Svantaggi: **esposizione ad errori a run-time**
    - Consentire di invocare add/remove sulle leaf potrebbe generare un errore a run-time
    - posso gestirlo (**eccezione**), ma avrei potuto segnalarlo in compilazione
    - posso ispezionare il tipo (reflection) dell'oggetto prima di invocarle, ma non ho trasparenza

Implementazione

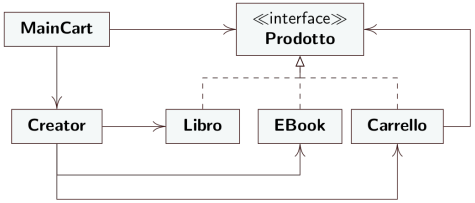
- L'attraversamento di una struttura composita non è trasparente: devo poter distinguere i Leaf dai Composite
  - Si può inserire in **Component** una operazione ***getComposite*() che ritorna null,** ridefinita in **Composite** per **ritornare this**.
  - In Java si può usare la reflection
- La **lista che contiene i child** deve essere definita **in Composite**, non in Component. le Leaf non ne fanno mai uso
- **L'ordinamento dei child** per un Composite potrebbe essere importante in certe implementazioni
- Il Composite potrebbe **implementare una cache**, per ottimizzare le prestazioni
  - Ad es. nella ricerca di un child tra tutti i child locali
  - I child devono poter accedere ad un'operazione che **invalida la cache**

Riferimenti espliciti al padre

- In component posso avere il riferimento all'oggetto padre
- Leaf e Composite implementeranno le operazioni per gestirlo
- Facilita la navigazione a ritroso tra gli oggetti della struttura

```
class Composite extends Component {
    Component[] myComponents;

    public void update() {
        if (myComponents != null)
            for (int k = 0; k < myComponents.length(); k++)
                myComponents[k].update();
    }
    public void add(Component aComponent) {
        myComponents.append(aComponent);
        aComponent.setParent(this);
    }
}

abstract class Component {
    protected Component parent;
    public void setParent(Component myParent) {
        parent = myParent;
    }
    // etc.
}
```

Esempio: AppCart



- Si vuol gestire un carrello di prodotti vari con la stessa semplicità con cui gestirei un solo prodotto
- Implemento il carrello come una struttura composita
- Mi avvalgo di un Creator (factory method) per riempirlo