6. Adapter

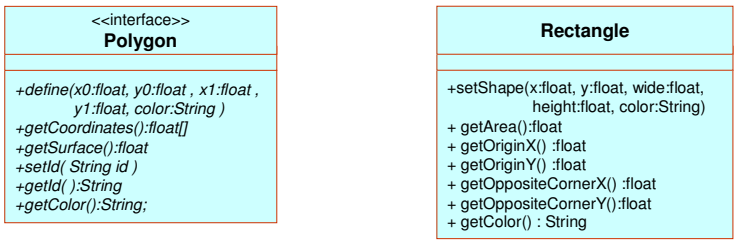
(GoF pag. 141)

6.1. Descrizione

Converte l'interfaccia di una classe in un'altra interfaccia aspettata dai clienti. In questo modo, si consente la collaborazione tra classi che in un altro modo non potrebbero interagire dovuto alle loro diverse interfacce.

6.2. Esempio

Si vuole sviluppare un'applicazione per lavorare con oggetti geometrici. Questi oggetti saranno gestiti dall'applicazione tramite un'interfaccia particolare (Polygon), che offre un insieme di metodi che gli oggetti grafici devono implementare. A questo punto si ha a disposizione una antica classe (Rectangle) che si potrebbe riutilizzare, che però ha un'interfaccia diversa, e che non si vuole modificare.



New interface Available class

Il problema consiste nella definizione di un modo di riutilizzare la classe esistente tramite una nuova interfaccia, ma senza modificare l'implementazione originale.

6.3. Descrizione della soluzione offerta dal pattern

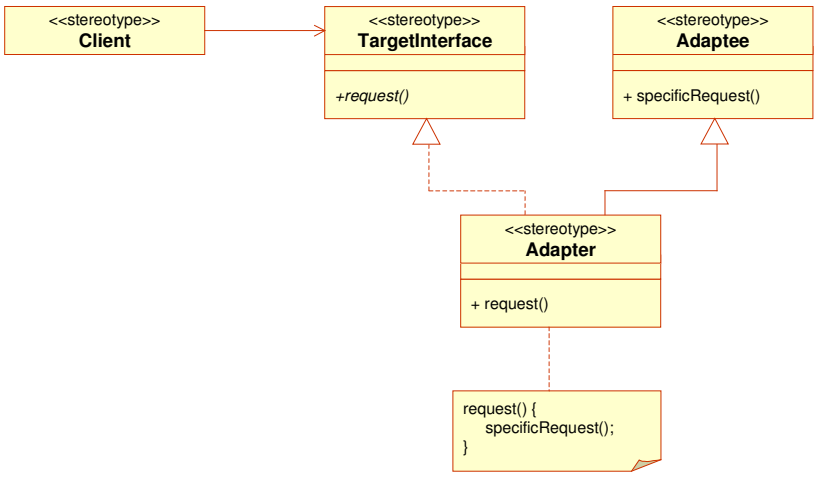
L'*Adapter* pattern offre due soluzioni possibili, denominate *Class Adapter* e *Object Adapter*, che si spiegano di seguito:

- Class Adapter: la classe esistente si estende in una sottoclasse (RectangleClassAdapter) che implementa la desiderata interfaccia. I metodi della sottoclasse mappano le loro operazioni in richieste ai metodi e attributi della classe di base.

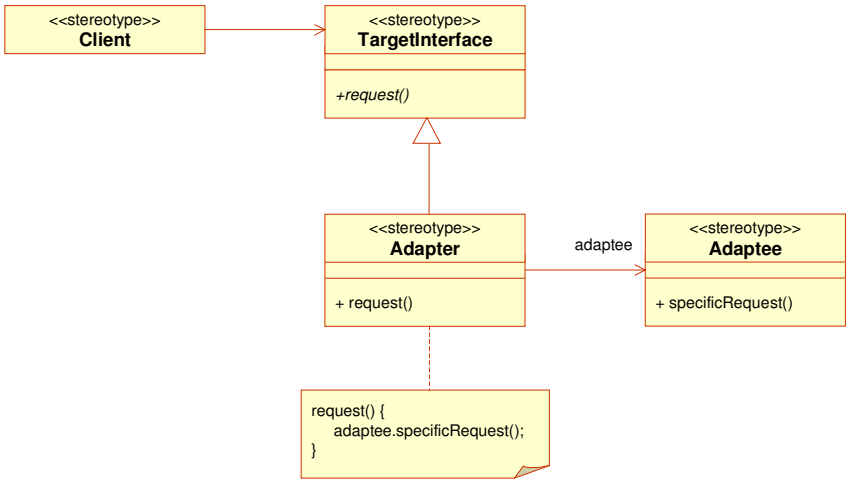
- Object Adapter: si crea una nuova classe (RectangleObjectAdapter) che implementa l'interfaccia richiesta, e che possiede al suo interno un'istanza della classe a riutilizzare. Le operazioni della nuova classe fanno invocazioni ai metodi dell'oggetto interno.

6.4. Struttura del Pattern

Per il *Class Adapter*:

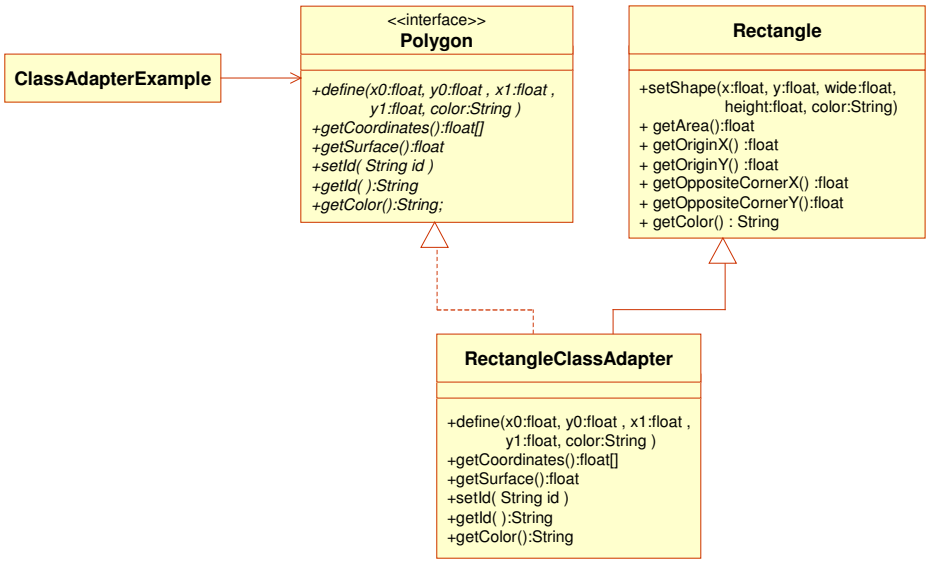


Per l'*Object Adapter*:

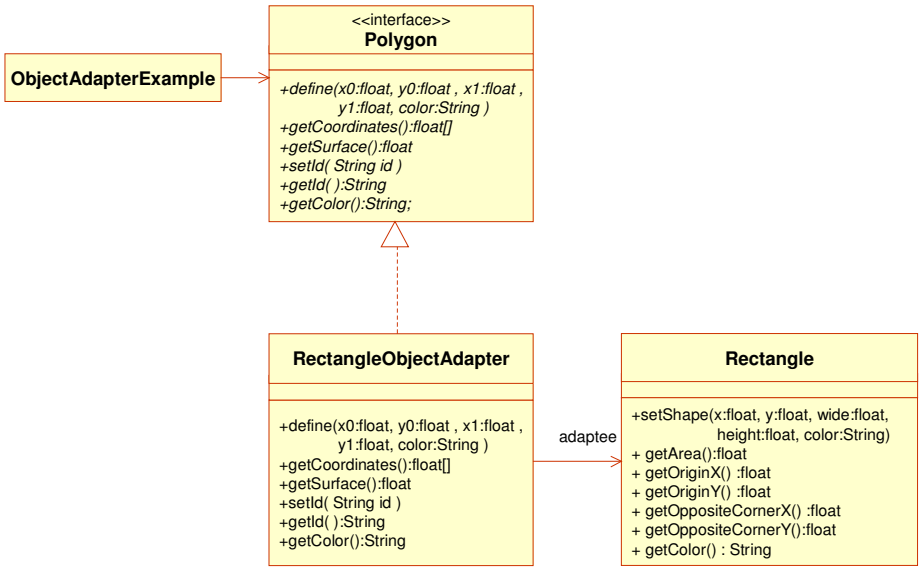


Schema del modello

Nel caso del Class Adapter il modello da implementare è il seguente:



Invece, se viene utilizzato l'Object Adapter, il modello corrisponde a:



Partecipanti

- **TargetInterface**: interfaccia Polygon.
  - Specifica l'interfaccia che il **Client** utilizza.
- **Client**: classi ClassAdapterExample e ObjectAdapterExample.
  - Comunica con l'oggetto interessato tramite la **TargetInterface**.
- **Adaptee**: classe Rectangle.
  - Implementa una interfaccia che deve essere adattata.
- **Adapter**: classi RectangleClassAdapter e RectangleObjectAdapter.
  - Adatta l'interfaccia dell'**Adaptee** verso la **TargetInterface**.

Implementazione

Si presenteranno esempi di entrambi tipi d'*Adapter*. In ogni caso il problema sarà la riutilizzazione della classe `Rectangle` in un'applicazione che crei oggetti che implementano l'interfaccia `Polygon`.

La classe `Rectangle` è implementata in questo modo:

```
public class Rectangle {

    private float x0, y0;
    private float  height, width;
    private String color;

    public void setShape(float x, float y, float a, float l, String c) {
        x0 = x;
        y0 = y;
        height = a;
        width  = l;
        color  = c;
    }

    public float getArea() {
        return x0 * y0;
    }

    public float getOriginX() {
        return x0;
    }

    public float getOriginY() {
        return y0;
    }

    public float getOppositeCornerX() {
        return x0 + height;
    }

    public float getOppositeCornerY() {
        return y0 + width;
    }

    public String getColor() {
        return color;
    }

}
```

Invece l'interfaccia `Polygon` si deve utilizzare è questa:

```
public interface Polygon {

   public void define( float x0, float y0, float x1, float y1,
                       String color );
   public float[] getCoordinates() ;
   public float getSurface();
   public void setId( String id );
   public String getId( );
   public String getColor();

}
```

Si osservi che le caratteristiche del rettangolo (classe `Rectangle`) vengono indicate nel metodo `setShape`, che riceve le coordinate del vertice superiore sinistro, l'altezza, la larghezza e il colore. Dall'altra

parte, l'interfaccia `Polygon` specifica che la definizione delle caratteristiche della figura, avviene tramite il metodo chiamato `define`, che riceve le coordinate degli angoli opposti e il colore.

Si osservi che il metodo `getCoordinates` dell'interfaccia `Polygon` restituisce un array contenente le coordinate degli angoli opposti (nel formato {x0, y0, x1, y1}), intanto nella classe `Rectangle` esistono metodi particolari per ricavare ogni singolo valore (`getOriginX`, `getOriginY`, `getOppositeCornerX` e `getOppositeCornerY`).

In quel che riguarda la superficie del rettangolo, in entrambi casi si ha a disposizione un metodo, ma con nome diverso.

Si noti, anche, che `Polygon` aggiunge metodi non mappabili sulla classe `Rectangle` (`setId` e `getId`), che consentono la gestione di un identificativo tipo `String` per ogni figura creata.

Finalmente si noti che il metodo `getColor` ha la stessa firma e funzione nell'interfaccia `Polygon` e nella classe `Rectangle`.

Di seguito si descrivono le due implementazioni proposte.

a) Implementazione come Class Adapter

La costruzione del *Class Adapter* per il `Rectangle` è basato nella sua estensione. Per questo obbiettivo viene creata la classe `RectangleClassAdapter` che estende `Rectangle` e implementa l'interfaccia `Polygon`:

```
public class RectangleClassAdapter extends Rectangle implements Polygon{

   private String name = "NO NAME";

   public void define( float x0, float y0, float x1, float y1,
                       String color ) {
       float a = x1 - x0;
       float l = y1 - y0;
       setShape( x0, y0, a,  l, color );
   }

   public float getSurface() {
       return getArea();
   }

   public float[] getCoordinates() {
       float aux[] = new float[4];
       aux[0] = getOriginX();
       aux[1] = getOriginY();
       aux[2] = getOppositeCornerX();
       aux[3] = getOppositeCornerY();
       return aux;
   }

   public void setId( String id ) {
      name = id;
   }

   public String getId( ) {
      return name;
   }

}
```

Si noti che la funzionalità riguardante l'identificazione del rettangolo, non presenti nella classe di base, si implementa completamente nella classe **Adapter**.

Le altre funzionalità si ottengono fornendo le operazioni particolari necessarie che richiamando i metodi della classe `Rectangle`.

Si noti che il metodo `getColor` è ereditato dalla classe di base.

Il **Client** di questo rettangolo adattato è la classe `ClassAdapterExample`:

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

       Polygon block = new RectangleClassAdapter();
         block.setId( "Demo" );
         block.define( 3 , 4 , 10, 20, "RED" );
         System.out.println( "The area of "+ block.getId() + " is "+
                             block.getSurface() + ", and it's " +
                             block.getColor() );

    }
}
```

b) Implementazione come Object Adapter

La costruzione dell'Object Adapter per il `Rectangle`, si basa nella creazione di una nuova classe (`RectangleObjectAdapter`) che avrà al suo interno un'oggeto della classe `Rectangle`, e che implementa l'interfaccia `Polygon`:

```
public class RectangleObjectAdapter implements Polygon {

    Rectangle adaptee;
    private String name = "NO NAME";

    public RectangleObjectAdapter() {
        adaptee = new Rectangle();
    }

   public void define( float x0, float y0, float x1, float y1,
                       String col ) {
       float a = x1 - x0;
       float l = y1 - y0;
       adaptee.setShape( x0, y0, a, l, col);
   }

   public float getSurface() {
       return adaptee.getArea();
   }

   public float[] getCoordinates() {
       float aux[] = new float[4];
       aux[0] = adaptee.getOriginX();
       aux[1] = adaptee.getOriginY();
       aux[2] = adaptee.getOppositeCornerX();
       aux[3] = adaptee.getOppositeCornerY();
       return aux;
   }

   public void setId( String id ) {
      name = id;
   }
```

```
public String getId( ) {
      return name;
   }

   public String getColor( ) {
      return adaptee.getColor();
   }


}
```

Si noti come in questo caso la costruzione di un `RectangleObjectAdapter` porta con se la creazione al suo interno di un oggetto della classe `Rectangle`. Ecco il codice del **Client** (`ObjectAdapterExample`) che fa uso di questo **Adapter**:

```
public class ObjectAdapterExample {

    public static void main( String[] arg ) {

       Polygon block = new RectangleObjectAdapter();
         block.setId( "Demo" );
         block.define( 3 , 4 , 10, 20, "RED" );
         System.out.println( "The area of "+ block.getId() + " is "+
                             block.getSurface() + ", and it's " +
                             block.getColor() );

    }

}
```

Osservazioni sull'esempio

In questo esempio si ha dimostrato l'*Adapter* pattern, considerando un caso nel quale l'interfaccia richiesta:

- Ha un metodo la cui funzionalità si può ottenere dall'**Adaptee**, previa esecuzione di alcune operazioni (metodo `define`).
- Ha un metodo la cui funzionalità si può ottenere dall'**Adaptee** tramite l'invocazione di un'insieme dei suoi metodi (`getCoordinates`).
- Ha un metodo la cui funzionalità si ricava direttamente da un metodo dell'**Adaptee**, che ha soltanto una firma diversa (metodo `getSurface`).
- Ha un metodo la cui funzionalità si ricava direttamente da un metodo dell'**Adaptee**, e con la stessa firma (`getColor`).
- Ha metodi che aggiungono nuove operazioni che non si ricavano dai metodi dell'**Adaptee** (`setId` e `getId`).

Esecuzione dell'esempio

```
C:\Design Patterns\Structural\Adapter>java ClassAdapterExample

The area of Demo is 12.0, and it's RED


C:\Design Patterns\Structural\Adapter>java ObjectAdapterExample

The area of Demo is 12.0, and it's RED
```

6.5. Osservazioni sull'implementazione in Java

La strategia di costruire un **Class Adapter** è possibile soltanto se l'**Adaptee** non è stato dichiarato come `final class`.