8. Composite

(Gof pag. 163)

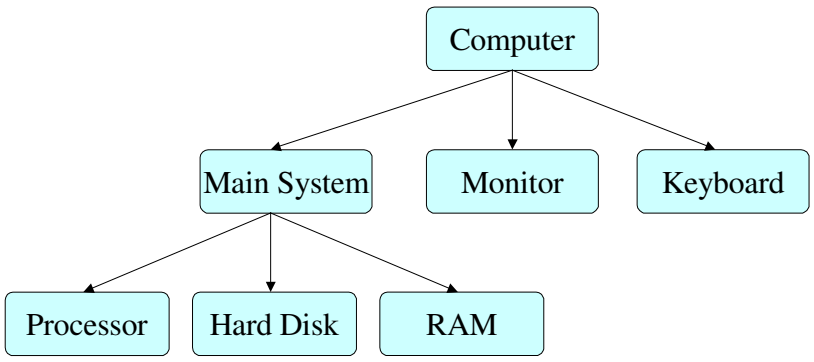
8.1. Descrizione

Consente la costruzione di gerarchie di oggetti composti. Gli oggetti composti possono essere conformati da oggetti singoli, oppure da altri oggetti composti. Questo pattern è utile nei casi in cui si vuole:

- Rappresentare gerarchie di oggetti tutto-parte.
- Essere in grado di ignorare le differenze tra oggetti singoli e oggetti composti.

8.2. Esempio

Nel magazzino di una ditta fornitrice di computer ci sono diversi prodotti, quali computer pronti per la consegna, e pezzi di ricambio (o pezzi destinati alla costruzione di nuovi computer). Dal punto di vista della gestione del magazzino, alcuni di questi pezzi sono pezzi singoli (indivisibili), altri sono pezzi composti da altri pezzi. Ad esempio, il "monitor", la "tastiera" e la "RAM" sono pezzi singoli, intanto il "main system", è un pezzo composto da tre pezzi singoli ("processore", "disco rigido" e "RAM"). Un altro esempio di pezzo composto è il "computer", che si compone di un pezzo composto ("main system"), e due pezzi singoli ("monitor" e "tastiera").

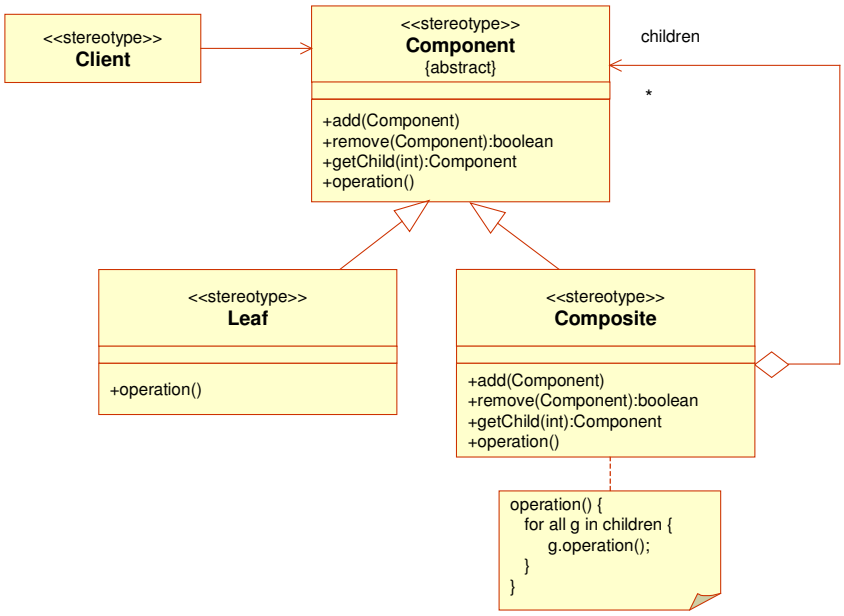


Il problema è la rappresentazione omogenea di tutti gli elementi presenti del magazzino, sia dei singoli componenti, sia di quelli composti da altri componenti.

8.3. Descrizione della soluzione offerta dal pattern

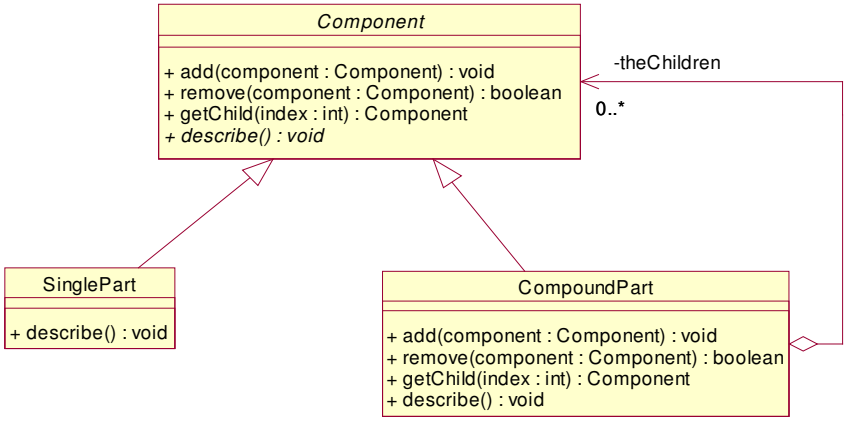
Il pattern “Composite” definisce la classe astratta componente (Component) che deve essere estesa in due sottoclassi: una che rappresenta i singoli componenti (Leaf), e un’altra (Composite) che rappresenta i componenti composti, e che si implementa come contenitore di componenti. Il fatto che quest’ultima sia un contenitore di componenti, li consente di immagazzinare al suo interno, sia componenti singoli, sia altri contenitori (dato che entrambi sono stati dichiarati come sottoclassi di componenti).

8.4. Struttura del Pattern



8.5. Applicazione del pattern

Schema del modello



Partecipanti

- **Component**: classe astratta Component.
  - Dichiara una interfaccia comune per oggetti singoli e composti.
  - Implementa le operazioni di default o comuni tutte le classi.
- **Leaf**: classe SinglePart**.**
  - Estende la classe Component, per rapperesentare gli oggetti che non sono composti (foglie).
  - Implementa le operazioni per questi oggetti.
- **Composite**: classe CompoundPart.
  - Estende la classe Component, per rappresentare gli oggetti che sono composti.
  - Immagazzina al suo interno i propri componenti.
  - Implementa le operazioni proprie degli oggetti composti, e particolarmente quelle che riguardano la gestione dei propri componenti.
- **Client**: in questo esempio sarà il programma principale quello che farà le veci di cliente.
  - Utilizza gli oggetti singoli e composti tramite l'interfaccia rappresentata dalla classe astratta Component.

Descrizione del codice

La classe astratta `Component` definisce l'interfaccia comune di oggetti singoli e composti, e implementa le loro operazioni di default. Particolarmente le operazioni `add(Component c)` e `remove(Component c)` sollevano una eccezione del tipo `SinglePartException` se vengono invocate su un oggetto foglia (tentativo di aggiungere o rimuovere un componente). Invece nel caso di `getChild(int n)`, che serve a restituire il componente di indice `n`, l'operazione di default restituisce `null` (questa è stata una scelta di progettazione, un'altra possibilità era sollevare anche in questo caso una eccezione)[9]. Il metodo `describe()` è dichiarato come metodo astratto, da implementare in modo particolare nelle sottoclassi. Il Costruttore di `Component` riceve una stringa contenete il nome del componente, che verrà assegnato ad ognuno di essi.

```
public abstract class Component {

  public String name;

  public Component(String aName){
      name = aName;
  }

  public abstract void describe();
```

[9] Questo modo di trattare eventuali tentativi di l'invocazione di metodi legati a oggetti composti, sulle foglie, è anche applicato da Landini [10].

```
  public void add(Component c) throws SinglePartException {
      if (this instanceof SinglePart)
         throw new SinglePartException( );
  }

  public void remove(Component c) throws SinglePartException{
      if (this instanceof SinglePart)
         throw new SinglePartException( );
  }

  public Component getChild(int n){
      return null;
  }

}
```

La classe `SinglePart` estende la classe `Component`. Possiede un costruttore che consente l'assegnazione del nome del singolo pezzo, il quale che verrà immagazzinato tramite l'invocazione al costruttore della superclasse. La classe `SinglerPart` fornisce, anche, l'implementazione del metodo `describe()`.

```
public class SinglePart extends Component {

  public SinglePart(String aName) {
      super(aName);
  }

  public void describe(){
      System.out.println( "Component: " + name );
  }

}
```

La classe `CompoundPart` estende anche `Component`, e implementa sia i metodi di gestione dei componenti (`add`, `remove`, `getChild`), sia il metodo `describe()`. Si noti che il metodo `describe()` stampa in primo luogo il proprio nome dell'oggetto, e poi scandisce l'elenco dei suoi componenti, invocando il metodo `describe()` di ognuno di essi. Il risultato sarà che insieme alla stampa del proprio nome dell'oggetto composto, verranno anche stampati i nomi dei componenti.

```
import java.util.Vector;
import java.util.Enumeration;

public class CompoundPart extends Component {

  private Vector children ;

  public CompoundPart(String aName) {
      super(aName);
      children = new Vector();
  }

  public void describe(){

      System.out.println("Component: " + name);
      System.out.println("Composed by:");
      System.out.println("{");

        int vLength = children.size();
        for( int i=0; i< vLength ; i ++ ) {
         Component c = (Component) children.get( i );
                c.describe();
      }
      System.out.println("}");
  }

  public void add(Component c) throws SinglePartException {
      children.addElement(c);
  }
```

```
  public void remove(Component c) throws SinglePartException{
      children.removeElement(c);
  }

  public Component getChild(int n) {
      return (Component)children.elementAt(n);
  }

}
```

Si noti che in questa implementazione ogni `CompoundPart` gestisce i propri componenti in un `Vector`.

La classe `SinglePartException` rappresenta l'eccezione che verrà sollevata nel caso che le operazioni di gestione dei componenti vengano invocate su una parte singola.

```
class SinglePartException extends Exception {
    public SinglePartException( ){
      super( "Not supported method" );
  }
}
```

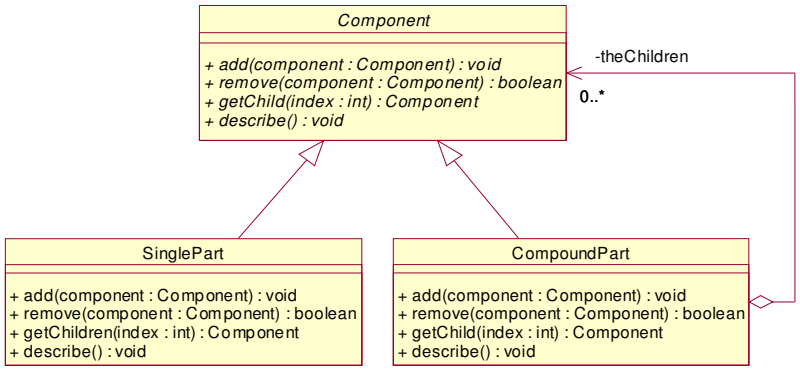
L'applicazione `CompositeExample` fa le veci del **Client** che gestisce i diversi tipi di pezzi, tramite l'interfaccia comune fornita dalla classe `Component`. Nella prima parte dell'esecuzione si creano dei pezzi singoli (`monitor`, `keyboard`, `processor`, `ram` e `hardDisk`), dopodiché viene creato un oggetto composto (`mainSystem`) con tre di questi oggetti singoli. L'oggetto composto appena creato serve, a sua volta, per creare, insieme ad altri pezzi singoli, un nuovo oggetto composto (`computer`). L'applicazione invoca poi il metodo `describe()` su un oggetto singolo, sull'oggetto composto soltanto da pezzi singoli, e sull'oggetto composto da pezzi singoli e pezzi composti. Finalmente fa un tentativo di aggiungere un componente ad un oggetto corrispondente a un pezzo singolo.

```
public class CompositeExample {

  public static void main(String[] args) {

    // Creates single parts
    Component monitor     = new SinglePart("LCD Monitor");
    Component keyboard    = new SinglePart("Italian Keyboard");
    Component processor   = new SinglePart("Pentium III Processor");
    Component ram         = new SinglePart("256 KB RAM");
    Component hardDisk    = new SinglePart("40 Gb Hard Disk");

    // A composite with 3 leaves
    Component mainSystem = new CompoundPart( "Main System" );
    try {
      mainSystem.add( processor );
      mainSystem.add( ram );
      mainSystem.add( hardDisk );
    }
    catch (SinglePartException e){
      e.printStackTrace();
    }

    // A Composite compound by another Composite and one Leaf
    Component computer = new CompoundPart("Computer");
    try{
        computer.add( monitor );
        computer.add( keyboard );
      computer.add( mainSystem );
    }
    catch (SinglePartException e){
      e.printStackTrace();
    }
```

```
    System.out.println("**Tries to describe the 'monitor' component");
    monitor.describe();
    System.out.println("**Tries to describe the 'main system' component"
                       );
    mainSystem.describe();
    System.out.println("**Tries to describe the 'computer' component" );
    computer.describe();

    // Wrong: invocation of add() on a Leaf
    System.out.println( "**Tries to add a component to a single part
                         (leaf) " );
    try{
      monitor.add( mainSystem );
    }
    catch (SinglePartException e){
      e.printStackTrace();
    }

  }

}
```

Osservazioni sull'esempio

Si noti che nell'esempio presentato, la classe astratta `Component` fornisce un'implementazione di default per i metodi di gestione dei componenti (`add`, `remove`, `getChild`). Dal punto di vista del *Composite pattern*, sarebbe anche valida la dichiarazione di questi metodi come metodi astratti, lasciando l'implementazione alle classi `SinglePart` e `CompoundPart`, come si può apprezzare nella seguente figura:



Se si implementa il pattern in questo modo, si devono modificare le classi `Component` e `SinglePart`. In particolare, il codice della classe `Component` dovrebbe dichiarare i metodi di gestione dei componenti (`add`, `remove` e `getChild`), come metodi astratti:

```
public abstract class Component {

  public String name;

  public Component(String aName){
      name = aName;
  }

  public abstract void describe();

  public abstract void add(Component c) throws SinglePartException;
```

```
  public abstract void remove(Component c) throws SinglePartException;

  public abstract Component getChild(int n);

}
```

E la classe `SinglePart` dovrebbe implementare il codice riguardante tutti i metodi dichiarati astratti nella superclasse:

```
public class SinglePart extends Component  {

  public SinglePart(String aName) {
      super(aName);
  }

  public void add(Component c) throws SinglePartException{
      throw new SinglePartException( );
  }

  public void remove(Component c) throws SinglePartException{
      throw new SinglePartException( );
  }

  public Component getChild(int n) {
      return null;
  }

  public void describe(){
      System.out.println( "Component: " + name );
  }
}
```

Esecuzione dell'esempio

```
C:\Design Patterns\Structural\Composite >java CompositeExample

** Tries to describe the 'monitor' component
Component: LCD Monitor

** Tries to describe the 'main system' component
Component: Main System
Composed by:
{
Component: Pentium III Processor
Component: 256 KB RAM
Component: 40 Gb Hard Disk
}

** Tries to describe the 'computer' component
Component: Computer
Composed by:
{
Component: LCD Monitor
Component: Italian Keyboard
Component: Main System
Composed by:
{
Component: Pentium III Processor
Component: 256 KB RAM
Component: 40 Gb Hard Disk
}
}

** Tries to add a component to a single part (leaf)
SinglePartException: Not supported method
        at SinglePart.add(SinglePart.java:9)
        at CompositeExample.main(CompositeExample.java:46)
```

8.6. Osservazioni sull'implementazione in Java

Non ci sono aspetti particolari da tenere in conto.