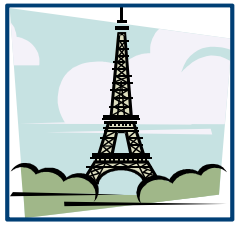
CORSO DESIGN PATTERN

I pattern Structural

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

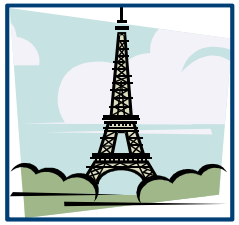
Pattern Structural



- **Scopo dei pattern di struttura**
  - Affrontare problemi che riguardano la **composizione** di classi e oggetti
  - Consentire il riutilizzo di oggetti esistenti fornendo agli utilizzatori un'interfaccia più adatta alle loro esigenze
    - integrazioni fra librerie/componenti diversi

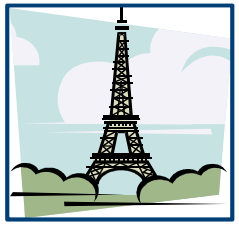
I pattern di tipo Class, sfruttano l'ereditarietà, gli altri prevalentemente l'aggregazione

Elenco Pattern

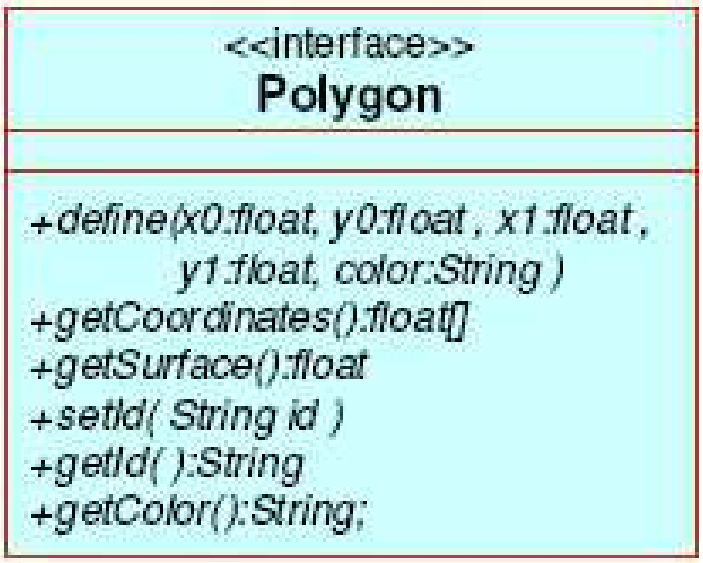


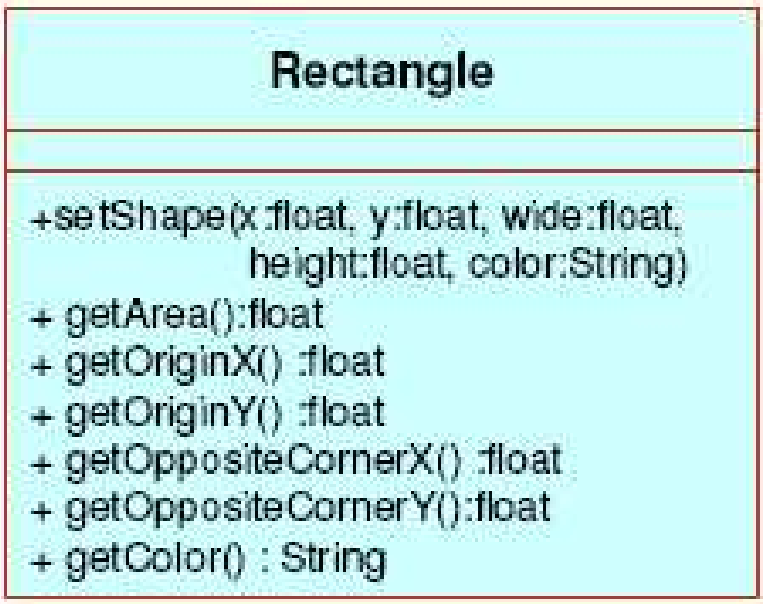
1. Adapter (Object & Class)
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Proxy
7. Flyweigth

1) Adapter

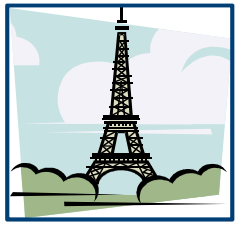


- **Esempio:** Voglio convertire (adattare) una vecchia classe Rectangle alla nuova interfaccia Polygon





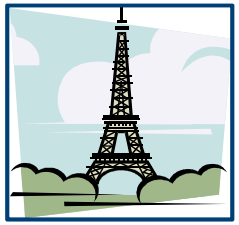
1) Adapter



- **Problema**
  - Convertire l'interfaccia di una data classe in una nuova interfaccia nota al Client
  - Rendere collaborative delle classi che altrimenti non saprebbero "parlarsi"

- **Soluzione**
  - Realizzare una classe che fa da ponte per tra la nuova interfaccia nota al client e la classe concreta da "adattare"

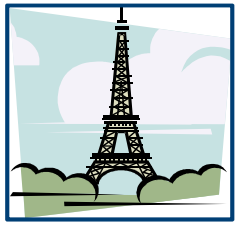
Due soluzioni: Class & Object

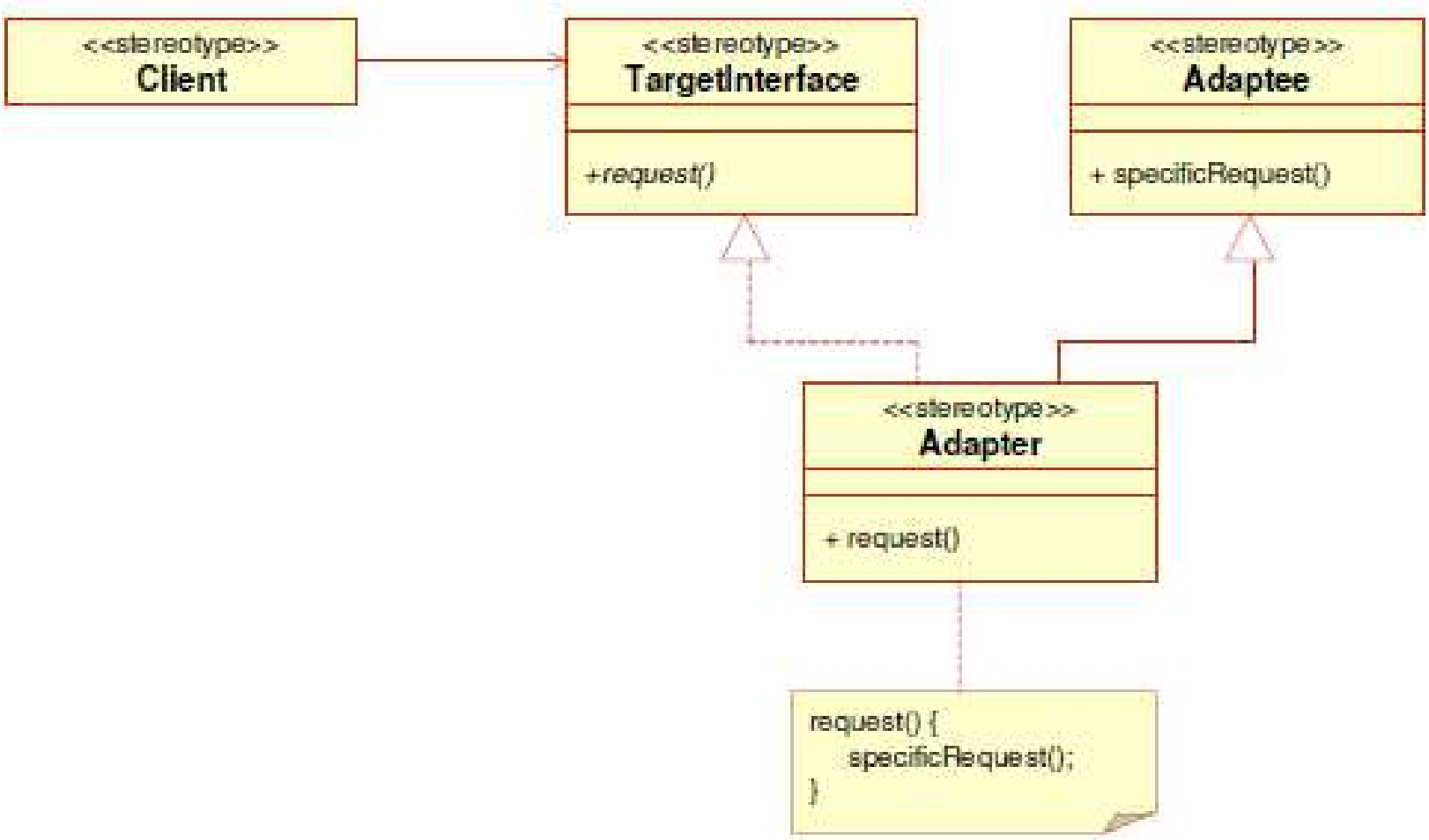


- **Class Adapter**
- La **classe esistente si estende** in una sottoclasse RectangleClassAdapter che implementa l'interfaccia desiderata.
- I metodi della sottoclasse mappano le loro operazioni in richieste ai metodi e attributi della classe di base.
- L'adattatore possiede metodi vecchi e nuovi

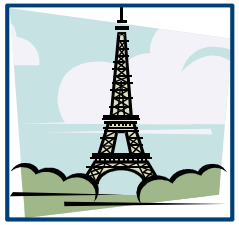
- **Object Adapter**
- Si crea una nuova classe RectangleObjectAdapter che implementa l'interfaccia richiesta, e che **possiede al suo interno un'istanza** della classe da riutilizzare.
- Le operazioni della nuova classe fanno invocazioni ai metodi dell'oggetto interno.
- L'adattatore possiede solo i metodi "nuovi"

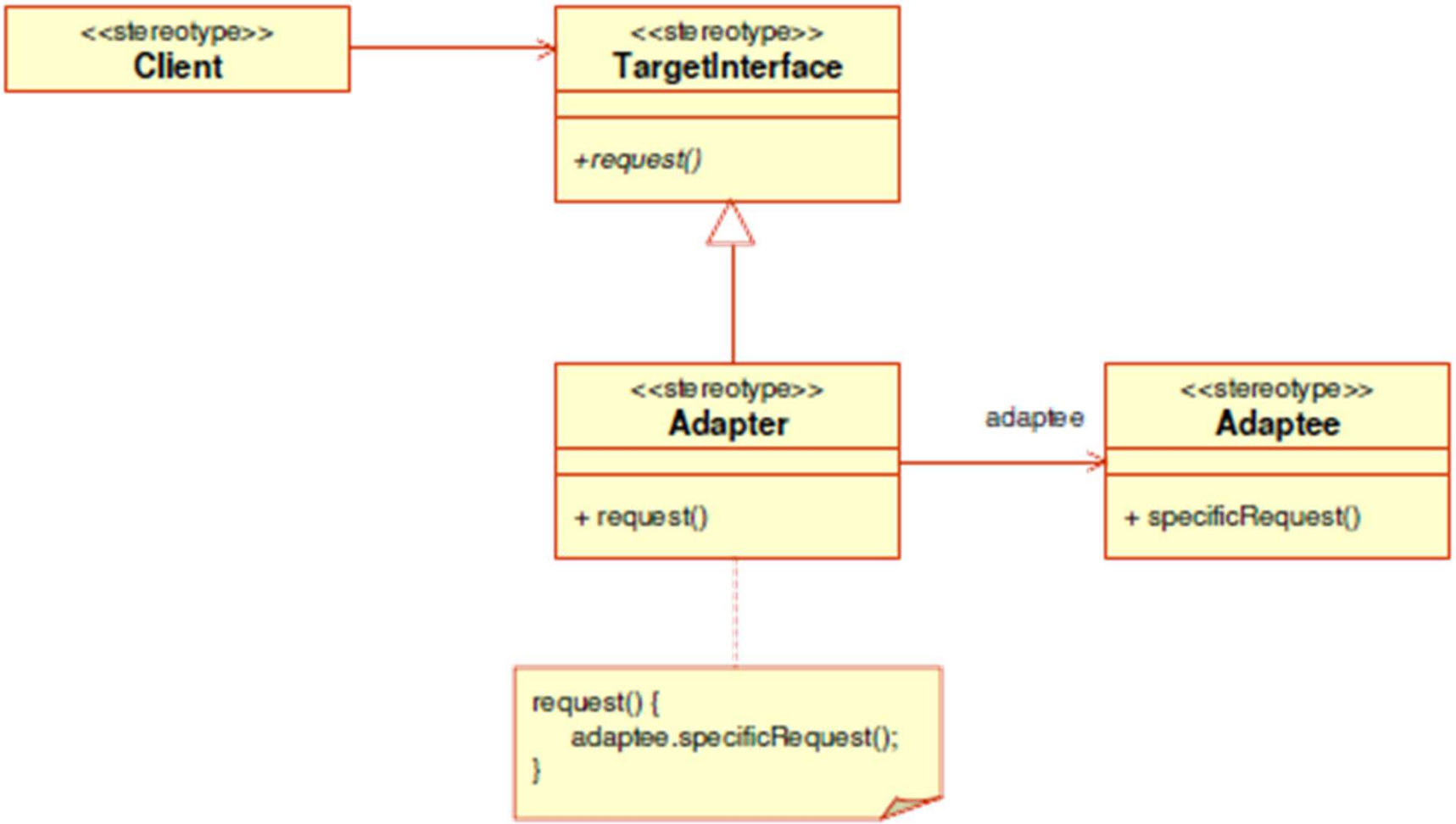
Class Adapter



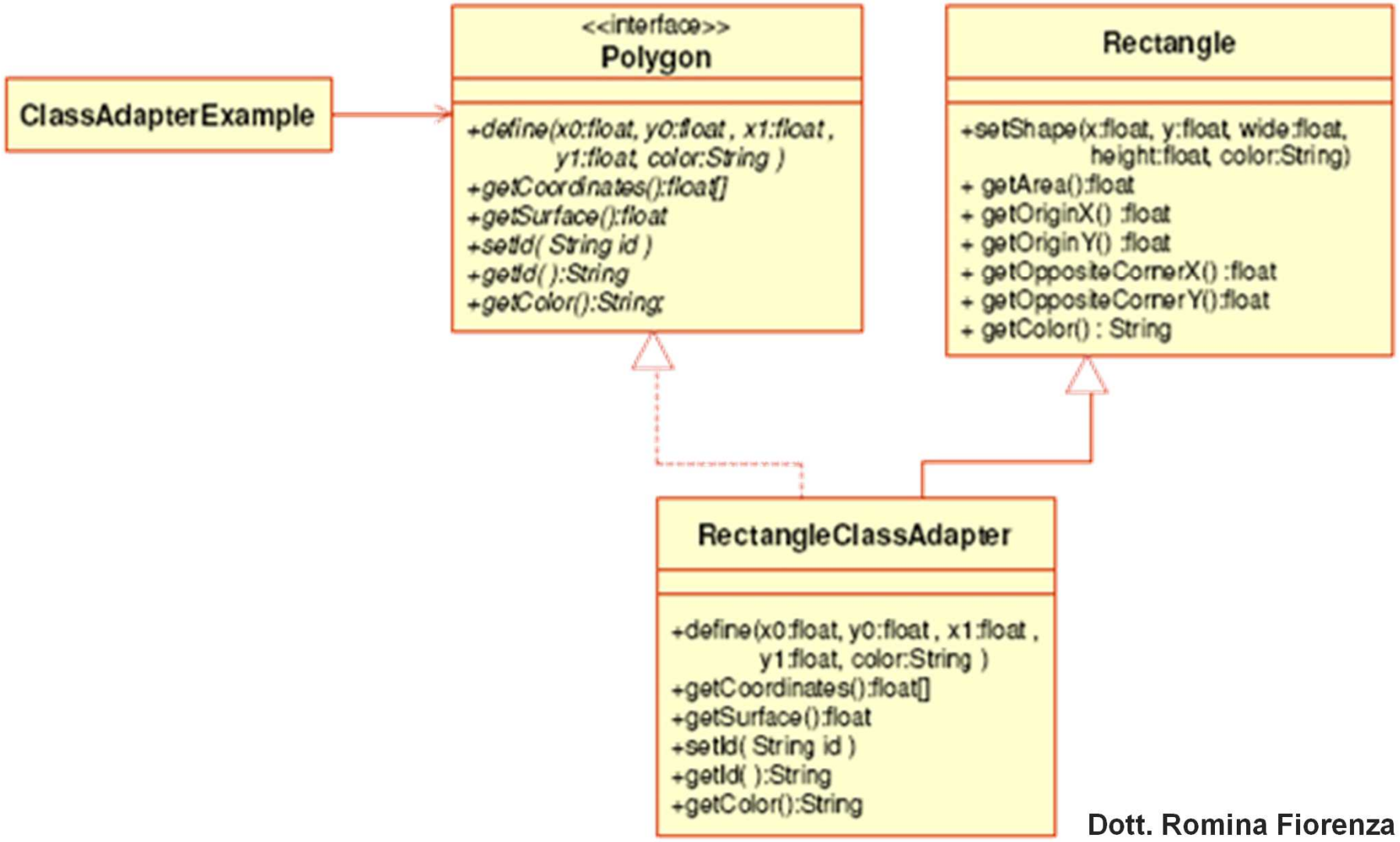


Object Adapter

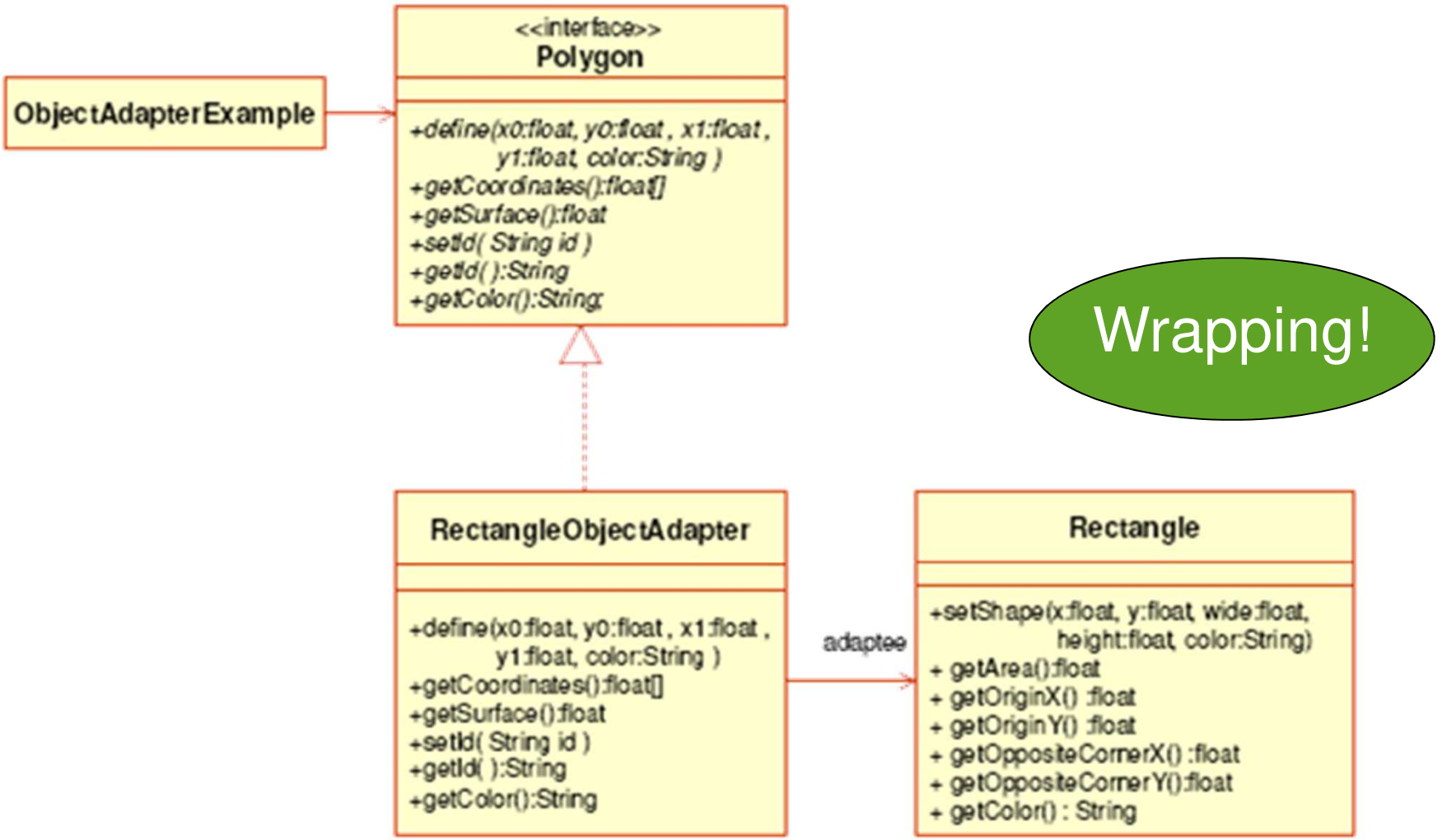




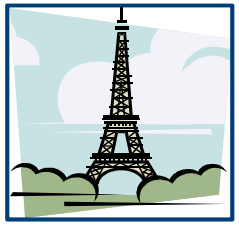
1) Class Diagram: Adapter Class



1) Class Diagram: Adapter Object



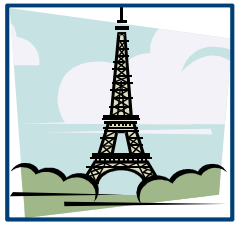
1) Adapter



Conseguenze:

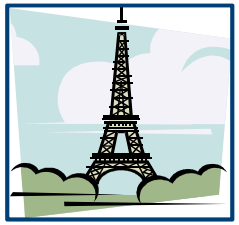
- Riutilizzo di classi esistenti
- Mascheramento di dettagli implementativi al client
- Alta responsabilità dell'adattatore concernente creazione e gestione

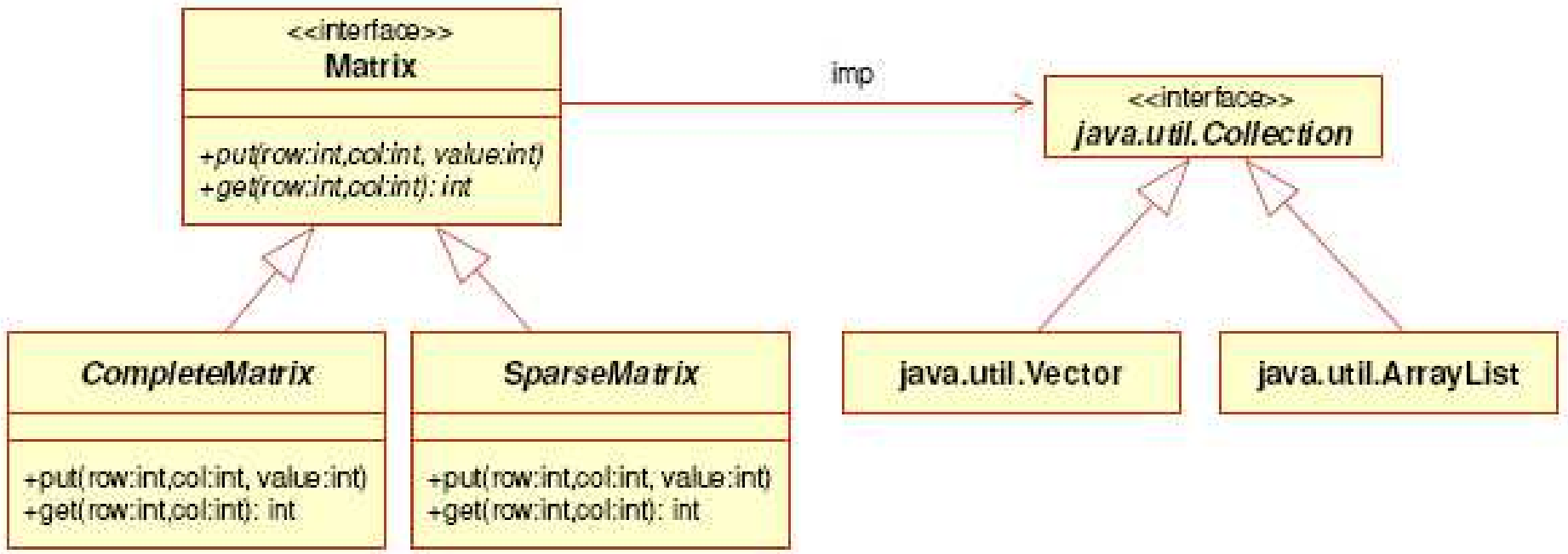
2) Bridge



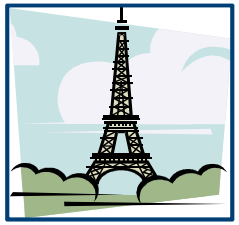
- **Esempio:**
  - Si vuole realizzare una libreria di classi che consenta di creare e usare matrici.
  - Le implementazioni previste si poggeranno sulle Collection della Sun
  - Si vuole rendere indipendente l'implementazione scelta dal particolare tipo di matrice (es. matrice completa, matrice sparsa, ecc.)

2) Bridge Esempio





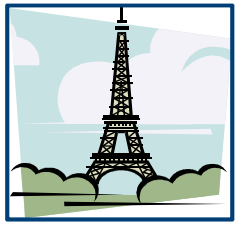
2) Bridge



- **Problema**
  - Si desidera scegliere tra implementazioni diverse da abbinare ad una stessa gerarchia di classi
  - Si vuole rendere indipendenti modalità e scopi
  - Si vuole evitare il proliferare di classi dovute alla modellizzazione di tutte le possibili combinazioni

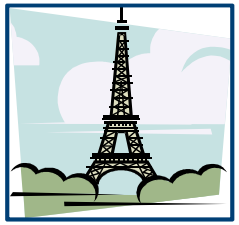
    modalità - scopo
- **Soluzione**
  - Disaccoppiare astrazione e implementazione in modo che le due possono variare indipendentemente

Individuare il Bridge (1)



- Verificare se nel dominio esistono 2 dimensioni ortogonali tra loro (“pilastri del ponte”), potrebbero essere:
  - Interfaccia e implementazione
  - Dominio e infrastruttura
  - Astrazione e piattaforma (basso livello)
  - Front-end/Back-end
  - Alto livello / basso livello

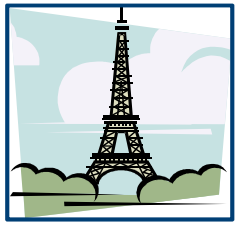
Realizzare il Bridge (2)

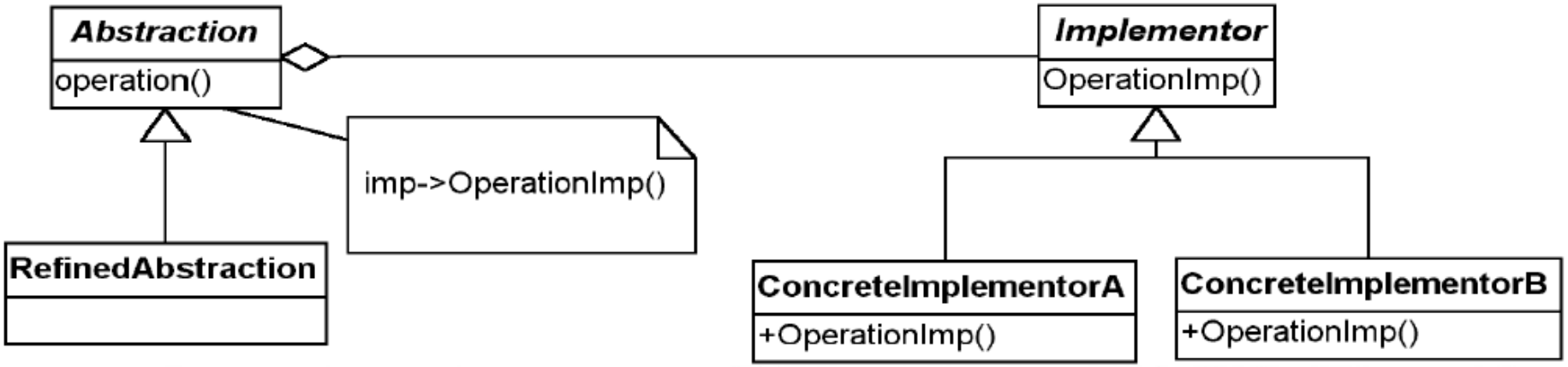


- BASSO LIVELLO
  - Definire un interfaccia platform-oriented relativa ai servizi da fornire
  - Realizzare le classi concrete platform-dependent

- ALTO LIVELLO
  - Creare la classe base di alto livello (l'abstraction vista dal Client) in modo che **disponga** di un oggetto platform-oriented (basso livello)
  - Raffinare con le classi concrete di alto livello.

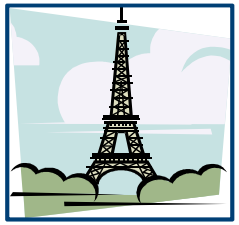
2) Class Diagram Bridge





- **`Abstraction`** possiede un **`Implementor`**.
- Entrambe sono astrazioni che vanno raffinate
- Astrazioni e implementazioni variano indipendentemente

2) Bridge Esempio

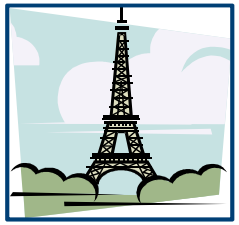


- L' interfaccia o classe astratta Matrix (**l'Abstraction**)
  - specifica l'interfaccia dell'astrazione
  - gestisce un riferimento ad un oggetto **Implementor.**
- Le classi CompleteMatrix e SparseMatrix rappresentano particolari tipi di Matrix (**Abstraction**)

- L'**Implementor,** interfaccia Collection che specifica l'interfaccia definita per le classi di implementazione.
- Le classi Vector e ArrayList che implementano l'interfaccia **Implementor,** sono i **ConcreteImplementor**

- Il client usa oggetti concreti di tipo Abstraction , ma sempre invocando i metodi dell'interfaccia

2) Bridge

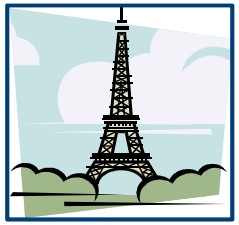


Conseguenze

- Disaccoppia interfaccia ed implementazione
- Buon livello di l'estensibilità (per le 2 gerarchie)
- I dettagli implementativi sono nascosti al Client
- No Class proliferation
  - Perché le strutture complesse sono scomposte in strutture ortogonali di classi "semplici e maneggevoli"

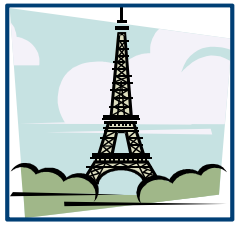
- Mentre l'Adapter si utilizza per adattare cose "vecchie", il Bridge consente di progettare cose "nuove" che devono avere configurazioni o implementazioni diverse

3) Composite

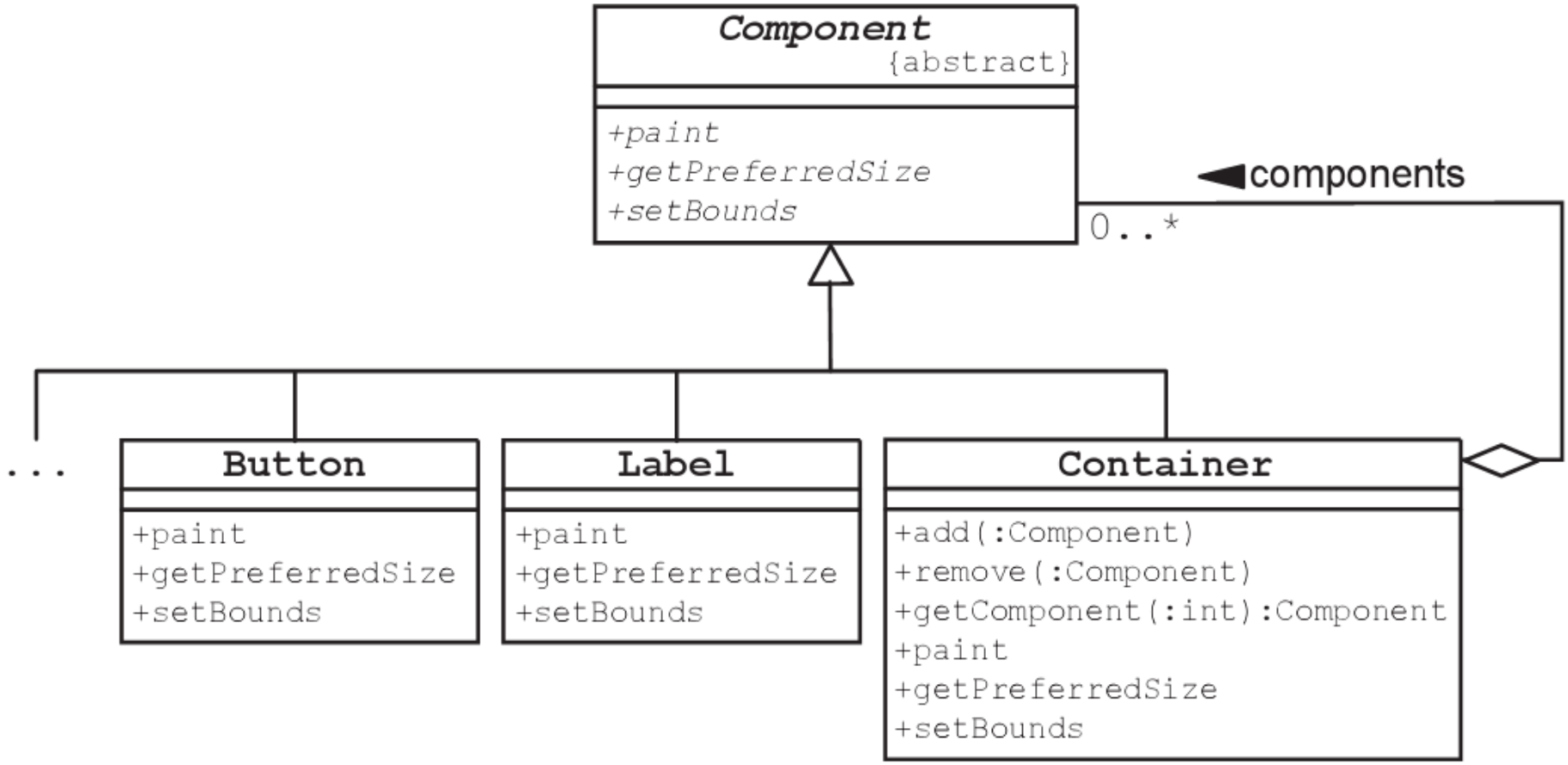


- **Problema**
  - vogliamo rappresentare una gerarchia di oggetti tutto-parte (che contengono se stessi: struttura ricorsiva)
  - il client deve poter accedere tramite una unica interfaccia che può essere tutto o parte
- **Soluzione**
  - Realizzare un'interfaccia generica dalla quale deriva il tutto ed il parte
  - Il tutto contiene metodi per trattare la sua struttura composta da parti

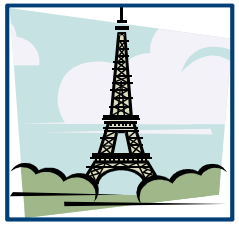
3) Composite

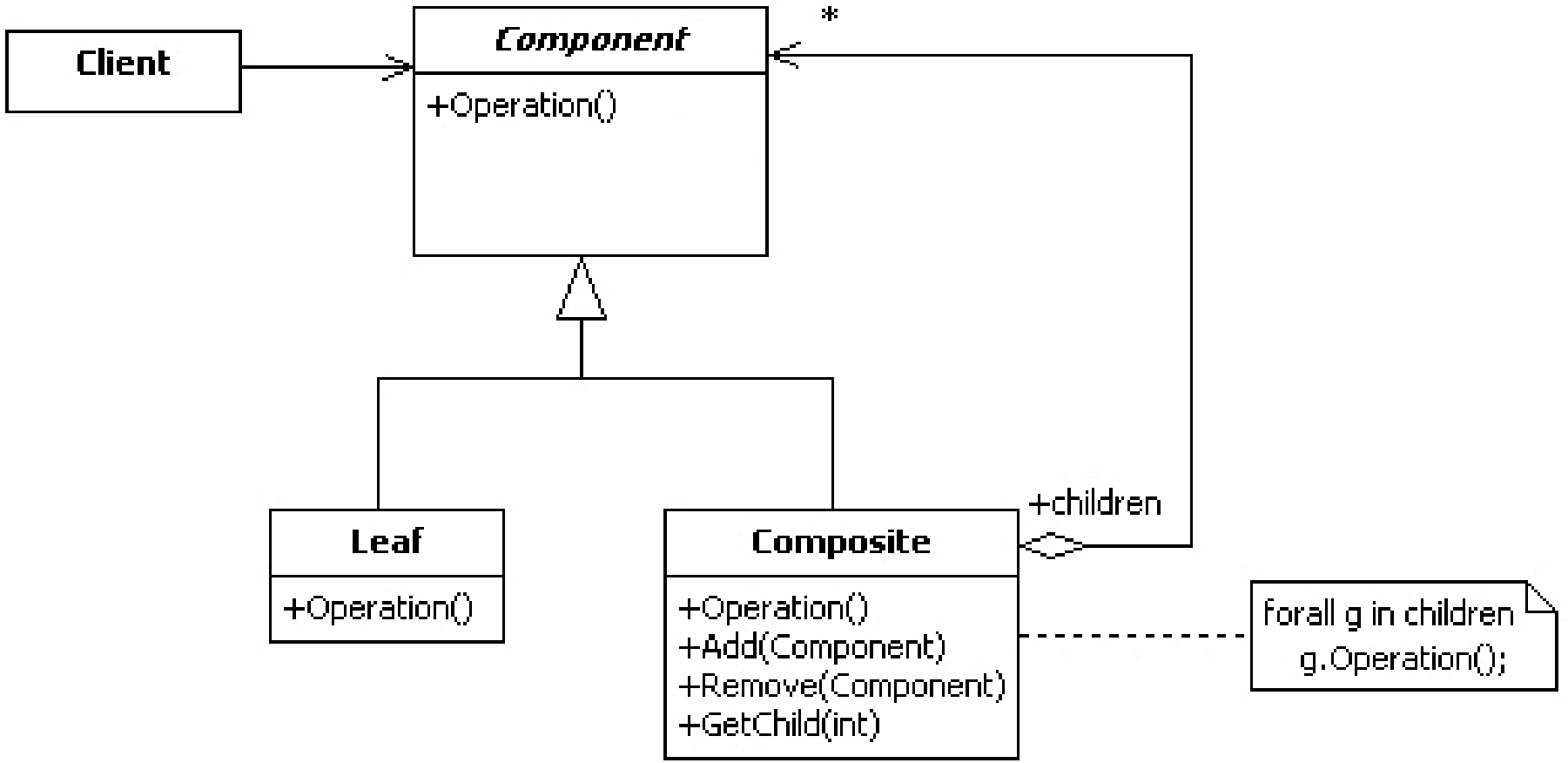


- **Esempio:** una finestra grafica

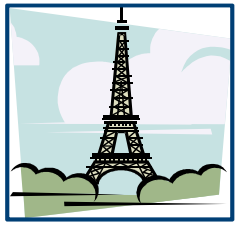


3) Class Diagram Composite





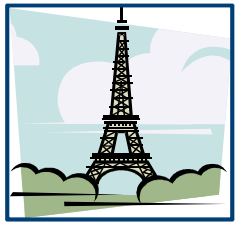
3) Composite



Conseguenze

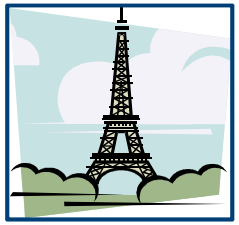
- E' il modo migliore per rappresentare strutture ricorsive
- Rende il modello della struttura più generico → perchè evita ripetizioni nel diagramma

4) Decorator



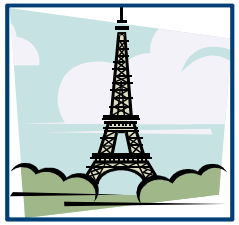
- **Problema**
  - Aggiungere dinamicamente responsabilità ad un oggetto (farlo “evolvere”)
  - Fornire un’alternativa al subclassing (per l’estensione delle funzionalità)
- **Soluzione:**
  - Costruire un oggetto a partire da un altro dello stesso albero di generalizzazione, MA senza ricorsione

4) Decorator

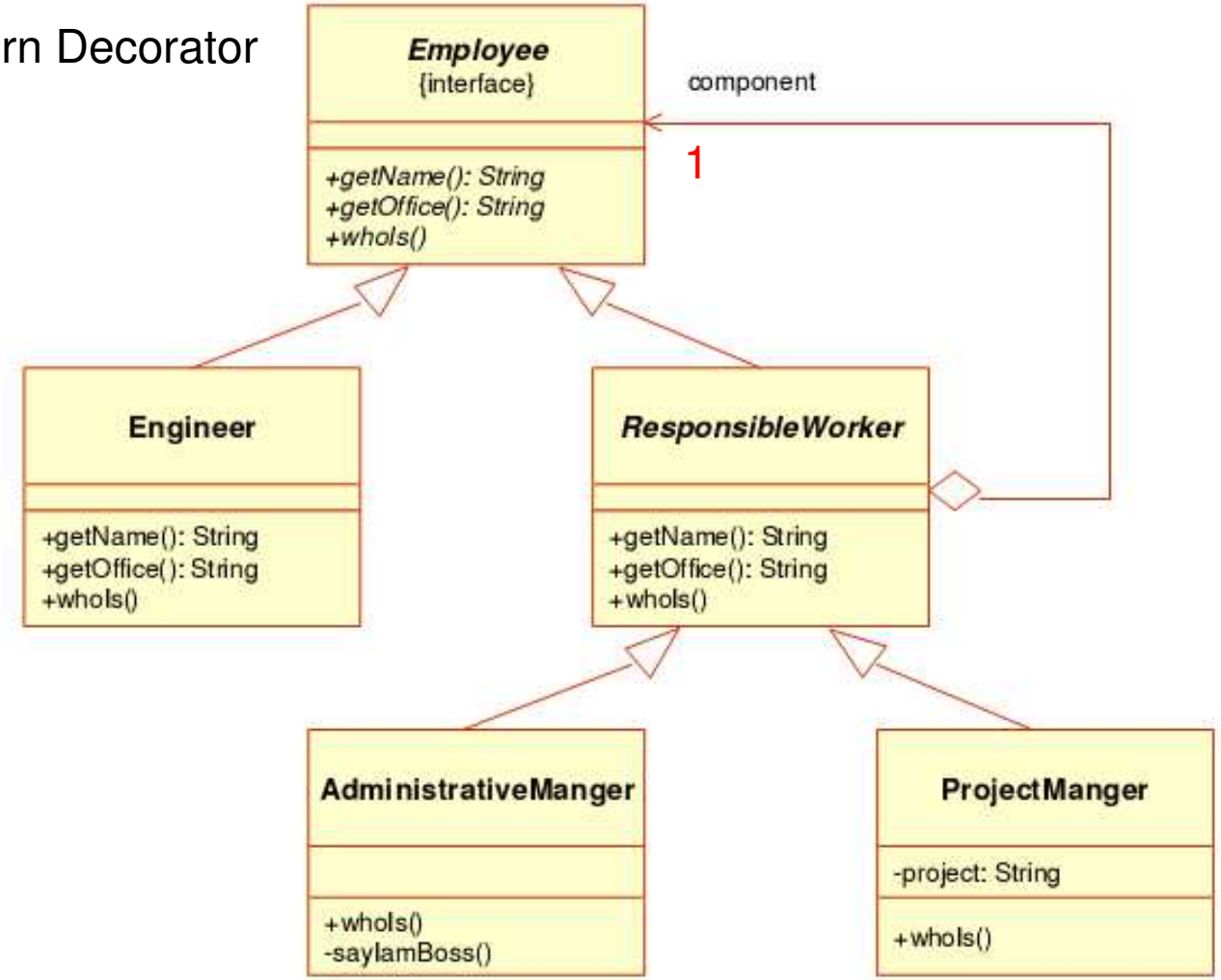


- **Esempio:** la promozione
- Si possiede un modello di gestione di oggetti che rappresentano gli impiegati (Employee) di una azienda.
- Il sistema vuole prevedere la possibilità di "promuovere" gli impiegati con delle responsabilità aggiuntive.
- Ad esempio, da impiegato a capoufficio (Administrative Manager) oppure da impiegato a capo progetto (Project Manager).
- Nota: Queste responsabilità non si escludono tra di loro.

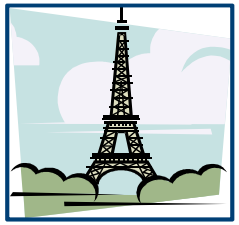
4) Decorator

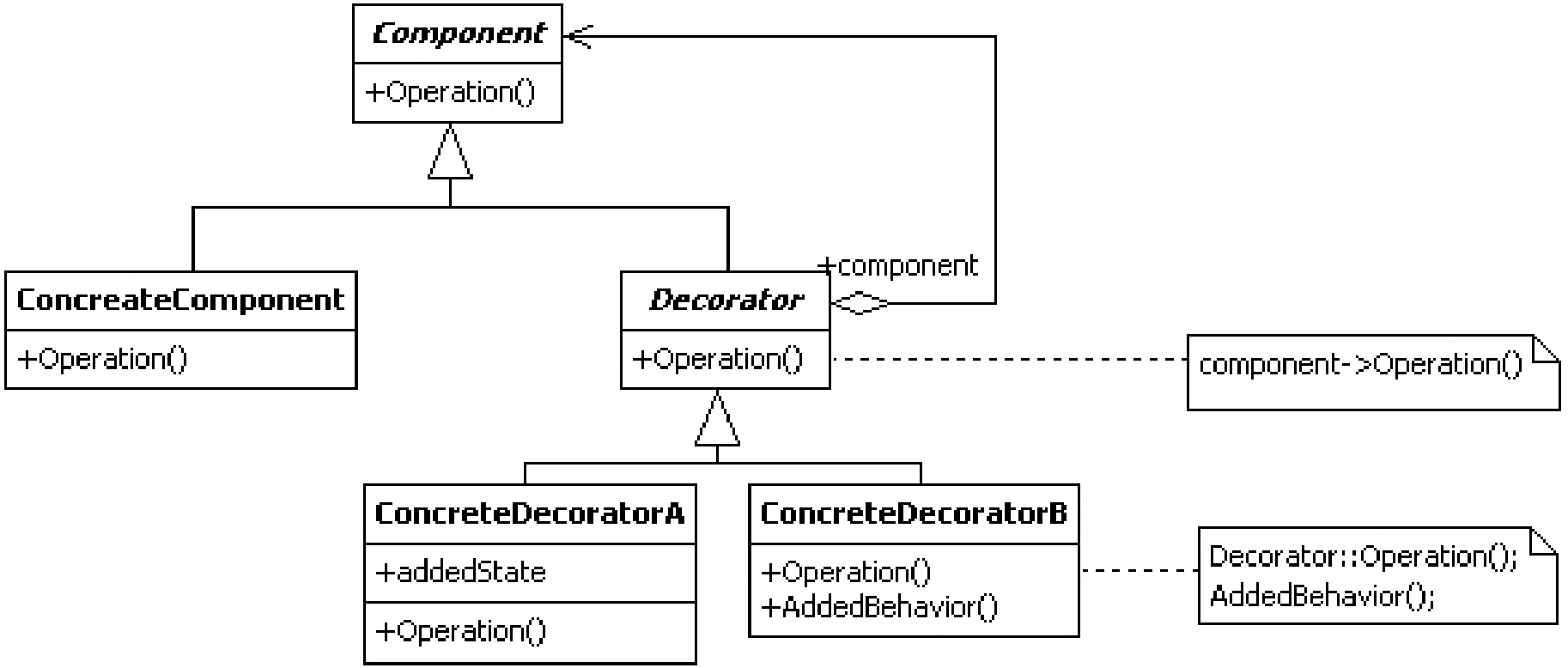


Uso del pattern Decorator



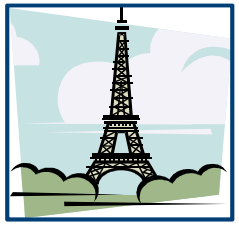
4) Class Diagram Decorator





- **Decorator** possiede un riferimento all'oggetto Component e specifica un'interfaccia concordante con l'interfaccia Component.
- **Il ConcreteDecorator** eredita i comportamenti di **Component** e aggiunge nuove responsabilità.

4) Decorator

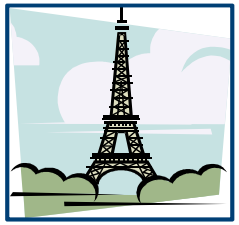


Conseguenze

- Estende le funzionalità di oggetti a RUN-TIME
- L'applicazione potrà interagire con gli oggetti decorati in modo trasparente, in quanto essi (come il Decorator) implementano un'interfaccia comune nota.

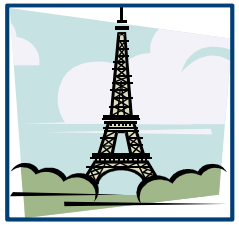
- Per una stessa interfaccia possono esserci più Decorator e dunque la possibilità di decorare oggetti già decorati.

5) Façade

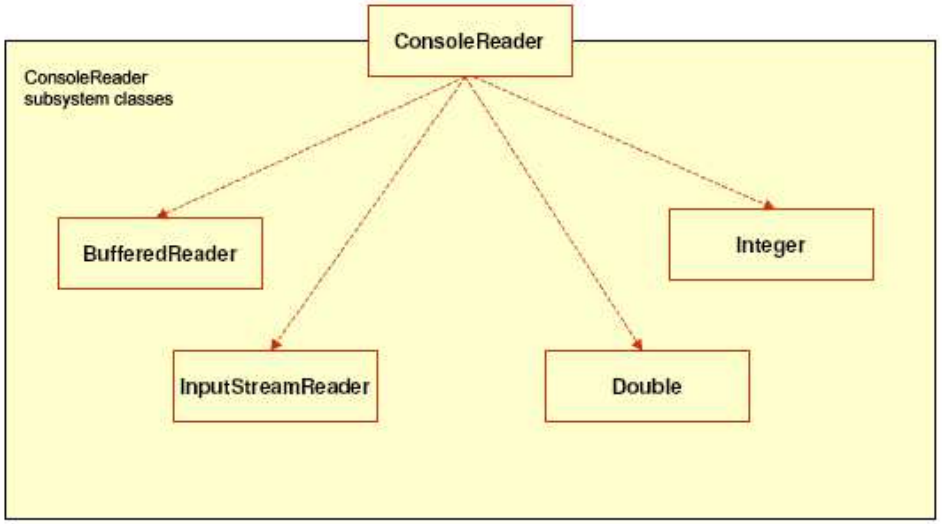


- **Problema**
  - Il sistema fornisce diverse funzionalità ma tramite diversi punti di accesso (diverse classi). Il client deve quindi conoscere il sistema "dall'interno" per utilizzarlo.
- **Soluzione**
  - Realizzare una classe che si interfaccia con le classi sottostanti ("la facciata")
    - unifica l'accesso alle funzionalità
    - opera per delegazione.
    - il client utilizza soltanto la facciata

5) Façade

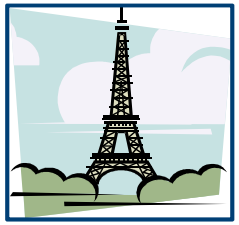


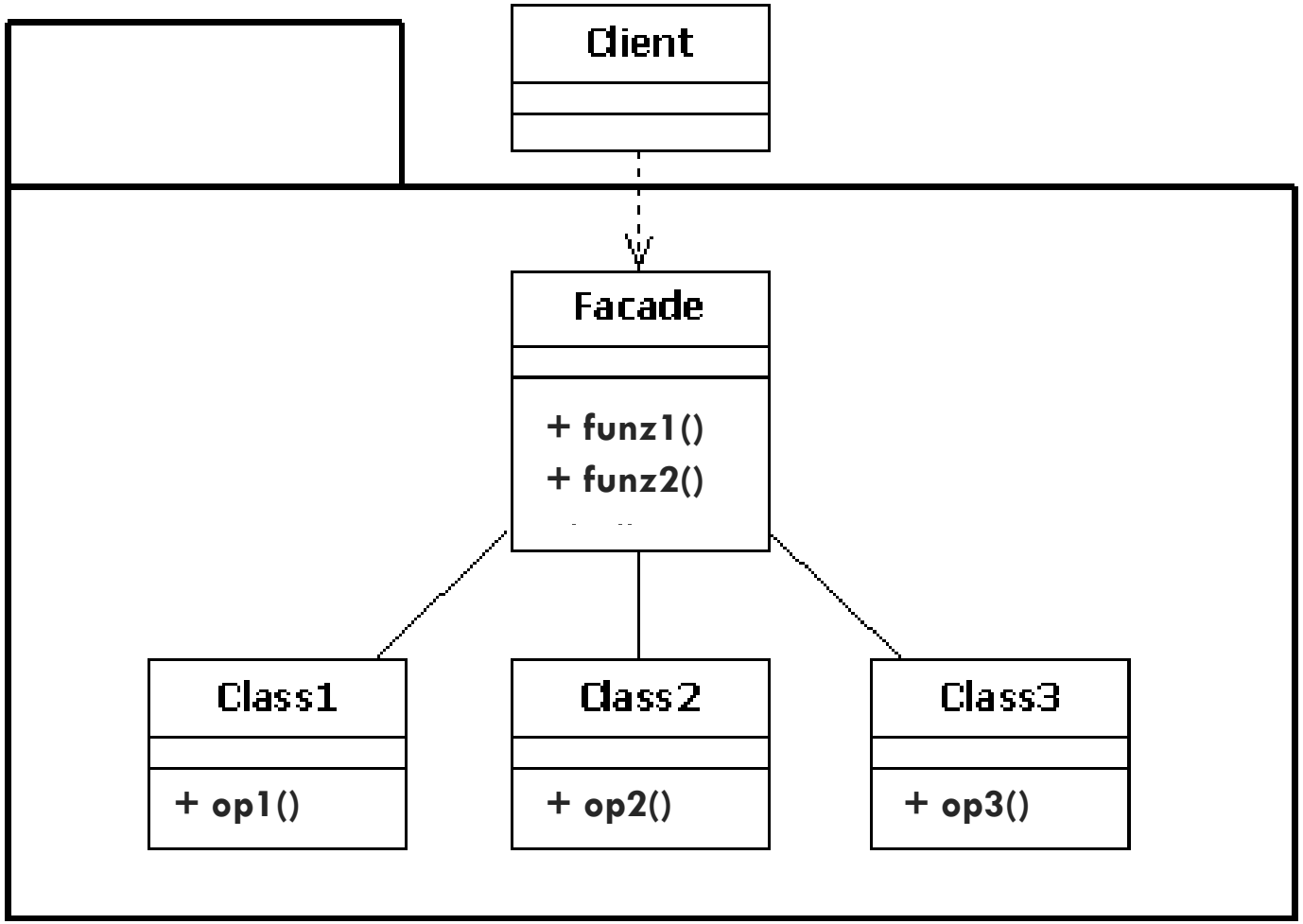
- **Esempio:** classe Console
- All'interno di un'applicazione si vuole consentire la lettura da tastiera di tipi di dati diversi (es. interi, float, stringhe, ecc).
- Questo comporta l'utilizzo delle
  - BufferedReader
  - InputStreamReader
  - System.in
  - Integer, Float, String, ecc



- Il pattern Facade sponsorizza lo sviluppo di una classe Console che espone metodi di lettura primitivi/stringhe e incapsula regole e conoscenza degli strumenti per la lettura dalla tastiera

4) Façade Class diagram



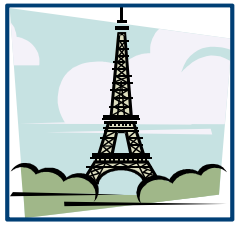


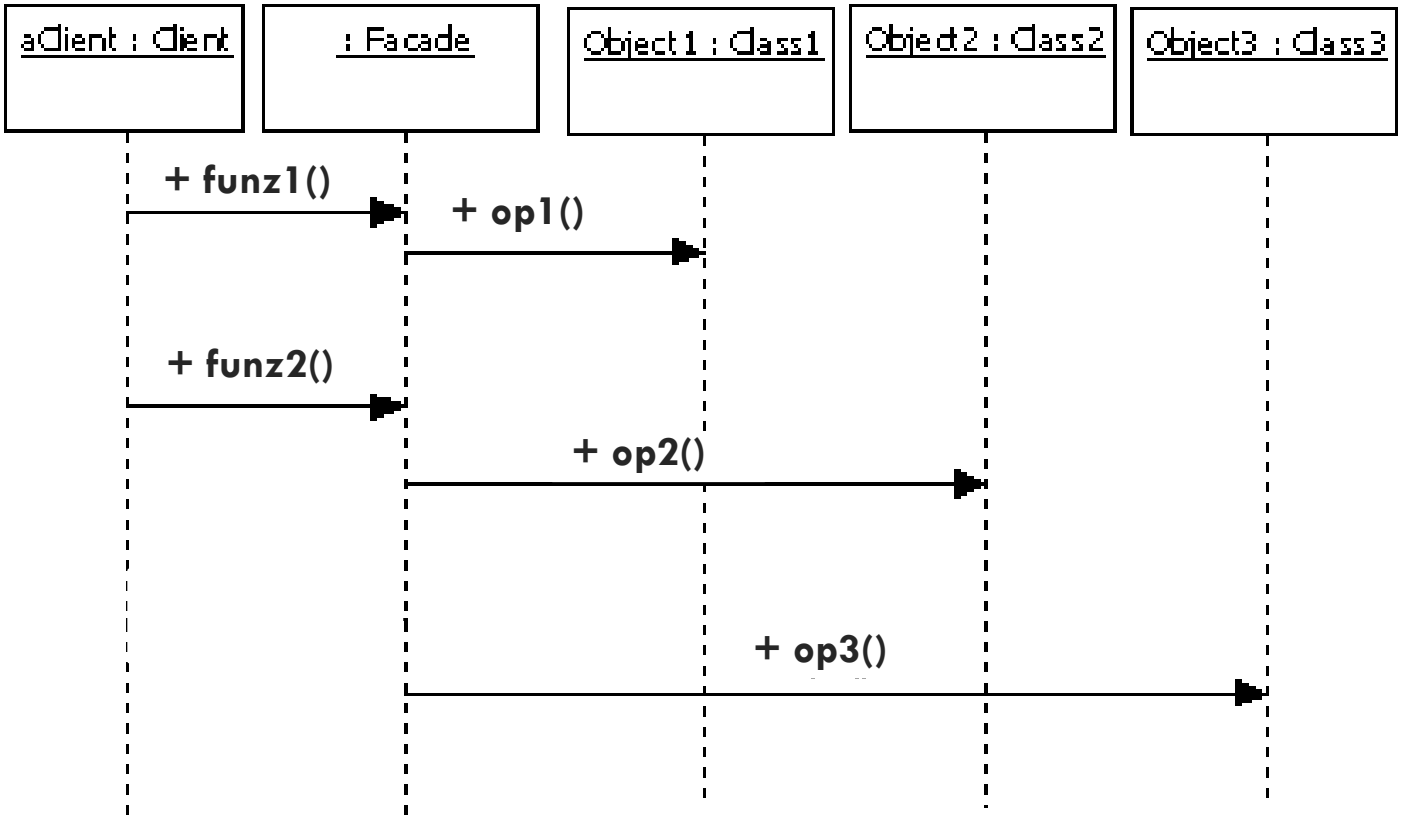
La complessità delle interazioni tra gli oggetti è gestita

della classe Facade al fine di raggiungere l'obbiettivo

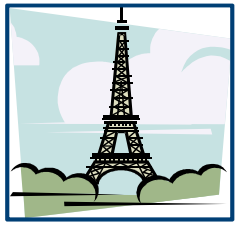
desiderato esposto attraverso i suoi metodi.

4) Façade Sequence Diagram





5) Façade

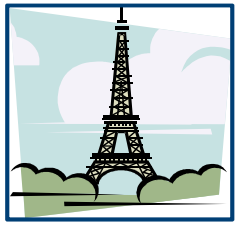


Conseguenze

- CONTRO: Una sola classe spesso troppo corposa
- Single point of failure → nella façade
- Unico punto di riferimento per il client (centralizzazione delle funzionalità)

- Utilizzato per le chiamate remote

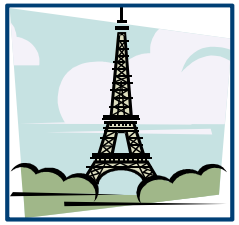
Façade *vs* Adapter



	Facade	Adapter
Are there preexisting classes?	Yes	Yes
Is there an interface we must design to?	No	Yes
Does an object need to behave polymorphically?	No	Probably
Is a simpler interface needed?	Yes	No

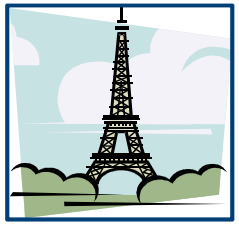
- il Facade *semplifica* un'interfaccia mentre l'Adapter la *converte* in una preesistente
- Il Facade di solito nasconde più classi, l'Adapter di solito 1

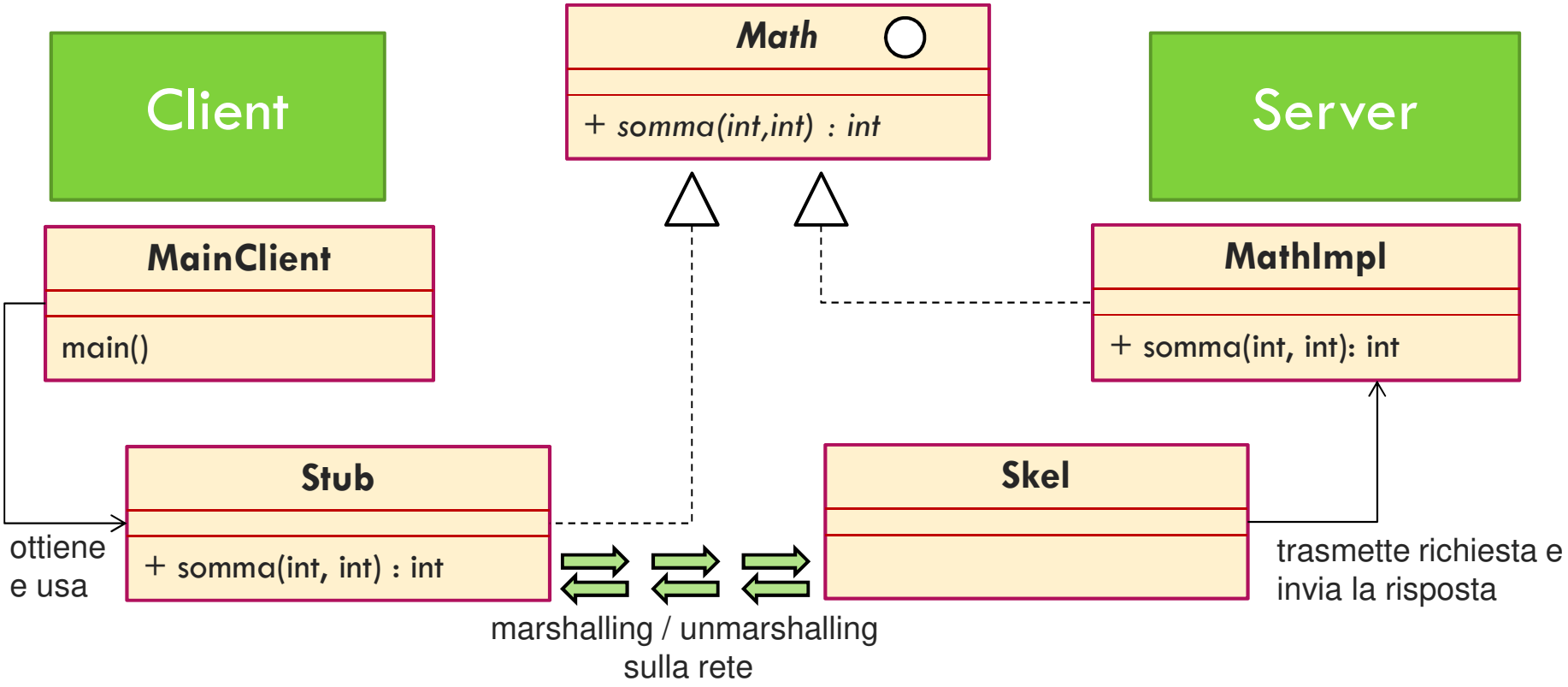
6) Proxy



- **Problema**
  - Per ragioni di sicurezza o di controllo accesso, bisogna mascherare un oggetto (e il suo comportamento) al client
- **Soluzione**
  - Una classe (Proxy) fornisce l'accesso ad un'altra classe, comportandosi come una sorta di filtro
  - Sia la classe Proxy, che quella che viene "filtrata" devono implementare la stessa interfaccia

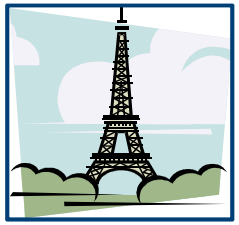
6) Proxy: RMI

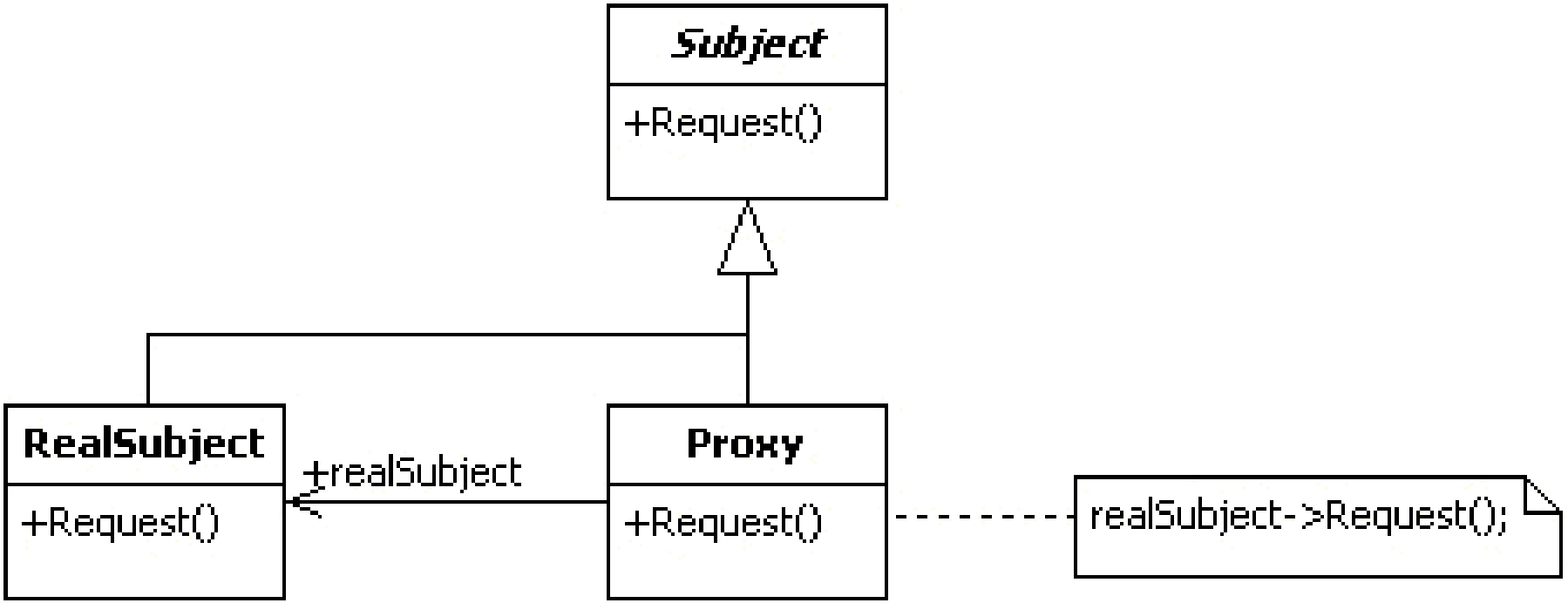




Il programma **Client** chiede e ottiene un oggetto **Math** attraverso una lookup
Il programma Client usa l'oggetto **Stub**, il filtro, come se fosse l'oggetto reale

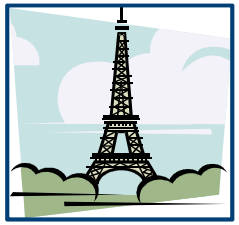
6) Class Diagram Proxy





Rappresentazione di un oggetto di accesso difficile o che richiede un tempo importante per l'accesso o creazione.

6) Proxy



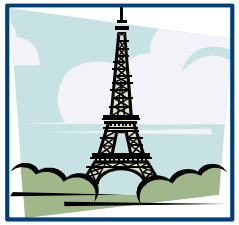
Conseguenze

- Il proxy può essere responsabile
  - della creazione dell'oggetto reale
  - della politica di accesso all'oggetto reale

- Potrebbe filtrare le richieste più semplici ed evaderle autonomamente e cedere il controllo alla classe filtrata solo per le richieste più complesse. (è il *virtual proxy*)

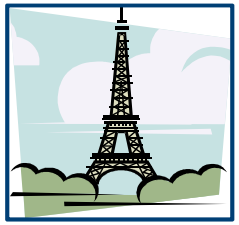
- Potrebbe immagazzinare dati relativi all'ultima/ultime operazioni effettuata, riducendo tempi di elaborazione per operazioni ripetute (è il *cache proxy*)

7) Flyweight



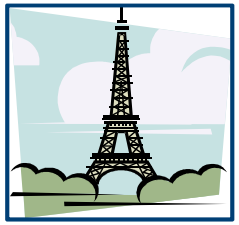
- **Problema:**
  - Un'applicazione deve gestire un gran numero di oggetti "piccoli" di una stessa classe
  - La classe prevede proprietà con valori condivisibili e proprietà con valori non condivisibili
- **Soluzione:**
  - Separare la creazione delle 2 parti, facendo gestire la parte condivisibile da una factory

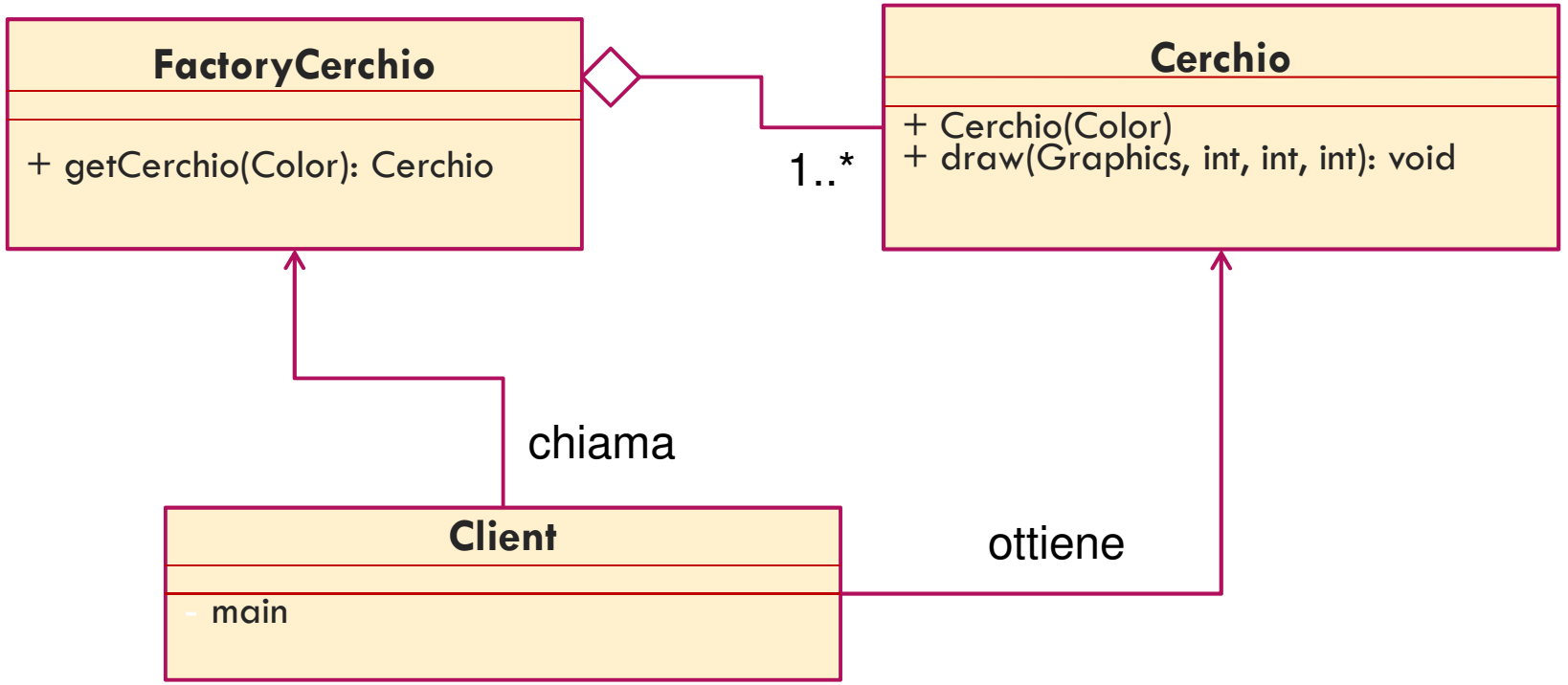
7) Flyweight



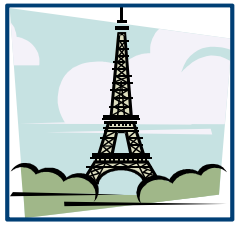
- **Esempio:**
  - Si vuole creare un programma che mostri una finestra grafica con 1000 cerchi che abbiamo:
    - **Colore** (che può variare tra 6 possibili colori)
    - **Posizione** (entro la finestra)
    - **Raggio** (massimo un decimo dell'altezza della finestra)
  - Le parti altamente variabili → range valori troppo vasto sono **Posizione** e **Raggio**
  - La parte condivisibile → legata un piccolo numero di valori (spesso immutabili) è **Colore**

7) Flyweight



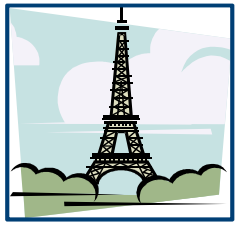

FactoryCerchio
+ getCerchio(Color): Cerchio
1..*
Cerchio
+ Cerchio(Color)
+ draw(Graphics, int, int, int): void
chiama
Client
- main
ottiene

7) Flyweight: Esempio



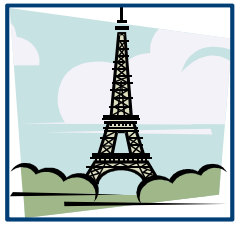
```
import java.awt.Color;
import java.awt.Graphics;

public class Circle {
      private Color color;

      public Circle(Color color) {
         this.color = color;
      }
      public void draw(Graphics g, int x, int y, int r) {
         g.setColor(color);
         g.drawOval(x, y, r, r);
      }
}
```

7) Flyweight: Esempio



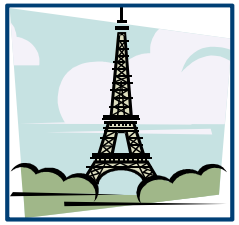
```
import java.awt.Color;
import java.util.HashMap;

public class CircleFactory {
      private static final HashMap circleByColor = new HashMap();

      public static Circle getCircle(Color color) {
         Circle circle = (Circle)circleByColor.get(color);
         if(circle == null) {
            circle = new Circle(color);
            circleByColor.put(color, circle);
            System.out.println("Creating " + color + " circle");
         } return circle;
      }
}
```

7) Flyweight



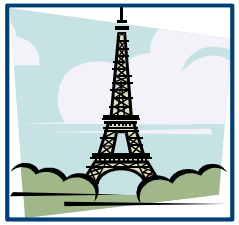
```
public class FinestraCerchietti extends JFrame{
   private static final Color colors[] = { Color.red, Color.blue,
   Color.yellow, Color.orange, Color.black, Color.white};
   private static final int WIDTH = 400, HEIGHT = 400,
                            NUMBER_OF_CIRCLES = 1000;
   public FinestraCerchietti() {
      Container contentPane = getContentPane();
      JButton button = new JButton("Draw Circle");
      final JPanel  panel  = new JPanel();
      contentPane.add(panel,  BorderLayout.CENTER);
      contentPane.add(button, BorderLayout.SOUTH);
      setSize(WIDTH,HEIGHT);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setVisible(true);
```

7) Flyweight

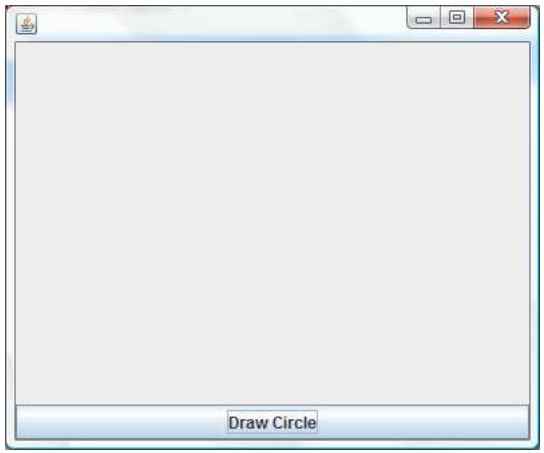


```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
           Graphics g = panel.getGraphics();
           for(int i=0; i < NUMBER_OF_CIRCLES; ++i) {
              Circle circle = CircleFactory.getCircle(getRandomColor());
              circle.draw(g, getRandomX(), getRandomY(),getRandomR());
              //si prevedono solo 6 differenti colori, si hanno solo 6 oggetti cerchio
           }}});
   }
   private int getRandomX() {  return (int)(Math.random()*WIDTH ); }
   private int getRandomY() {  return (int)(Math.random()*HEIGHT); }
   private int getRandomR() {  return (int)(Math.random()*(HEIGHT/10)); }
   private Color getRandomColor() {
           return colors[(int)(Math.random()*colors.length)];
    }
}
```

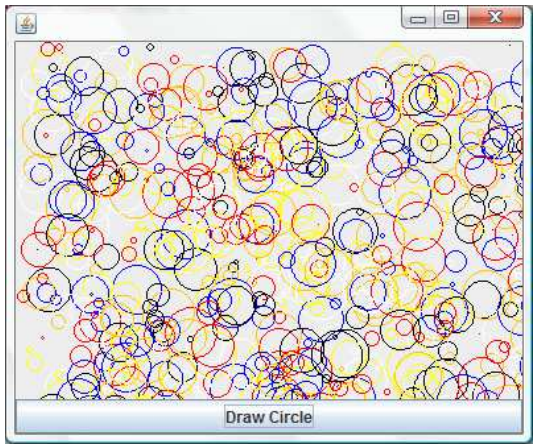
7) Flyweight: Esempio



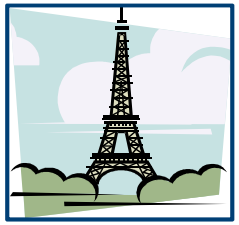
```
import java.awt.Color;
import java.util.HashMap;

public class Main{
    public static void main(String[] args) {
    FinestraCerchietti test = new FinestraCerchietti();
    }
}
```

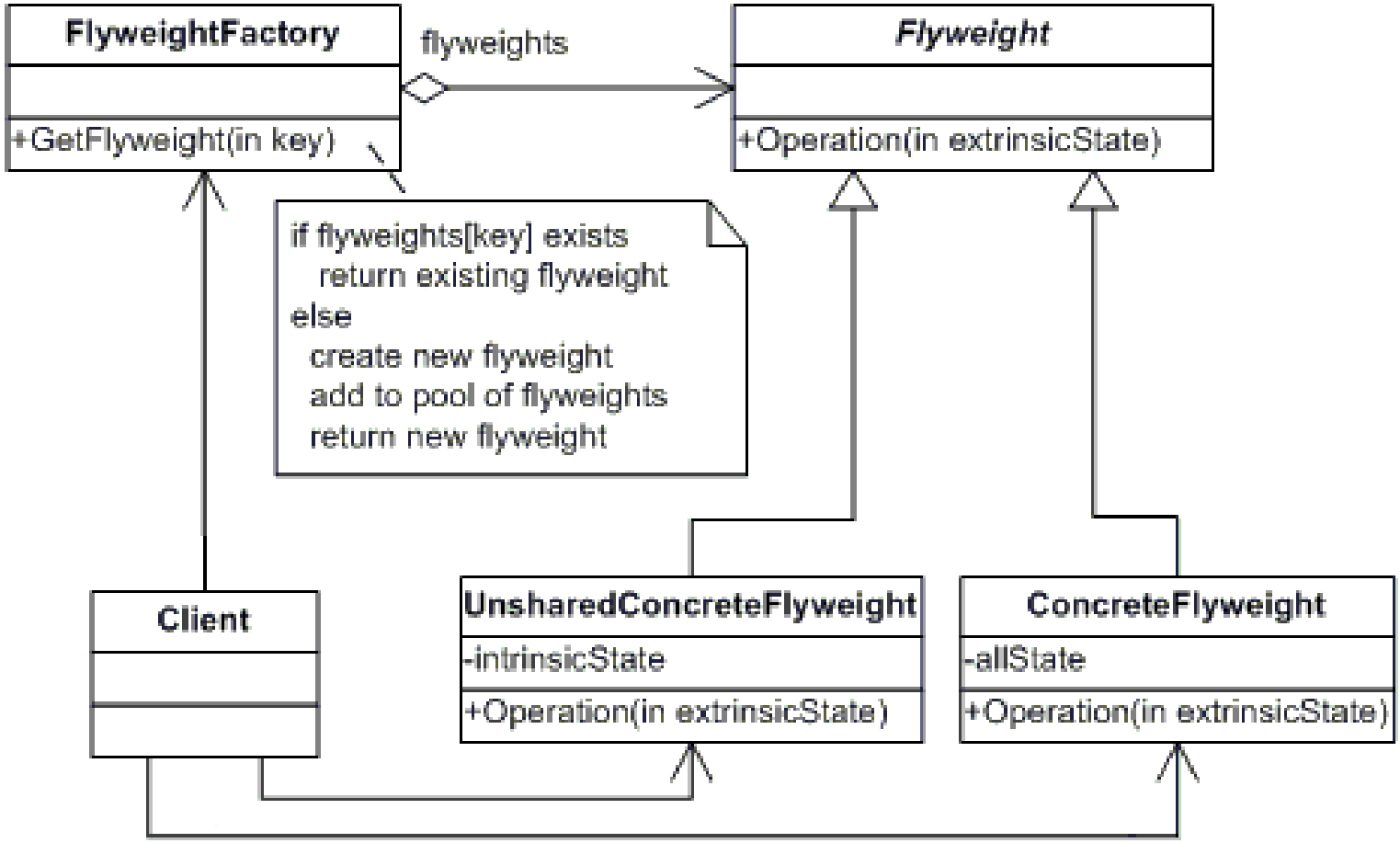


Clikkando sul pulsante si ottiene

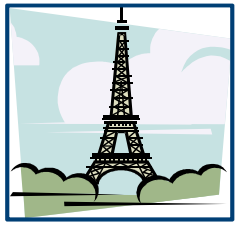


7) Flyweight




FlyweightFactory
flyweights
Flyweight
+GetFlyweight(in key)
+Operation(in extrinsicState)
if flyweights[key] exists
return existing flyweight
else
create new flyweight
add to pool of flyweights
return new flyweight
Client
UnsharedConcreteFlyweight
-intrinsicState
+Operation(in extrinsicState)
ConcreteFlyweight
-allState
+Operation(in extrinsicState)

7) Flyweight



Se invece volessimo modellare la classe Cerchio anche con attributi coordinate e raggio, questa è la scomposizione con la parte flyweight data dalla classe Color