9. Decorator

(GoF pag. 117)

9.1. Descrizione

Aggiunge dinamicamente responsabilità addizionali ad un oggetto. In questo modo si possono estendere le funzionalità d'oggetti particolari senza coinvolgere complete classi.

9.2. Esempio

Si pensi ad un modello di oggetti che rappresenta gli impiegati (Employee) di una azienda. Tra gli impiegati, ad esempio, esistono gli Ingegneri (Engineer) che implementano le operazioni definite per gli impiegati, secondo le proprie caratteristiche.

Il sistema comprende la possibilità di investire gli impiegati con delle responsabilità aggiuntive, ad esempio, quando un impiegato diventa capoufficio (Administrative Manager), oppure, quando viene assegnato alla direzione di un progetto (Project Manager), essendo entrambe responsabilità non esculenti tra di loro.

Questi cambiamenti di tipologia di alcuni impiegati coinvolgono modifiche delle responsabilità definite per gli oggetti, alterandone le esistenti o aggiungendone nuove. Per questa ragione, sarebbe di interesse definire un modo per aggiungere dinamicamente nuove responsabilità ad oggetto specifico, eventualmente con la ulteriore possibilità di toglierle.

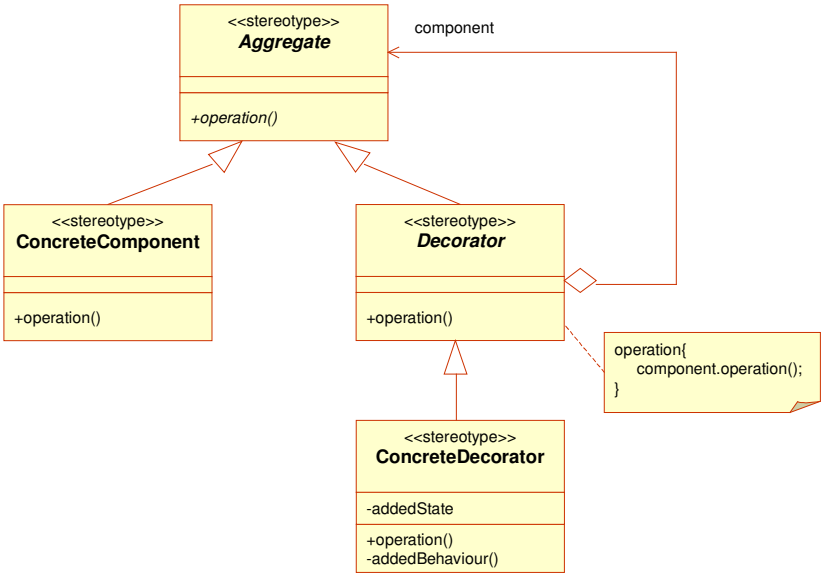
9.3. Descrizione della soluzione offerta dal pattern

Il pattern suggerisce la creazione di *wrapper classes* (Decorator) che racchiudono gli oggetti ai quali si vuole aggiungere le nuove responsabilità. Questi ultimi oggetti, insieme ai Decorator devono implementare una interfaccia comune, in modo che l'applicazione possa continuare ad interagire con gli oggetti decorati.

Per una stessa interfaccia possono esserci più Decorator, ad esempio, per investire i ruoli di capoufficio e di responsabile di un progetto.

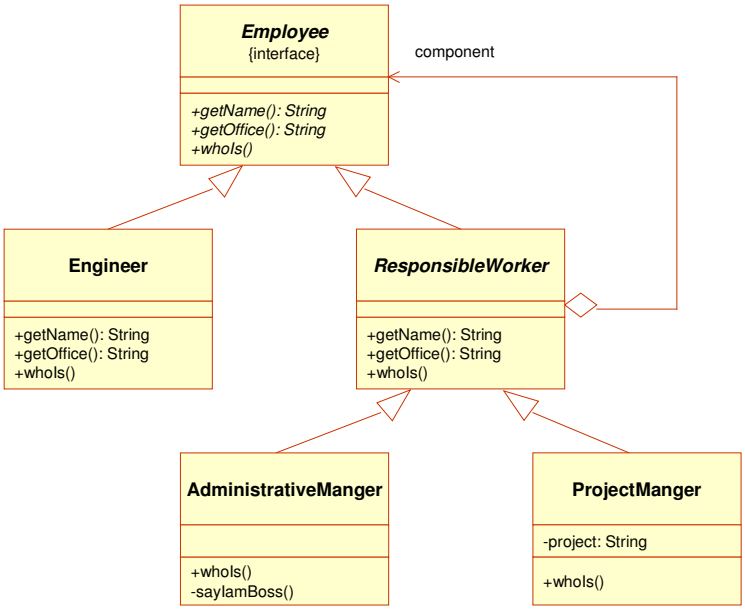
Il fatto che Decorator e oggetti decorati implementino la stessa interfaccia, consente anche l'applicazione di un Decorator ad un altro oggetto già decorato, ottenendo in questo modo la sovrapposizione di funzioni (ad esempio, un impiegato potrebbe essere investito come capoufficio e responsabile di un progetto contemporaneamente).

9.4. Struttura del Pattern



9.5. Applicazione del pattern

Schema del modello



Partecipanti

- **Component**: classe Employee.
  - Specifica l'interfaccia degli oggetti che possono avere delle responsabilità aggiunte dinamicamente.
- **ConcreteComponent**: classe Engineer.

  - Implementa l'oggetto in cui si possono aggiungere nuove responsabilità.
- **Decorator**: classe ResponsibleWorker.
  - Possiede un riferimento all'oggetto Component e specifica una interfaccia concordante con l'interfaccia Component.
- **ConcreteDecorator**: classe AdministrativeManager e ProjectManager.
  - Aggiunge nuove responsabilità al Component.

Descrizione del codice

Tutti i componenti di questo modello implementano l'interfaccia `Employee`, che specifica soltanto tre metodi: uno per restituire il proprio nome (`getName`), un altro per restituire il nome dell'ufficio al quale appartiene (`getOffice`), e un ultimo che fa la presentazione di se stesso (`whoIs`).

```
public interface Employee {

  public String getName();
  public String getOffice();
  public void whoIs();

}
```

La classe `Engineer` implementa l'interfaccia `Employee`:

```
public class Engineer implements Employee {

   private String name, office;

   public Engineer( String nam, String off ) {
      name    = nam;
      office  = off;
   }

   public String getName() {
      return name ;
   }

   public String getOffice() {
      return office ;
   }

   public void whoIs() {
      System.out.println( "I am " + getName() + ", and I am with the "
                          + getOffice() +".");
   };

}
```

La classe astratta `ResponsibleWorker` corrisponde al **Decorator** del modello. Contiene il codice necessario per immagazzinare al suo interno l'oggetto decorato (**Component**), e mappa verso di lui le operazioni richieste. Si noti che questa classe implementa l'intrerfaccia `Employee`, e al suo interno utilizza questa stessa interfaccia per comunicare con il **Component**.

```
abstract class ResponsibleWorker implements Employee {

    protected Employee responsible;
```

```
    public ResponsibleWorker(Employee employee) {
        responsible = employee;
    }
    public String getName() {
      return responsible.getName();
    }

    public String getOffice() {
      return responsible.getOffice();
    }

    public void whoIs() {
      responsible.whoIs();
    }

}
```

La responsabilità particolare che riguarda le funzioni di un capoufficio sono codificate nell'`AdministrativeManager`. Questa classe estende le funzioni del **Decorator**, particolarmente aggiungendo l'operazione `sayIamBoss`, che viene chiamata come parte della ridefinizione del metodo `whoIs` :

```
public class AdministrativeManager extends ResponsibleWorker {

   public AdministrativeManager( Employee empl ) {
      super( empl );
   }

   public void whoIs() {
       sayIamBoss();
       super.whoIs();
   }

   private void sayIamBoss(){
       System.out.print( "I am a boss. " );
   }

}
```

La classe `ProjectManager`, invece estende le variabili di stato dell'oggetto decorato, e modifica il comportamento dell'oggetto (metodo `whoIs`):

```
public class ProjectManager extends ResponsibleWorker {

   private String project;

   public ProjectManager( Employee empl, String proj ) {
     super( empl );
     project = proj;
   }

   public void whoIs() {
     super.whoIs();
     System.out.println( "I am the Manager of the Project:" + project );
   }

}
```

Finalmente si presenta l'applicazione che crea un `Employee`, lo investe come `AdministrativeManager` (per indicare che l'impiegato è capoufficio), e poi, due volte come `ProjectManager` (per segnare che l'impiegato è anche responsabile della gestione di due progetti):

```
public class DecoratorExample1 {
```

```
  public static void main(String arg[]) {

     Employee thisWillBeFamous = new Engineer( "William Gateway",
                                               "Programming Department" );
     System.out.println( "Who are you?");
     thisWillBeFamous.whoIs();

     thisWillBeFamous = new AdministrativeManager( thisWillBeFamous );
     System.out.println( "Who are you now?");
     thisWillBeFamous.whoIs();

     thisWillBeFamous = new ProjectManager( thisWillBeFamous,
                                      "D.O.S.- Doors Operating System" );
     System.out.println( "Who are you now?");
     thisWillBeFamous.whoIs();

     thisWillBeFamous = new ProjectManager( thisWillBeFamous,
                                    "EveryoneLoggedToInternet Explorer" );
     System.out.println( "Who are you now?");
     thisWillBeFamous.whoIs();

  }

}
```

Osservazioni sull'esempio

Si noti che in questo esempio furono creati **ConcreteDecorators** che:

- Estendono operazioni esistenti del **Component** (metodo `sayIamBoss` del `AdministrativeManager`).
- Estendono lo stato del Component (attributo `project` del `ProjectManager`).

Deve notarsi che prima un **ConcreteDecorator** fu applicato su un **ConcreteComponent** (`AdministrativeManager` su `Engineer`). Poi su quel **ConcreteDecorator** fu applicato un altro **ConcreteDecorator** (`ProjectManager`). Finalmente, su quest'ultimo venne applicato un'altra volta un'altra istanza dello stesso **ConcreteDecorator**.

Esecuzione dell'esempio

```
C:\Design Patterns\Structural\Decorator\Exmple1>java DecoratorExample1

Who are you?
I am William Gateway, and I am with the Programming Department.

Who are you now?
I am a boss. I am William Gateway, and I am with the Programming
Department.

Who are you now?
I am a boss. I am William Gateway, and I am with the Programming
Department.
I am the Manager of the Project: D.O.S.- Doors Operating System

Who are you now?
I am a boss. I am William Gateway, and I am with the Programming
Department.
I am the Manager of the Project: D.O.S.- Doors Operating System
I am the Manager of the Project: EveryoneLoggedToInternet Explorer
```

Un altro Esempio

Un uso che può avere un **Decorator** è sincronizzare un oggetto costruito originalmente per essere utilizzato in un ambiente single-threading, quando si vuol portare a un ambiente d'esecuzione multi-threading.

Per esemplificare questo uso si ha definito la interfaccia `DiagonalDraggablePoint` (**Component**) che fornisce la firma dei metodi per muovere un punto diagonalmente una distanza specificata, e per scrivere la posizione attuale.

```
public interface DiagonalDraggablePoint {

     public void moveDiagonal( int distance, String draggerName );
     public void currentPosition( );

}
```

La classe `SequentialPoint` (**ConcreteComponent**) implementa la citata interfaccia. Si noti che il metodo `moveDiagonal` soltanto aggiunge lo stesso valore alle variabili d'istanza `x` e `y`, e stampa questi valori prima di modificarli. Questo significa che sempre `x` e `y` dovrebbero essere numericamente uguali.

```
public class SequentialPoint implements DiagonalDraggablePoint {

     private int x, y;

     public SequentialPoint( ) {
         this.x = 0;
         this.y = 0;
     }

     public void moveDiagonal( int distance, String draggerName ) {
         int aux = x + distance ;
         System.out.println( "Moved by " + draggerName +
                             " - Origin x=" + x + " y=" + y );
         x = aux;
         y = y + distance;
     }


     public void currentPosition( ) {
         System.out.println( "Current position : x=" + x + " y=" + y );
     }


}
```

In una esecuzione multi-threaded il codice anteriore ha il gravissimo problema che non blocca l'oggetto nel processo di movimento (`moveDiagonal`), così che se il thread in esecuzione è interrotto en questa fase, alcune delle variabili potrebbero essere aggiornate da un altro thread, prima che il controllo ritornassi, ottenendosi come risultato valori diversi di `x` e `y`. Il problema si può risolvere creando un **Decorator** (`SynchronizedPoint`) che implementi l'interfaccia `DiagonalDraggablePoint` e che sincronizzi il codice rischioso:

```
public class SynchronizedPoint implements DiagonalDraggablePoint {

     DiagonalDraggablePoint theSequentialPoint;

     public SynchronizedPoint(DiagonalDraggablePoint np) {
```

```
        theSequentialPoint = np;
     }


     public void moveDiagonal( int distance, String draggerName ) {
         synchronized(theSequentialPoint) {
         theSequentialPoint.moveDiagonal( distance, draggerName );
         }
      }

     public void currentPosition( ) {
        theSequentialPoint.currentPosition();
     }

}
```

L'applicazione `DecoratorExample2` fa una prima prova d'uso con l'oggetto sequenziale che è contemporaneamente mosso da due thread. In una seconda fase si fa la stessa prova, ma adesso con l'oggetto sincronizzato:

```
public class DecoratorExample2 {

  public static void main( String[] arg ) {

     System.out.println( "Non synchronized point:" );
     DiagonalDraggablePoint p = new SequentialPoint();
     PointDragger mp1 = new PointDragger( p, "Thread 1" );
     PointDragger mp2 = new PointDragger( p, "Thread 2" );
     Thread t1 = new Thread( mp1 );
     Thread t2 = new Thread( mp2 );
     t1.start();
     t2.start();
     while( t1.isAlive() || t2.isAlive()  );
     p.currentPosition();

     System.out.println( "Synchronized point:" );
     p = new SynchronizedPoint( new SequentialPoint() );
     mp1 = new PointDragger( p, "Thread 1" );
     mp2 = new PointDragger( p, "Thread 2" );
     t1 = new Thread( mp1 );
     t2 = new Thread( mp2 );
     t1.start();
     t2.start();
     while( t1.isAlive() || t2.isAlive()  );
     p.currentPosition();

  }

}

class PointDragger implements Runnable {
  DiagonalDraggablePoint point;
  String name ;
  public PointDragger( DiagonalDraggablePoint p, String nom ) {
     point = p;
     name  = nom;
  }
  public void run() {
    for( int i=1; i < 5; i++ ) {
       point.moveDiagonal( 1, name );
    }
  }
}
```

Esecuzione dell'esempio

L'esecuzione del programma mostra che l'oggetto non protetto dal meccanismo di blocco, raggiunge uno stato inconsistente, che in questo caso si è raggiunto quando il secondo thread, nel suo secondo movimento mostra che i valori di `x` e `y` sono diversi. L'utilizzo, invece, del oggetto protetto consente di osservare che questo mai raggiunge uno stato inconsistente.

```
C:\Design Patterns\Structural\Decorator\Exmpl2>java DecoratorExample2

Non synchronized point:
Moved by Thread 1 - Origin x=0 y=0
Moved by Thread 1 - Origin x=1 y=1
Moved by Thread 2 - Origin x=1 y=1
Moved by Thread 1 - Origin x=2 y=2
Moved by Thread 2 - Origin x=2 y=3
Moved by Thread 1 - Origin x=3 y=4
Moved by Thread 2 - Origin x=3 y=5
Moved by Thread 2 - Origin x=4 y=7
Current position : x=5 y=8

Synchronized point:
Moved by Thread 1 - Origin x=0 y=0
Moved by Thread 1 - Origin x=1 y=1
Moved by Thread 2 - Origin x=2 y=2
Moved by Thread 1 - Origin x=3 y=3
Moved by Thread 2 - Origin x=4 y=4
Moved by Thread 1 - Origin x=5 y=5
Moved by Thread 2 - Origin x=6 y=6
Moved by Thread 2 - Origin x=7 y=7
Current position : x=8 y=8
```

9.6. Osservazioni sull'implementazione in Java

Java utilizza Decorator simili al presentato in questo ultimo esempio per la sincronizzazione delle collezioni.