UNIVERSITÀ PARTHENOPE
DiST
Dipartimento di Scienze e Tecnologie

Programmazione 3
e
Laboratorio di Programmazione 3

Structural Patterns

Angelo Ciaramella

Structural Patterns

- Pattern strutturali
  - consentono di riutilizzare degli oggetti esistenti fornendo agli utilizzatori un'interfaccia più adatta alle loro esigenze

- Design Pattern
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy

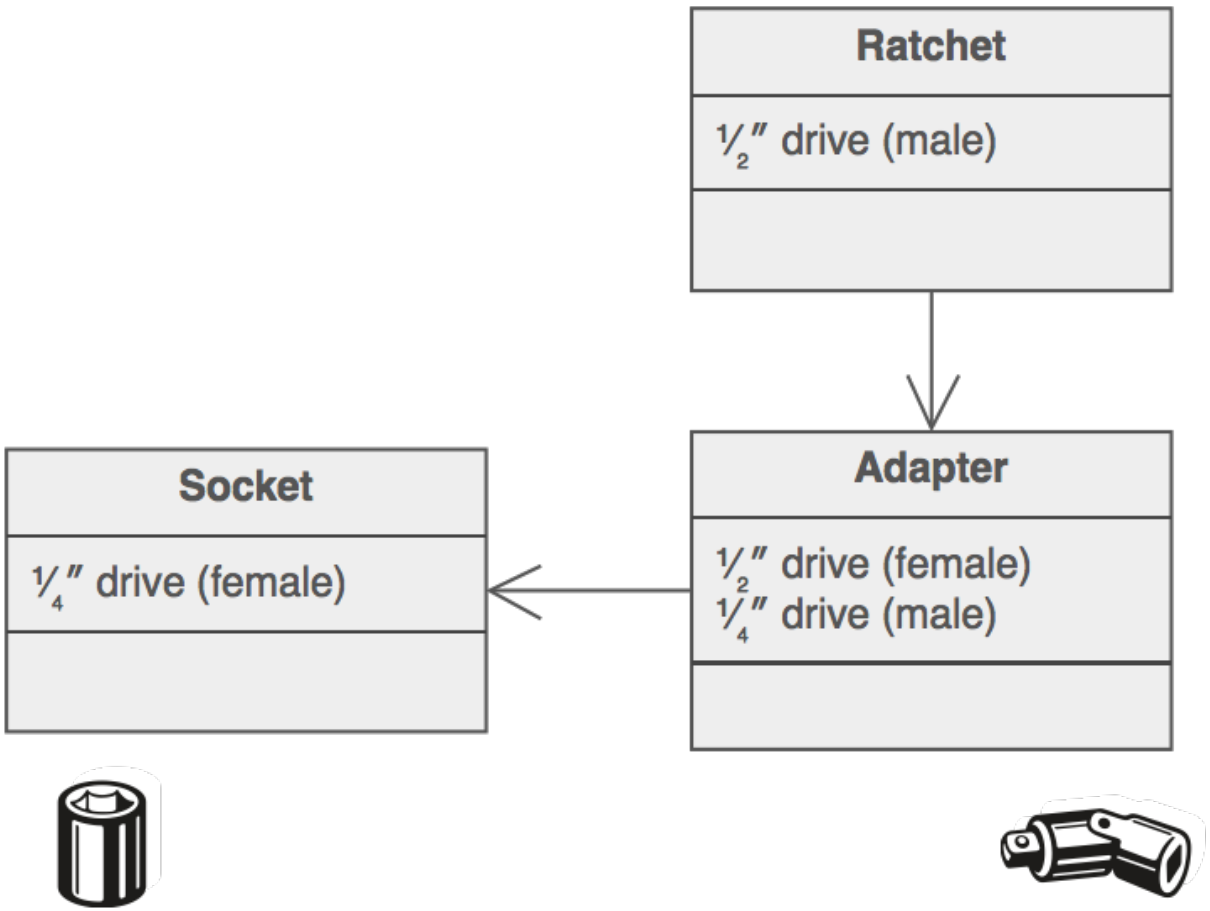
Adapter

- Scopo
  - *Convertire l'interfaccia di una classe in un'altra richiesta dai client. Permette alle classi di collaborare sebbene esista un'incompatibilitò di interfacce.*
- Anche conosciuto come
  - Wrapper (involucro)
- Motivazione
  - fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti
    - e.g., in un software si devono utilizzare sistemi di supporto (come per esempio librerie) la cui interfaccia non è perfettamente compatibile con quanto richiesto da applicazioni già esistenti

Adapter

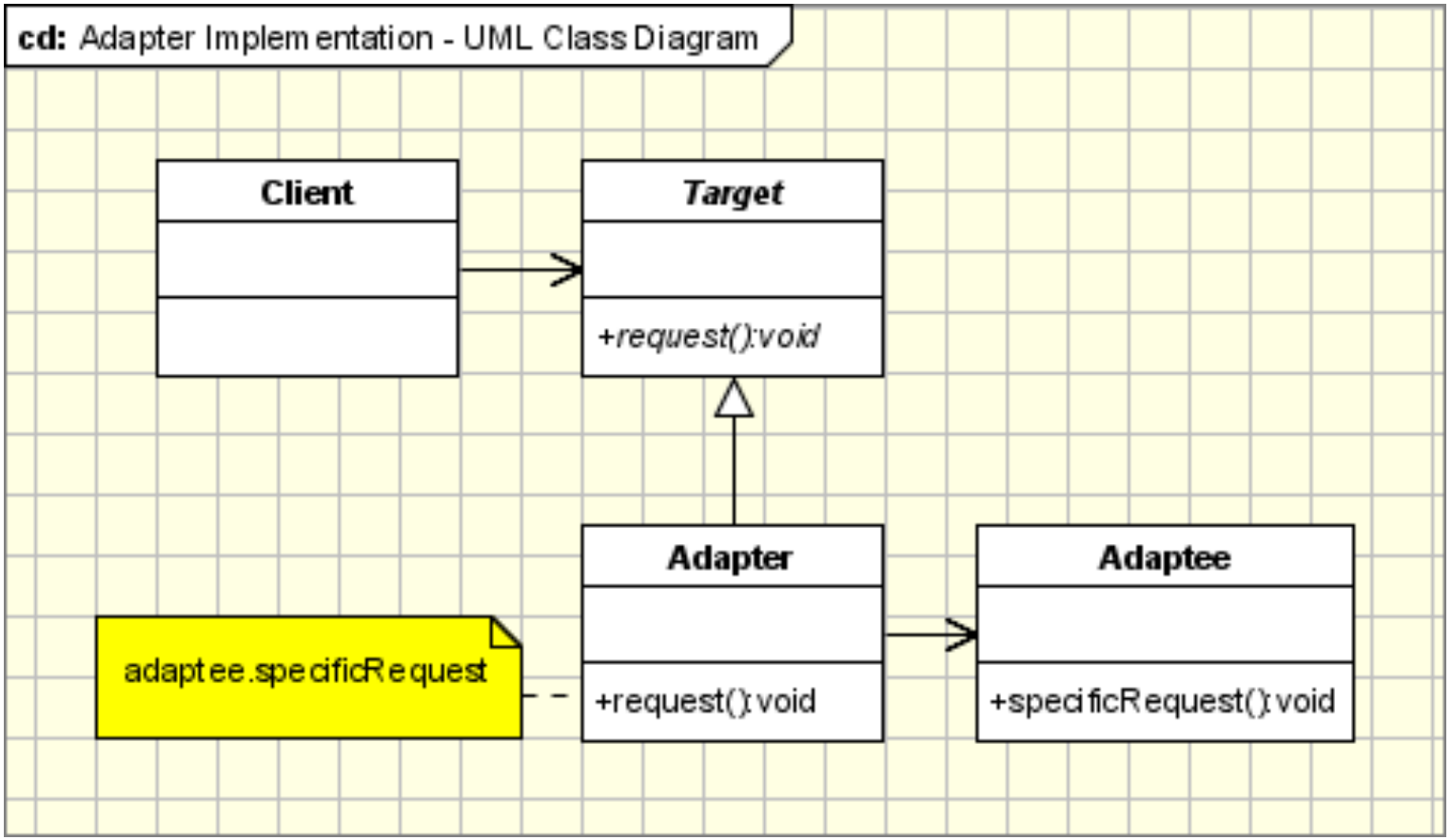
- Applicabilità
  - utilizzo di una classe esistente che presenti un'interfaccia diversa da quella desiderata
  - scrittura di una determinata classe senza poter conoscere a priori le altre classi con cui dovrà operare

Adapter



Esemplificazione del pattern `Adapter`

Adapter - Struttura

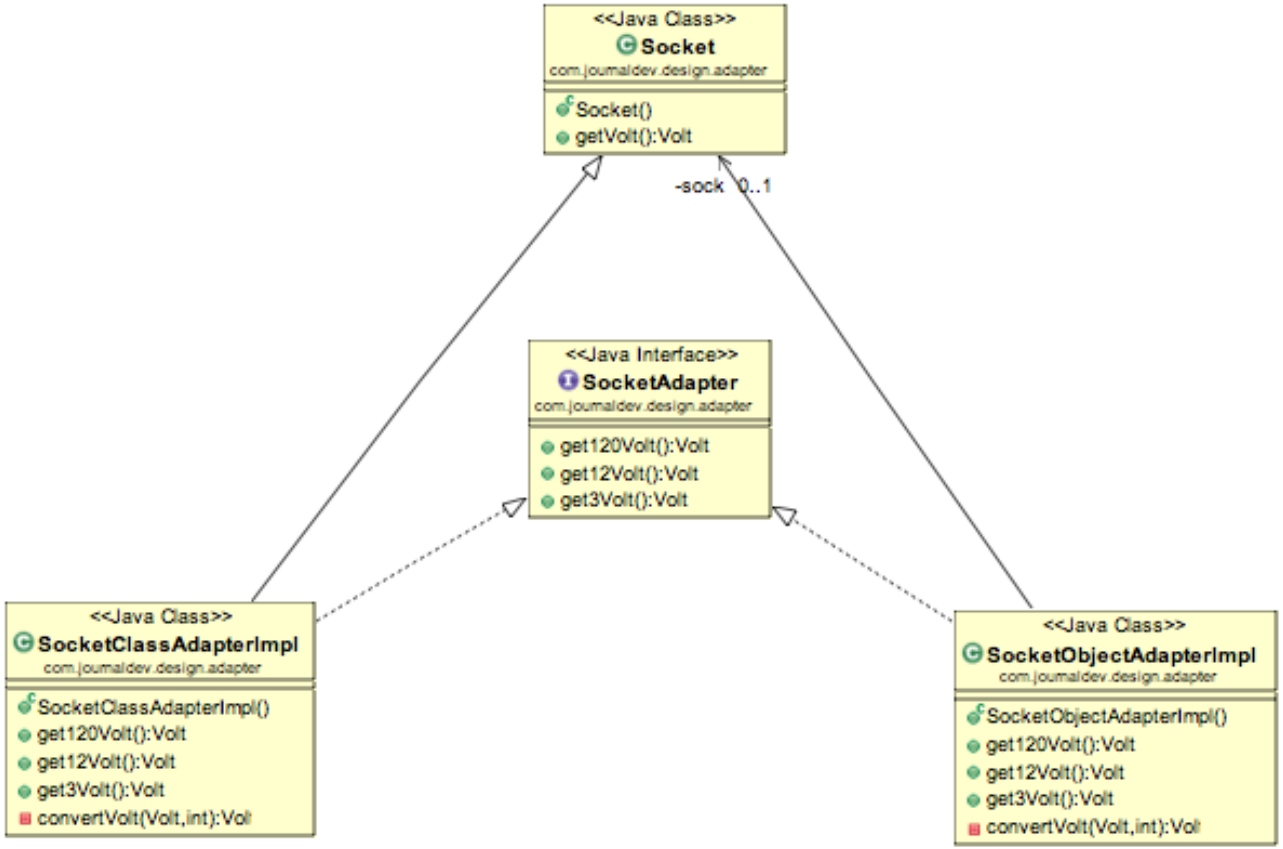


Struttura del pattern `Adapter`

Adapter - Esempio

- Adattatore per corrente
  - socket che produce 120 Volt
  - costruire un adattatore che produce 3 Volt, 12 Volt e 120 Volt

Adapter



Esempio di implementazione del pattern `Adapter`

Codice di riferimento

`Socket (directory)`

Esempio Adapter

```
class LegacyLine
{
    public void draw(int x1, int y1, int x2, int y2)
    {
        System.out.println("line from (" + x1 + ',' + y1
+ ") to (" + x2 + ','
          + y2 + ')');
    }
}
```

Esempio di implementazione del pattern `Adapter`. Shape (senza `Adapter`).

Codice di riferimento

```
Shape (directory)
```


Esempio Adapter

```
class LegacyRectangle
{
    public void draw(int x, int y, int w, int h)
    {
        System.out.println("rectangle at (" + x + ',' + y
+ ") with width " + w
            + " and height " + h);
    }
}
```


Esempio di implementazione del pattern `Adapter`. (senza `Adapter`).

Esempio Adapter

```
public class AdapterDemo
{
    public static void main(String[] args)
    {
        Object[] shapes =
        {
            new LegacyLine(), new LegacyRectangle()
        };
        // A begin and end point from a graphical editor
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i)
          if
(shapes[i].getClass().getName().equals("LegacyLine"))
            ((LegacyLine)shapes[i]).draw(x1, y1, x2, y2);
          else if
(shapes[i].getClass().getName().equals("LegacyRectangle"))
            ((LegacyRectangle)shapes[i]).draw(Math.min(x1, x2),
Math.min(y1, y2)
              , Math.abs(x2 - x1), Math.abs(y2 - y1));
    }
}
```

Esempio di implementazione del pattern `Adapter`. Shape (senza `Adapter`).

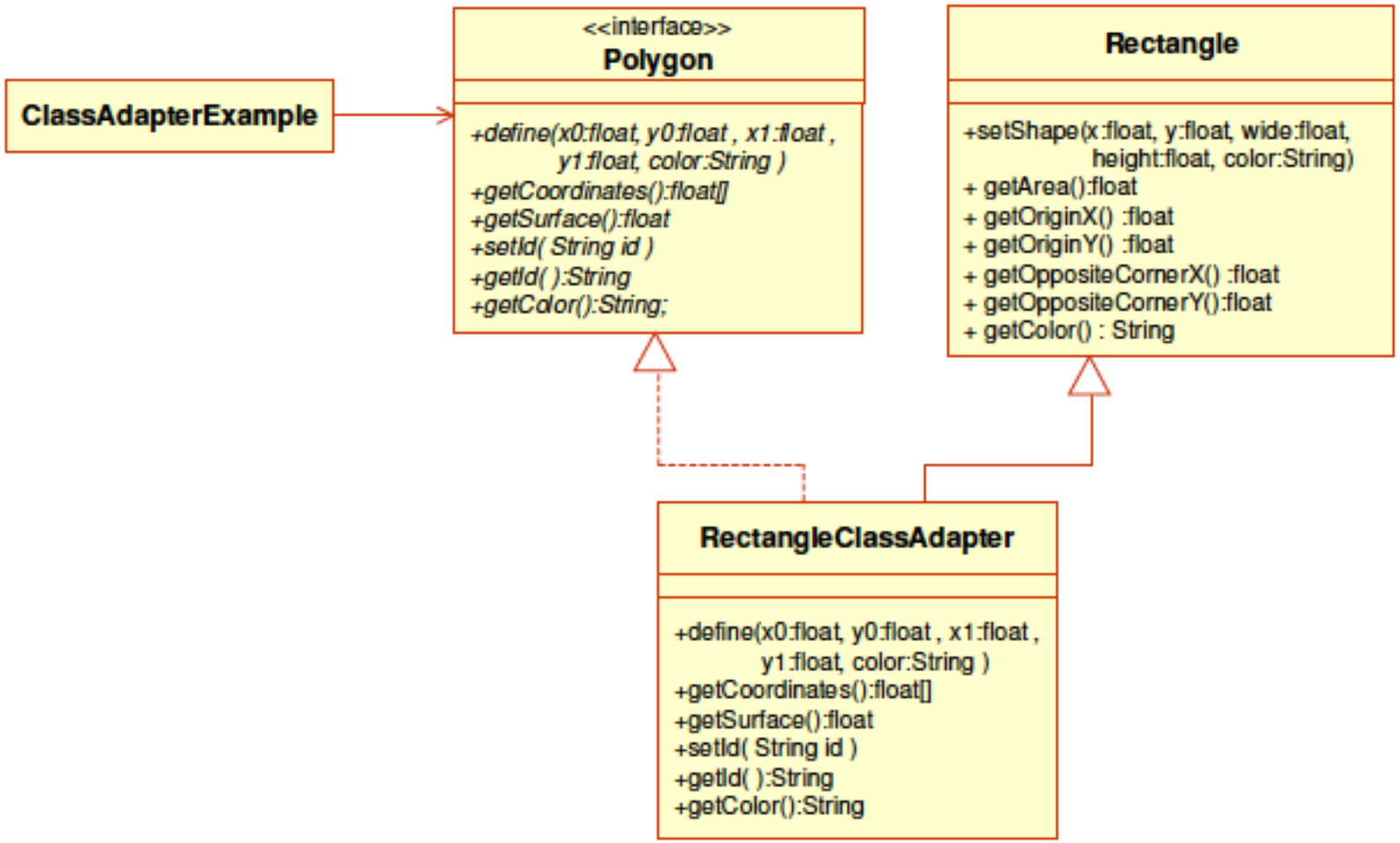
Esempio Adapter

```
interface Shape
{
  void draw(int x1, int y1, int x2, int y2);
}
class Line implements Shape
{
    private LegacyLine adaptee = new LegacyLine();
    public void draw(int x1, int y1, int x2, int y2)
    {
        adaptee.draw(x1, y1, x2, y2);
    }
}

class Rectangle implements Shape
{
    private LegacyRectangle adaptee = new LegacyRectangle();
    public void draw(int x1, int y1, int x2, int y2)
    {
    adaptee.draw(Math.min(x1, x2), Math.min(y1, y2), Math.abs(x2
- x1), Math.abs(y2 - y1));
    }
}
```

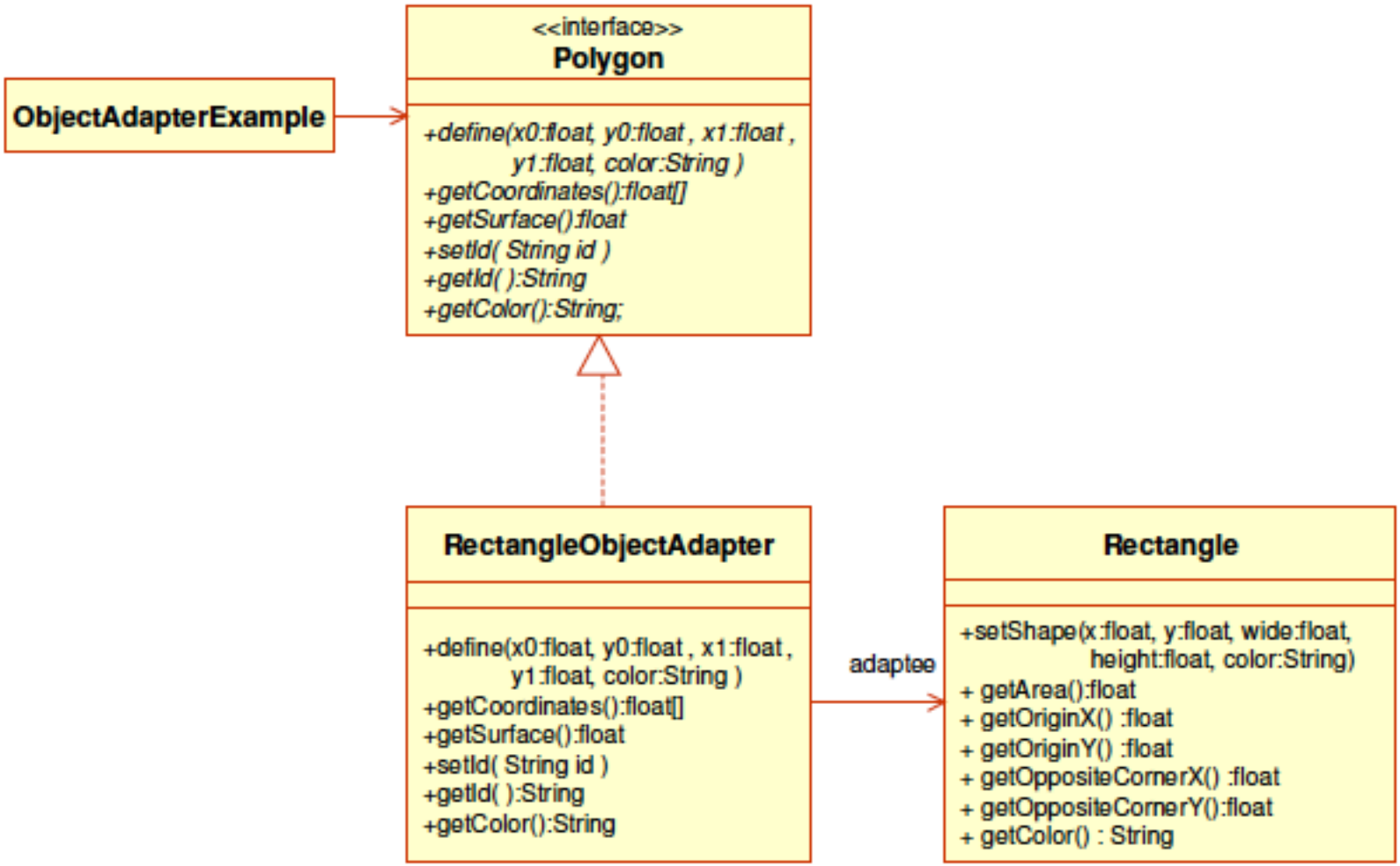
Esempio di implementazione del pattern `Adapter`. Shape.

Esercizi



Esercizio pattern `Adapter (Class)`

Esercizi


ObjectAdapterExample
<<interface>>
Polygon
+define(x0:float, y0:float , x1:float , y1:float, color:String)
+getCoordinates():float[]
+getSurface():float
+setId(String id)
+getId( ):String
+getColor():String;
RectangleObjectAdapter
+define(x0:float, y0:float , x1:float , y1:float, color:String)
+getCoordinates():float[]
+getSurface():float
+setId(String id)
+getId( ):String
+getColor():String
adaptee
Rectangle
+setShape(x:float, y:float, wide:float, height:float, color:String)
+ getArea():float
+ getOriginX() :float
+ getOriginY() :float
+ getOppositeCornerX() :float
+ getOppositeCornerY():float
+ getColor() : String

Considerazioni

- Adapter è usato in (JDK)
  - java.util.Arrays#asList()
  - java.io.InputStreamReader(InputStream) (returns a Reader)
  - java.io.OutputStreamWriter(OutputStream) (returns a Writer)

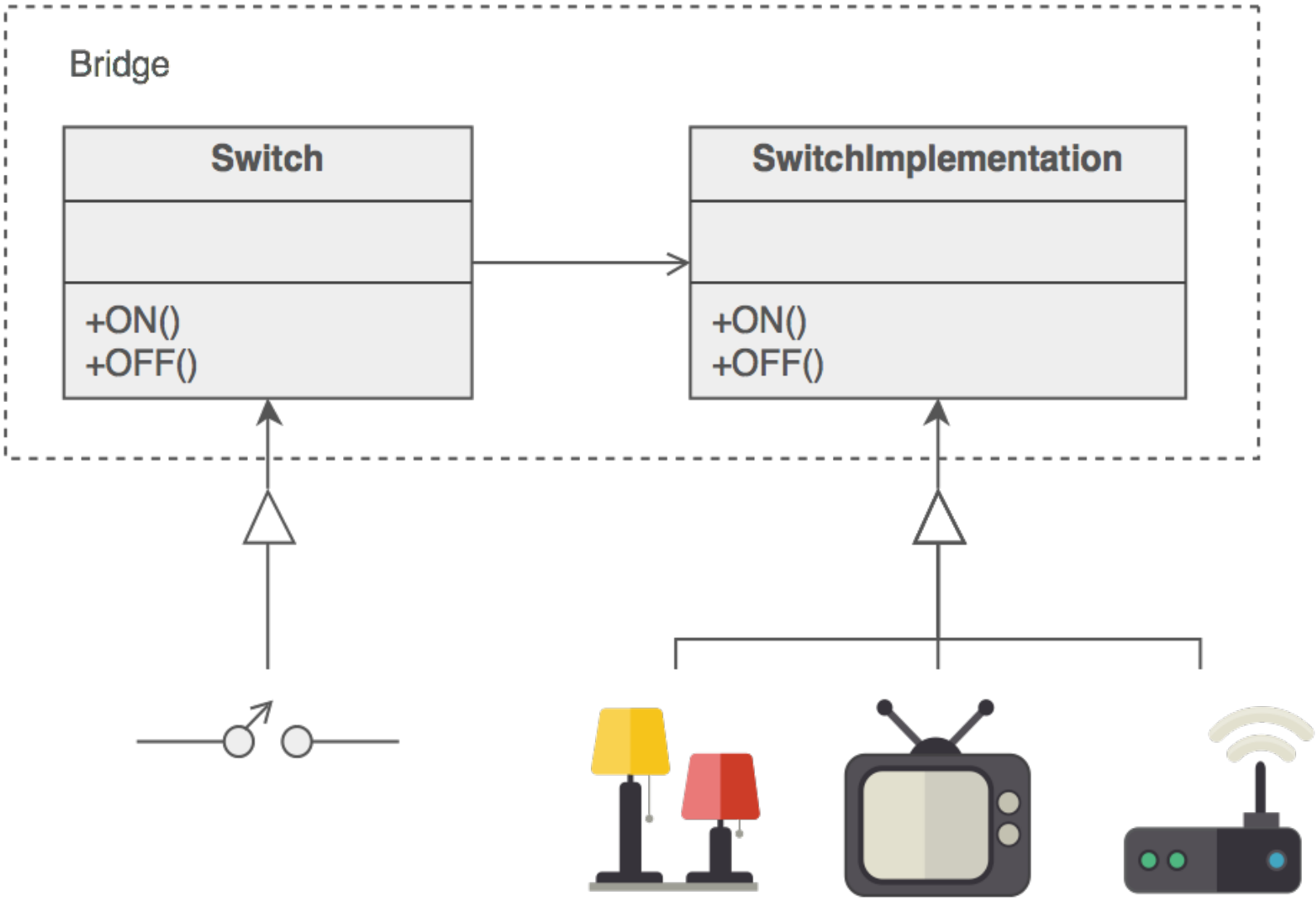
Bridge

- Scopo
  - *Separa un'astrazione dalla sua implementazione, in modo che entrambe possano variare indipendentemente.*
- Anche conosciuto come
  - Handle/Body
- Motivazione
  - Spesso un'astrazione deve avere diverse implementazioni
    - gestione di database relazionali o file system
      - soluzione - ereditarietà
    - l'ereditarietà lega un'implementazione all'astrazione

Bridge

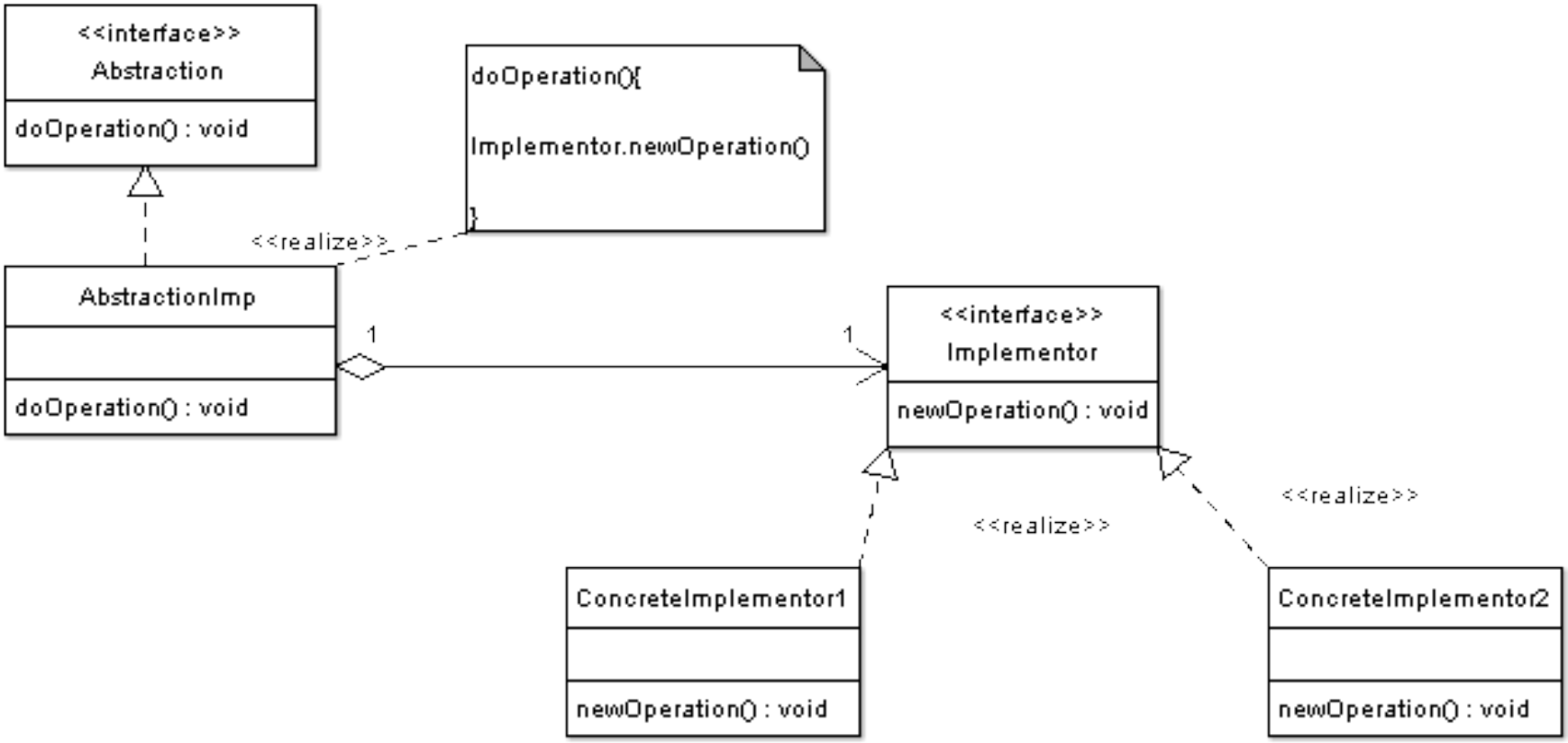
- Applicabilità
  - necessità di evitare un collegamento permanente tra l'astrazione e l'implementazione
  - quando l'astrazione e l'implementazione hanno bisogno di cambiare indipendentemente
  - usando il pattern è possibile lasciare il codice del client inalterato senza la necessità di ricompilarlo
  - preferire la composizione all'eredità

Bridge



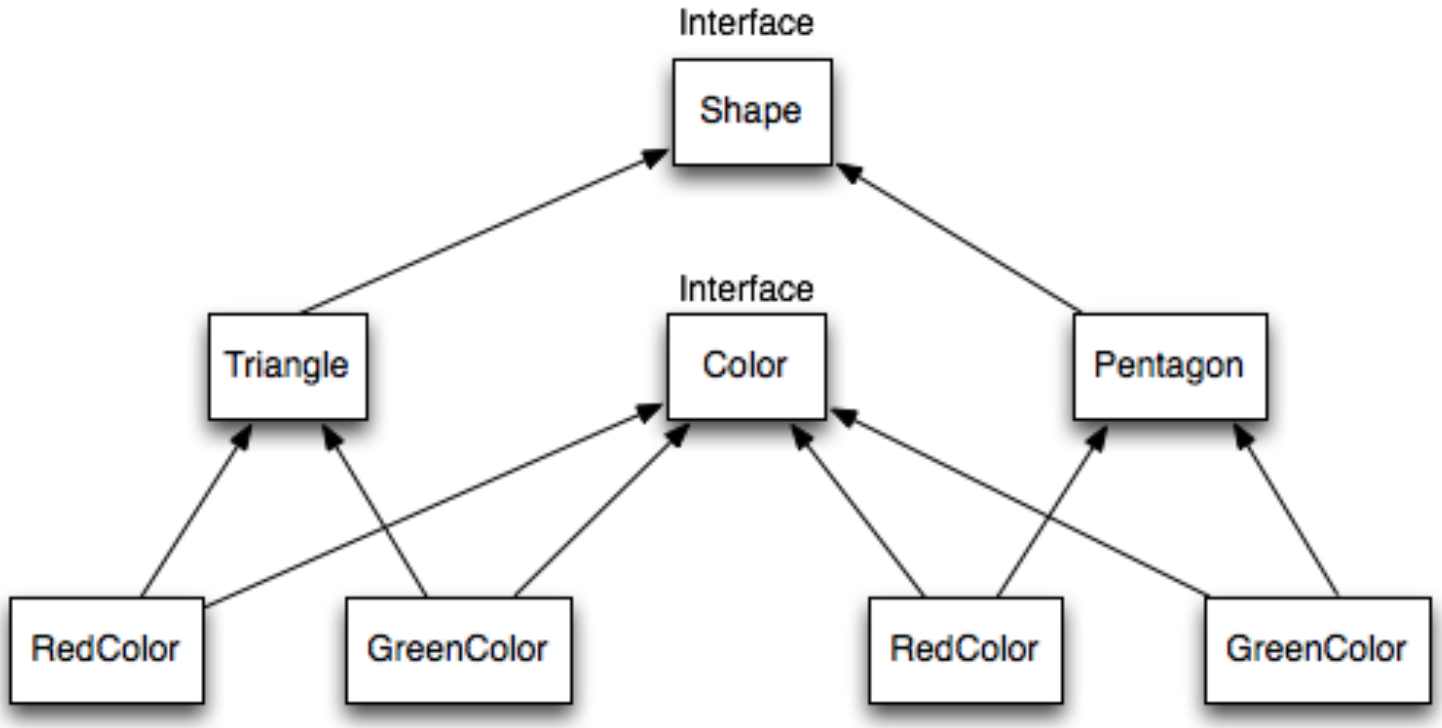
Esemplificazione del pattern `Bridge`

Bridge - Struttura



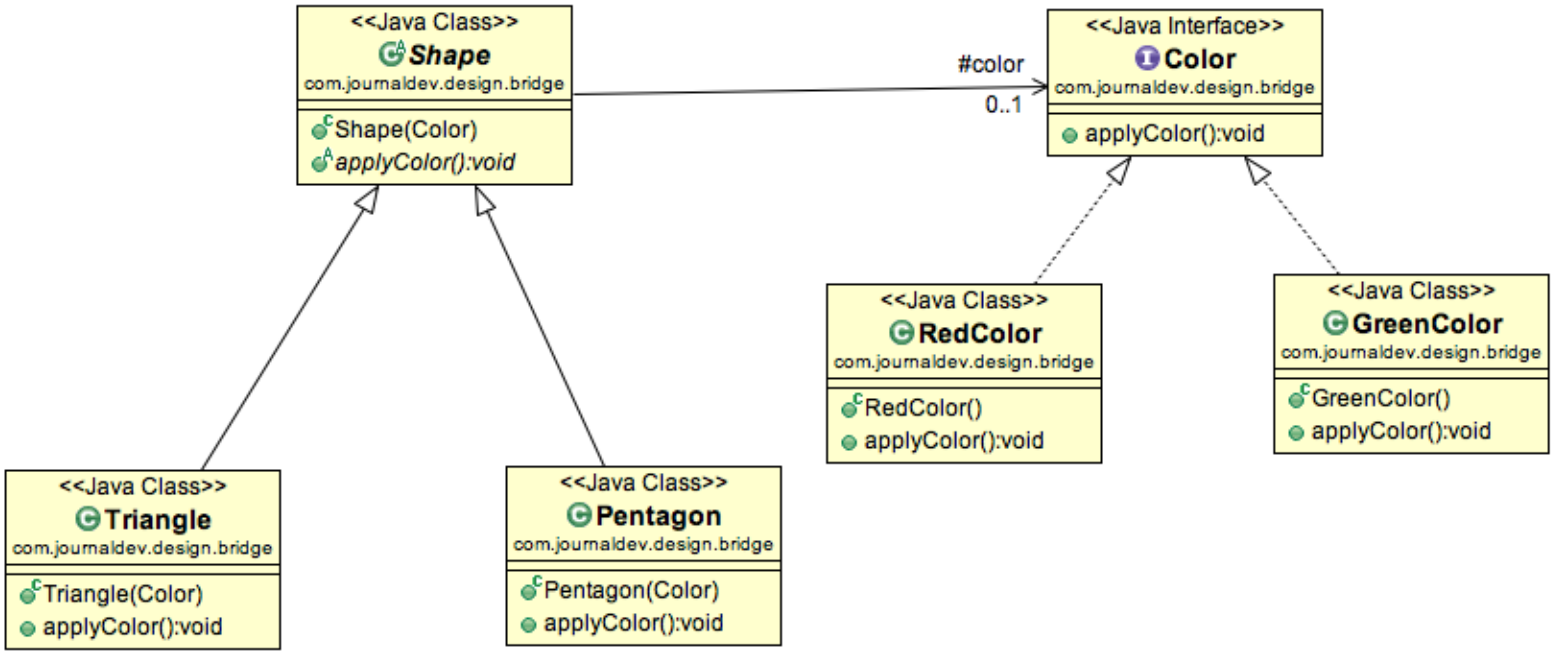
Struttura del pattern `Bridge`

Esempio Bridge



Esempio di implementazione del pattern `Adapter`. Shape and Color.

Esempio Bridge

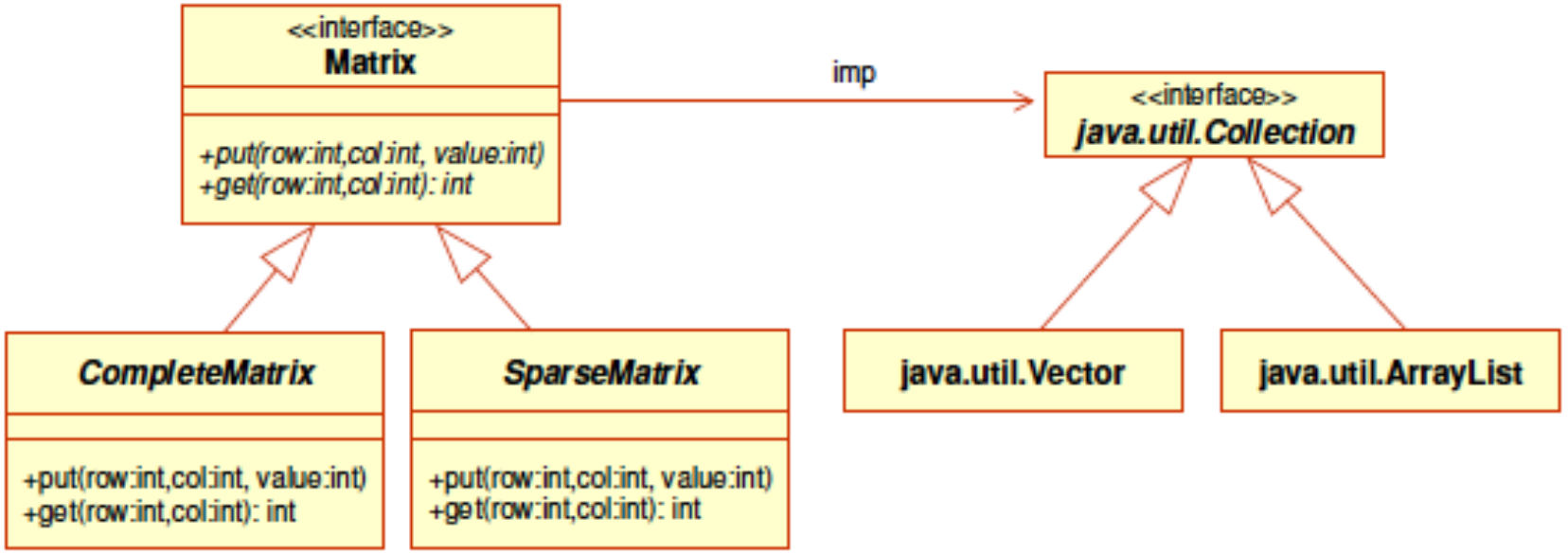


Esempio di implementazione del pattern `Bridge`. Shape and Color.

Codice di riferimento

`Shape_and_Color (directory)`

Esercizio



Esercizio `Bridge`. Matrici.

Composite

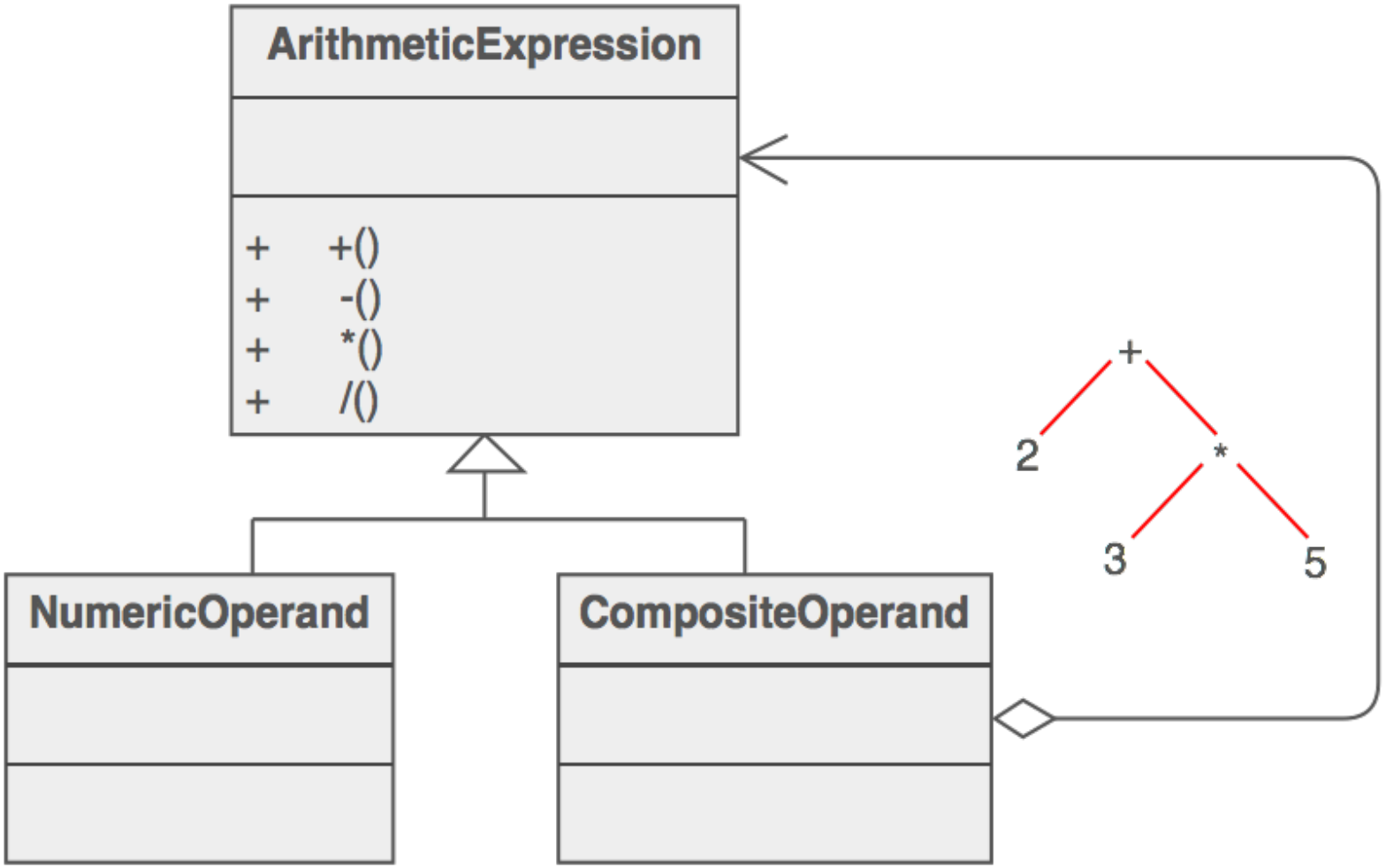
- Scopo
  - *Comporre oggetti una struttura ad albero per rappresentare gerarchie (whole-part). Il pattern permette ai client di trattare oggetti singoli e composizioni di oggetti uniformi.*

- Motivazione
  - applicazioni hanno bisogno di manipolare una collezione di oggetti primitivi o composti
    - se l'oggetto desiderato è una Leaf, la richiesta è processata direttamente
    - se è una Composite, viene rimandata ai figli cercando di svolgere le operazioni prima

Composite

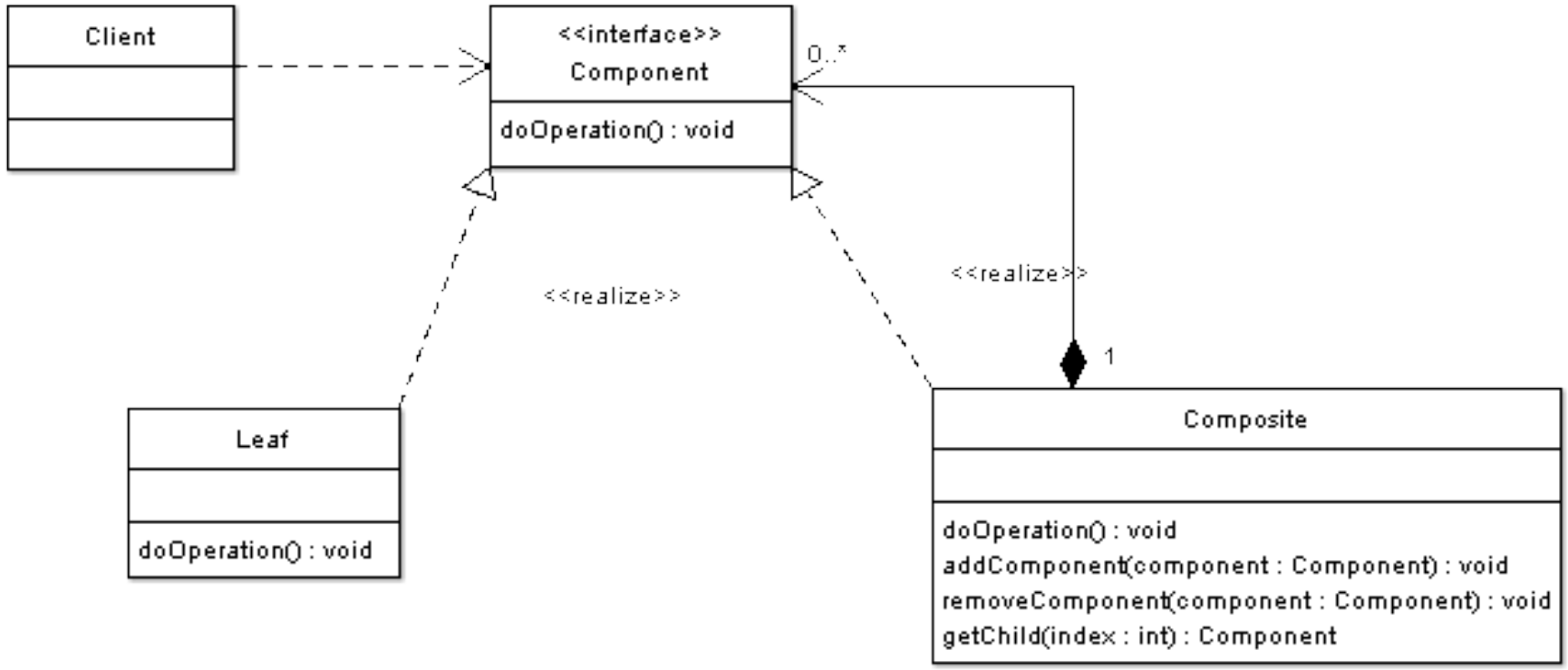
- Applicabilità
  - quando i client dovrebbero ignorare la differenza tra oggetti composti e oggetti singoli
  - scelta di rifattorizzazione
    - durante lo sviluppo i programmatori scoprono che stanno usando più oggetti nello stesso modo e spesso il codice per gestirli è molto simile

Composite



Esemplificazione del pattern `Composite`

Composite - Struttura



Struttura del pattern `Composite`

Esercizio

- Drawing Scenario
  - Componenti base
    - Class Shape
  - Composti
    - Drawing

Realizzare il diagramma delle classi

Codice di riferimento

`Drawing_Shape (directory)`

Considerazioni

- Usi conosciuti
  - Implementazione dei File System
  - Editor Grafici
- E' usato in (JDK)
  - `java.awt.Container#add(Component)`
  - `Swing`

Decorator

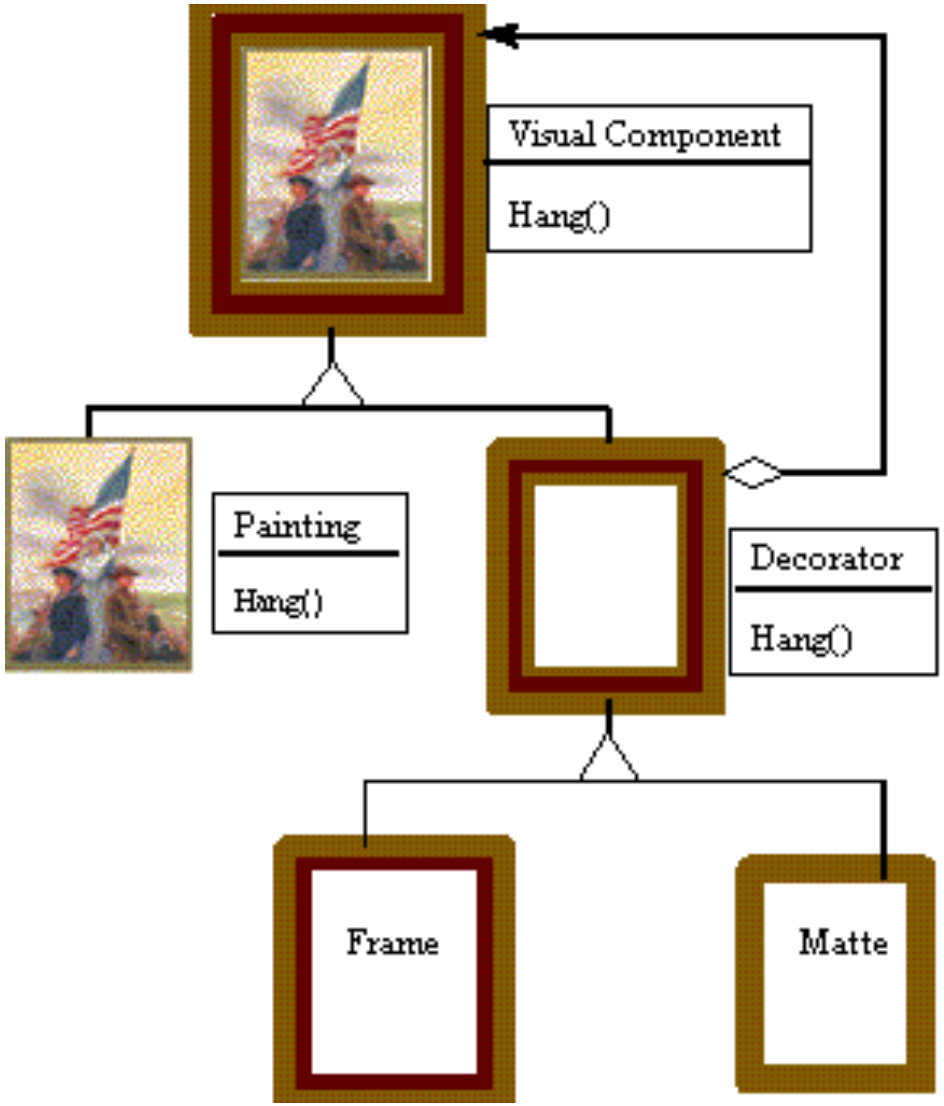
- Scopo
  - *Aggiunge responsabilità addizionali ad un oggetto dinamicamente. Fornisce un'alternativa flessibile alla costruzione di sottoclassi per estendere delle funzionalità.*

- Motivazione
  - consente di aggiungere durante il run-time nuove funzionalità ad oggetti già esistenti
    - nuova classe decoratore che "avvolge" l'oggetto originale
  - valida alternativa all'uso dell'ereditarietà singola o multipla

Decorator

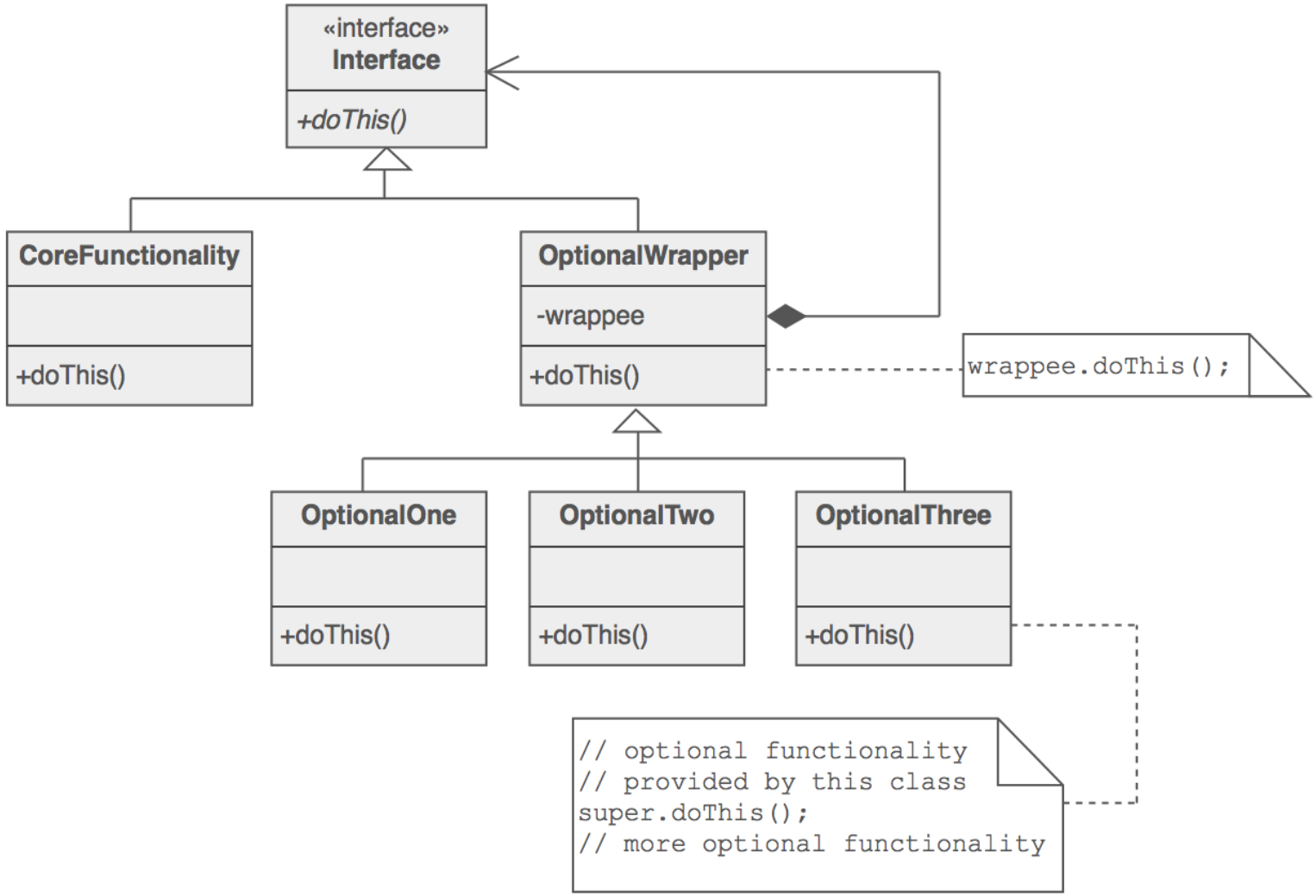
- Applicabilità
  - quando vogliamo aggiungere comportamenti o stati ad oggetti individuali a run-time
  - l'eredtarietà non è flessibile poiché è statica ed è applicata ad una intera classe

Decorator



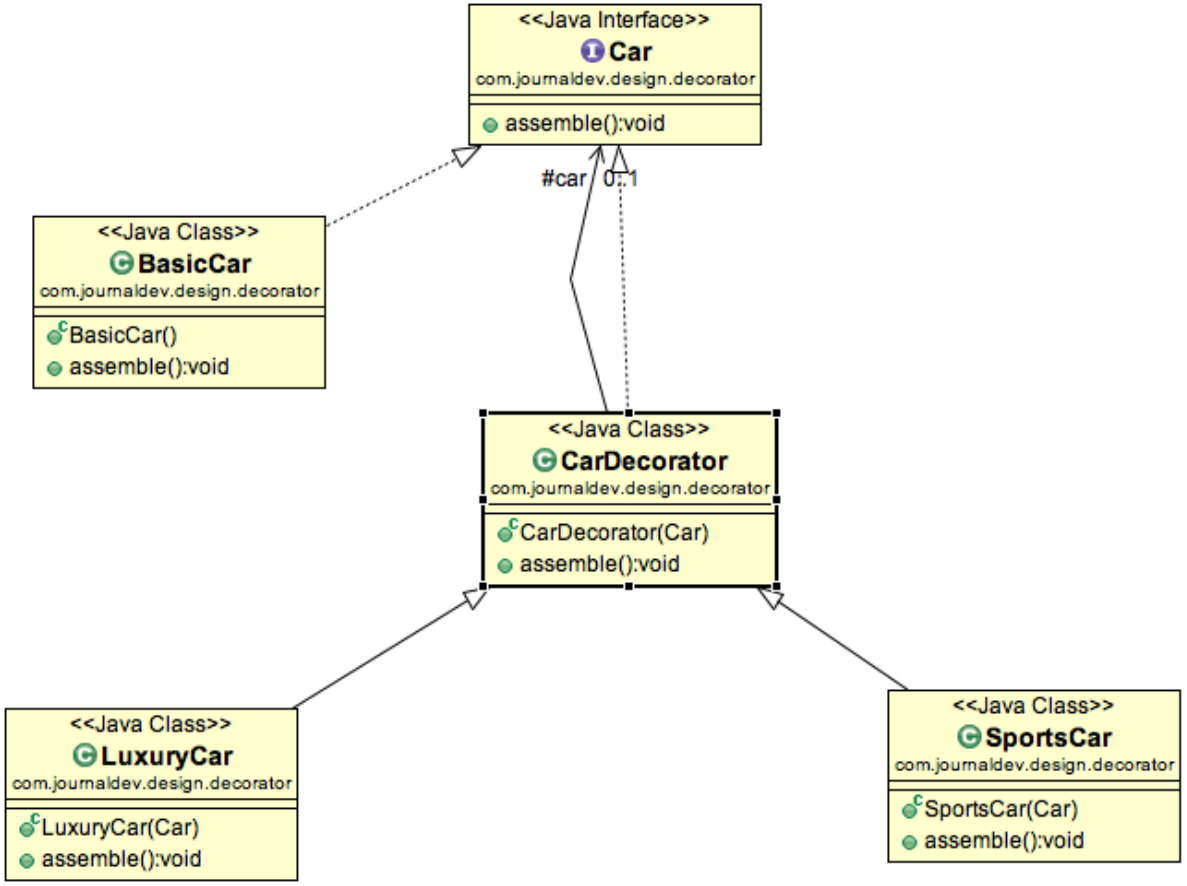
Esemplificazione del pattern `Decorator`

Decorator - Struttura



Struttura del pattern `Decorator`

Esempio

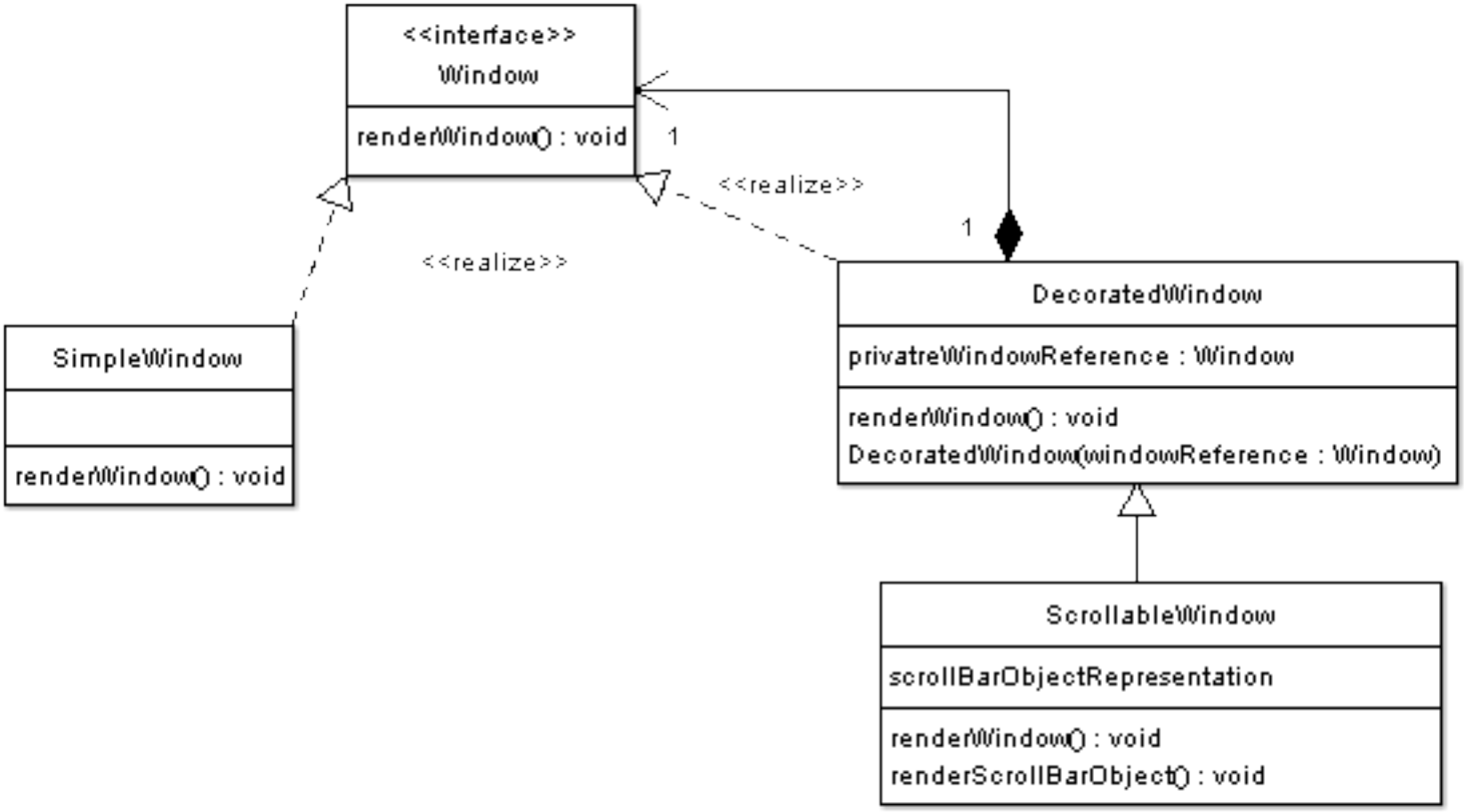


Esempio del pattern `Decorator`

Codice di riferimento

`Car (directory)`

Esercizio


<<interface>>
Window
renderWindow() : void
1
<<realize>>
1
<<realize>>
SimpleWindow
renderWindow() : void
DecoratedWindow
privatreWindowReference : Window
renderWindow() : void
DecoratedWindow(windowReference : Window)
ScrollableWindow
scrollBarObjectRepresentation
renderWindow() : void
renderScrollBarObject() : void

Considerazioni

- Usi conosciuti
  - GUI
  - File Systems
- E' usato in (JDK)
  - `Java IO`
    - `FileReader`
    - `BufferedReader`

Facade

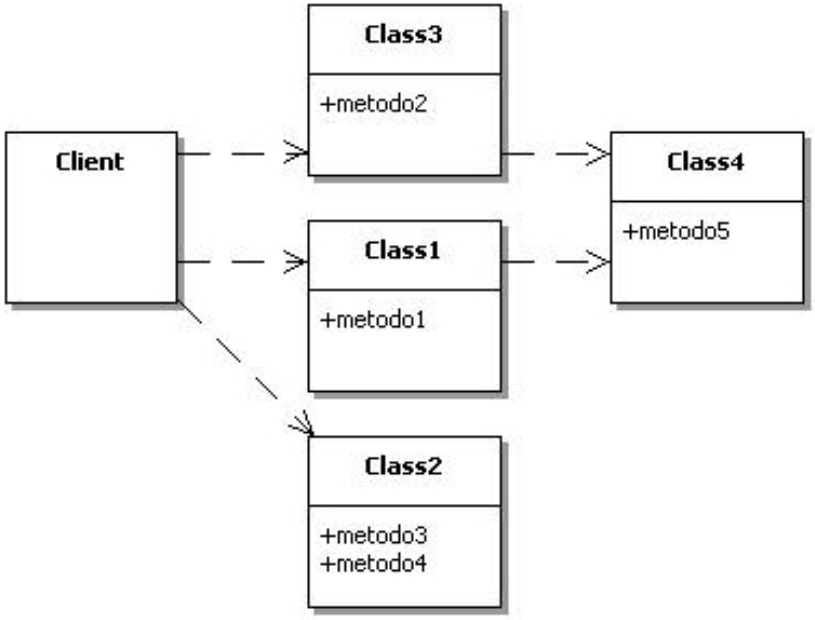
- Scopo
  - *Fornisce un'interfaccia unificata ad un insieme di interfacce in un sottosistema. Definisce un'interfaccia di alto livello che permette la facile gestione del sottosistema.*

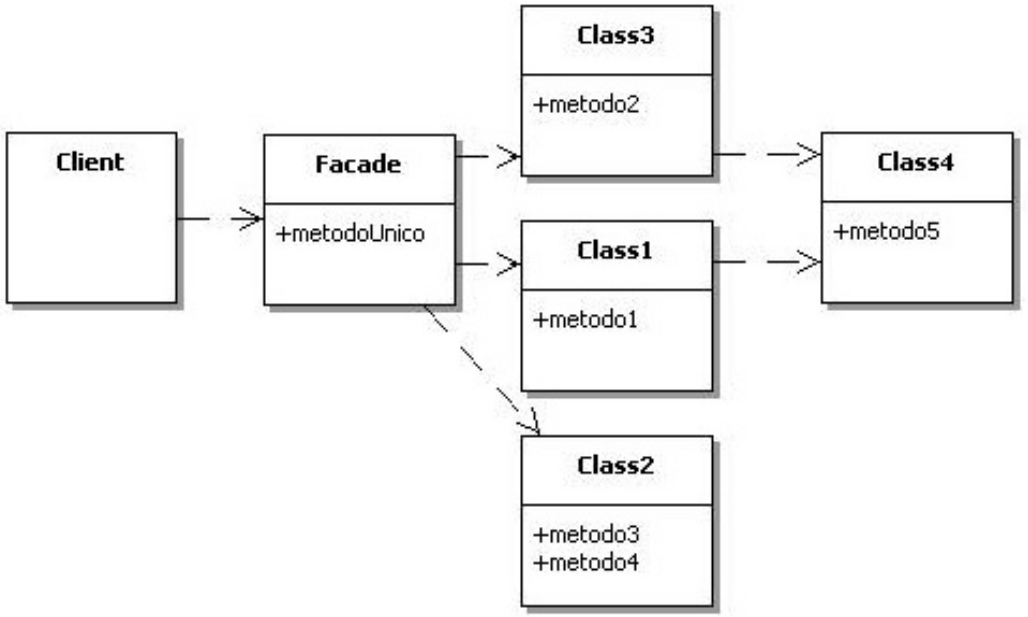
- Motivazione
  - permettere attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse e molto diverse tra loro, nonché a blocchi di codice complessi

Facade

- Applicabilità
  - quando vogliamo fornire un'interfaccia semplice ad un sottosistema complesso
  - sdoppiare client e sottosistemi
  - definire un *entry point* per ogni sottosistema

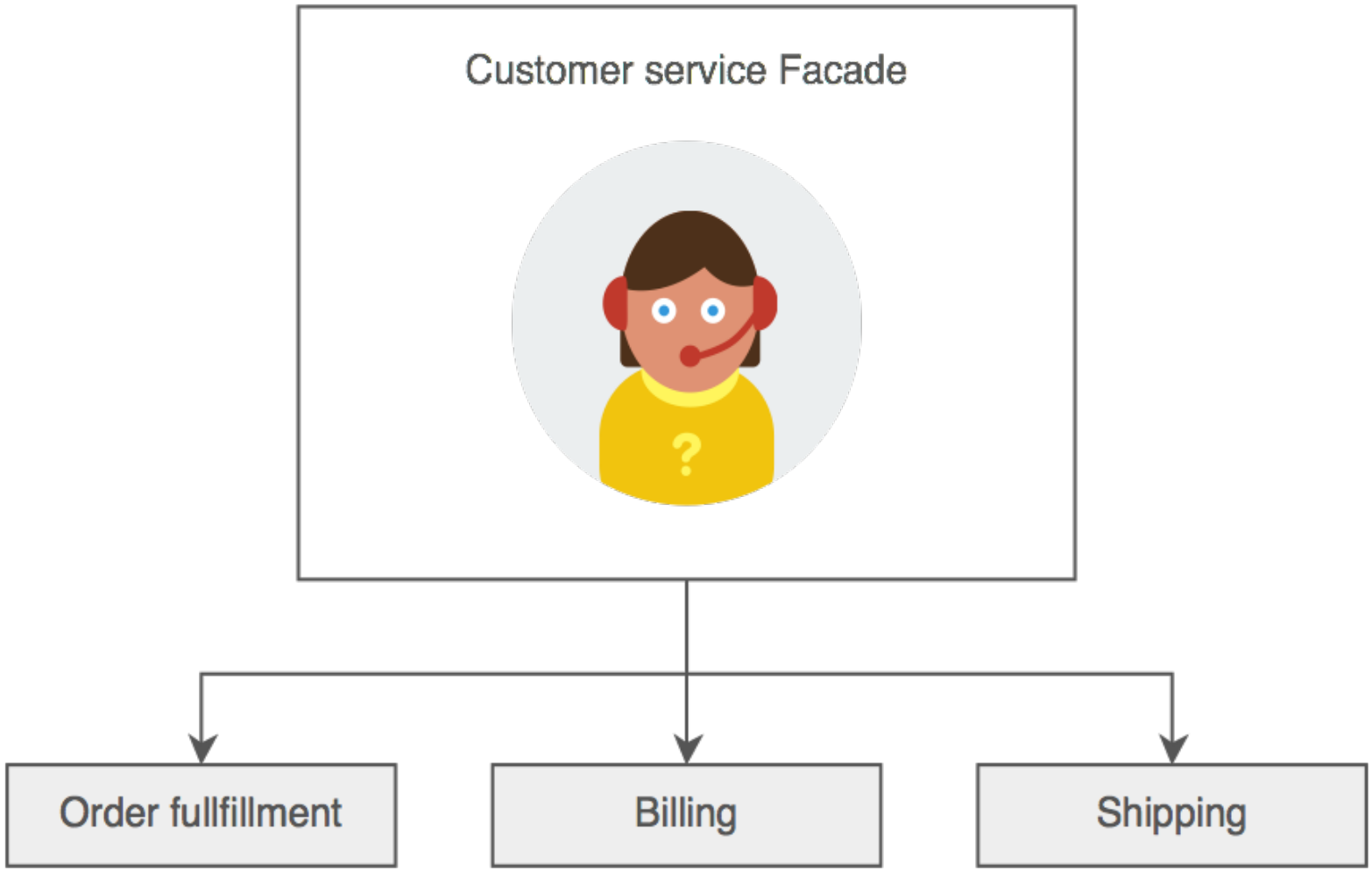
Facade





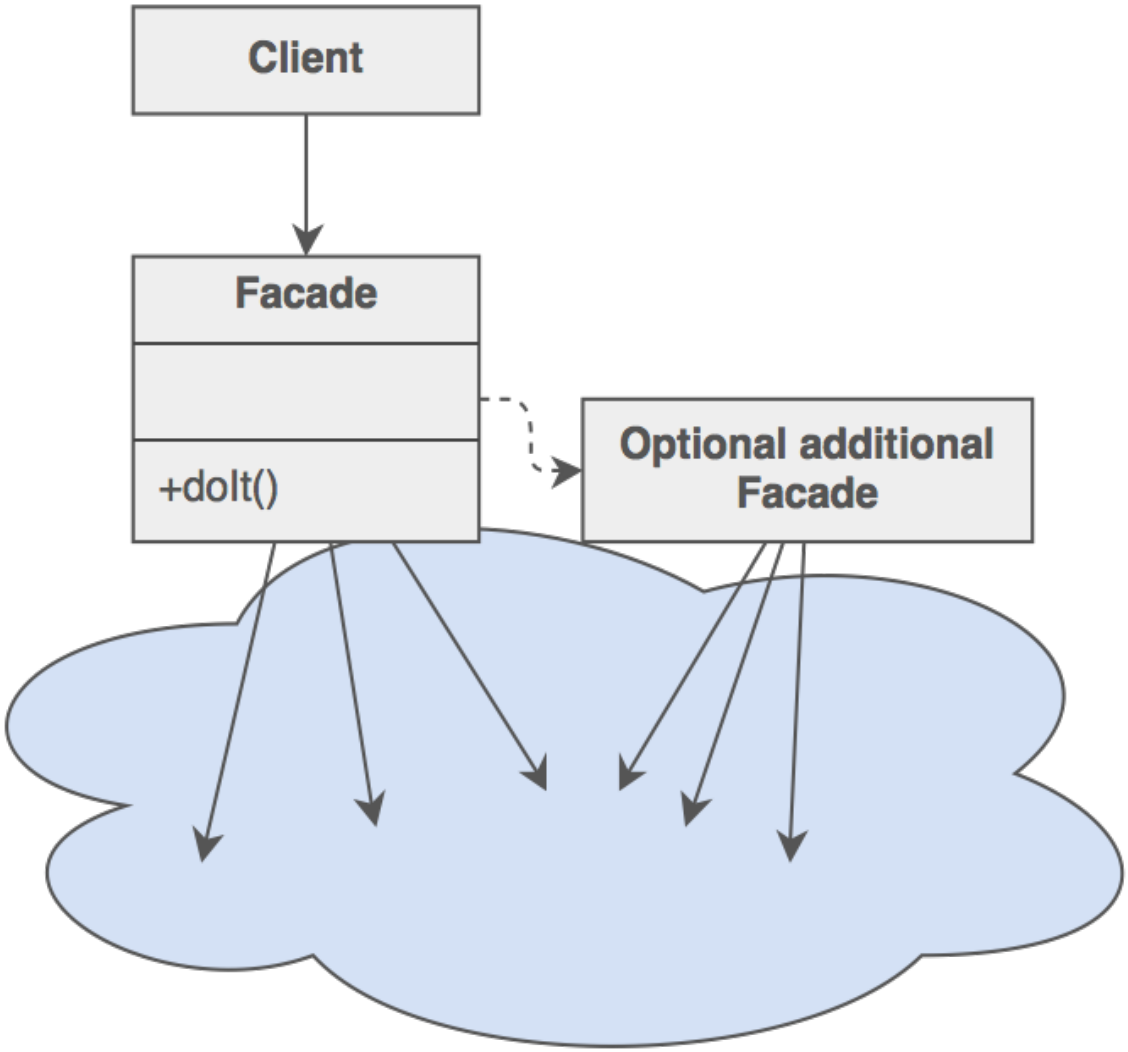
Esemplificazione del pattern `Facade`

Facade



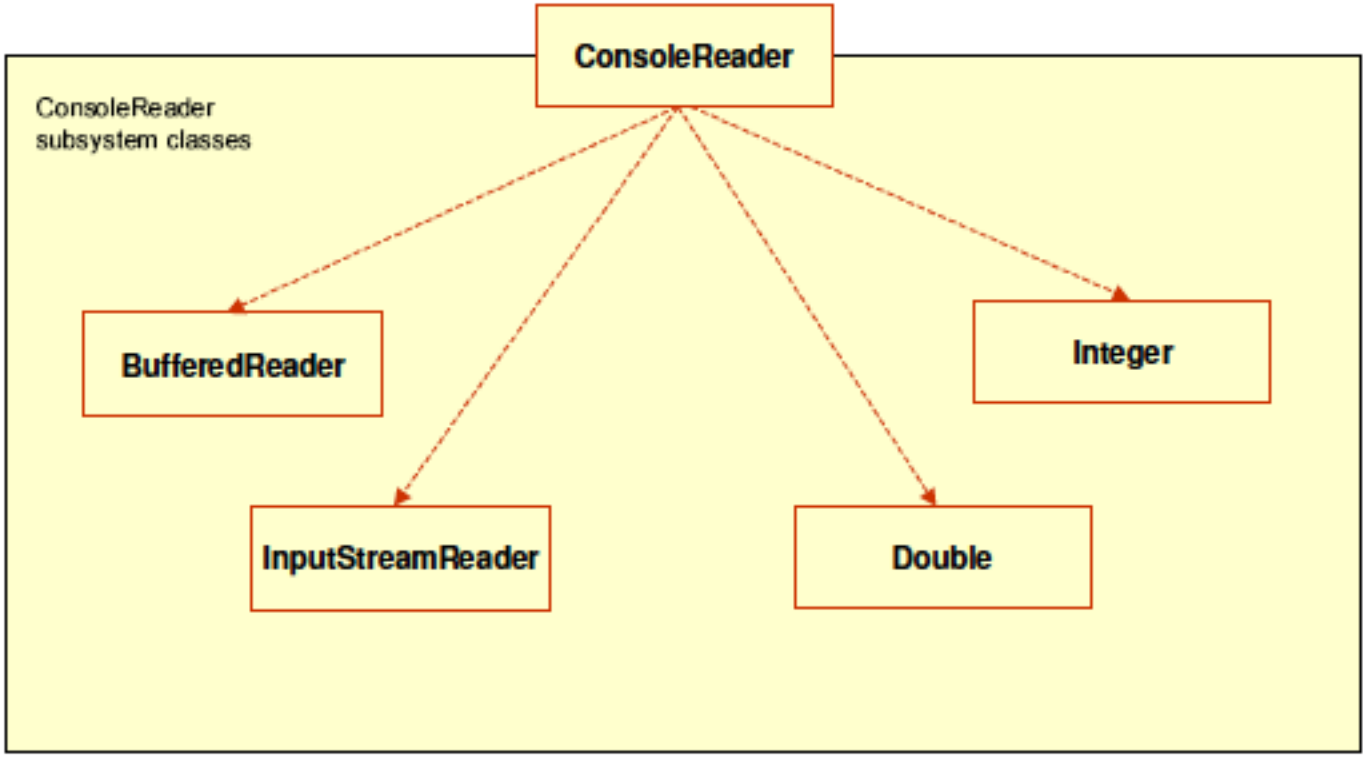
Esemplificazione del pattern `Facade`

Facade - Struttura



Struttura del pattern `Facade`

Esempio



Codice di riferimento

```
ConsoleReader (directory)
```


Flyweight

- Scopo
  - *Separare la parte variabile di una classe dalla parte che può essere riutilizzata, in modo tale da condividere quest'ultima tra differenti istanze.*

- Motivazione
  - uso della condivisione per supportare un grande numero di oggetti che hanno parti di stato interno in comune e l'altra parte dello stato può cambiare

Flyweight

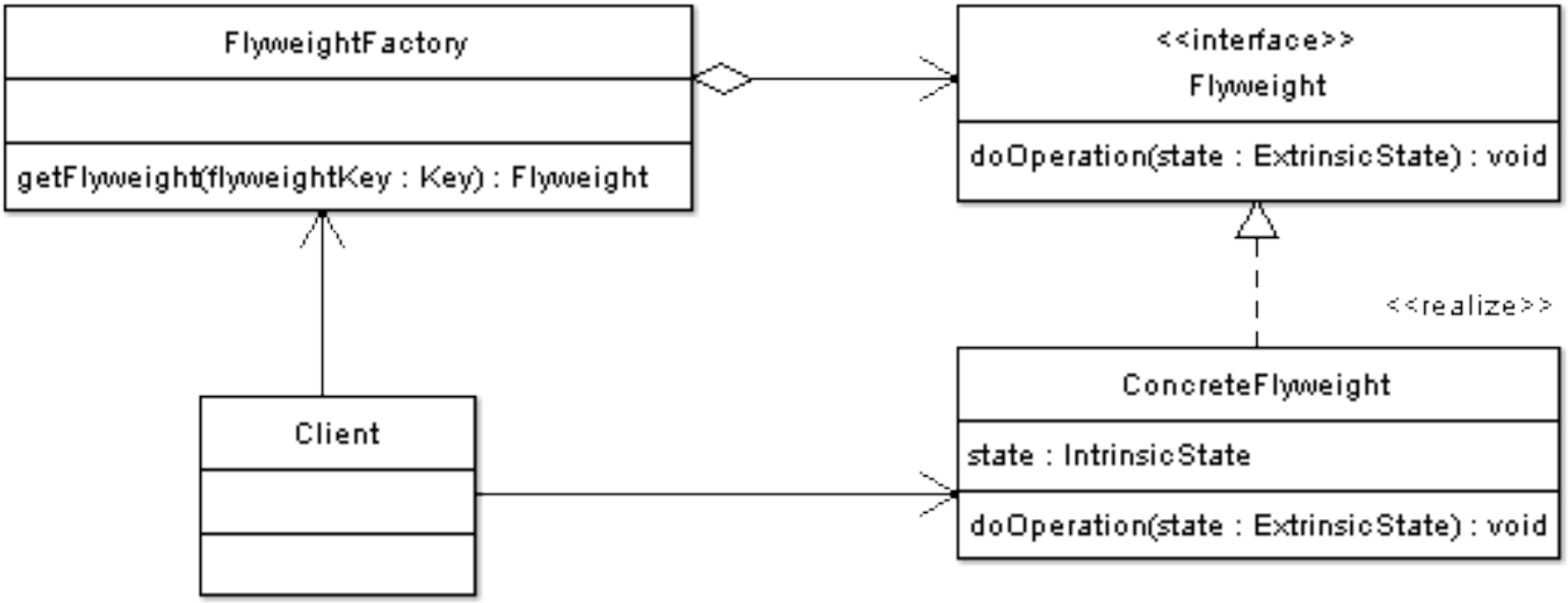
- Applicabilità
  - un'applicazione usa un grande numero di oggetti
  - i costi di memorizzazione sono alti
  - gruppi di oggetti possono essere rimpiazzati da un numero minore di oggetti condivisi

Flyweight

Browser loads images just once and then reuses them from pool:

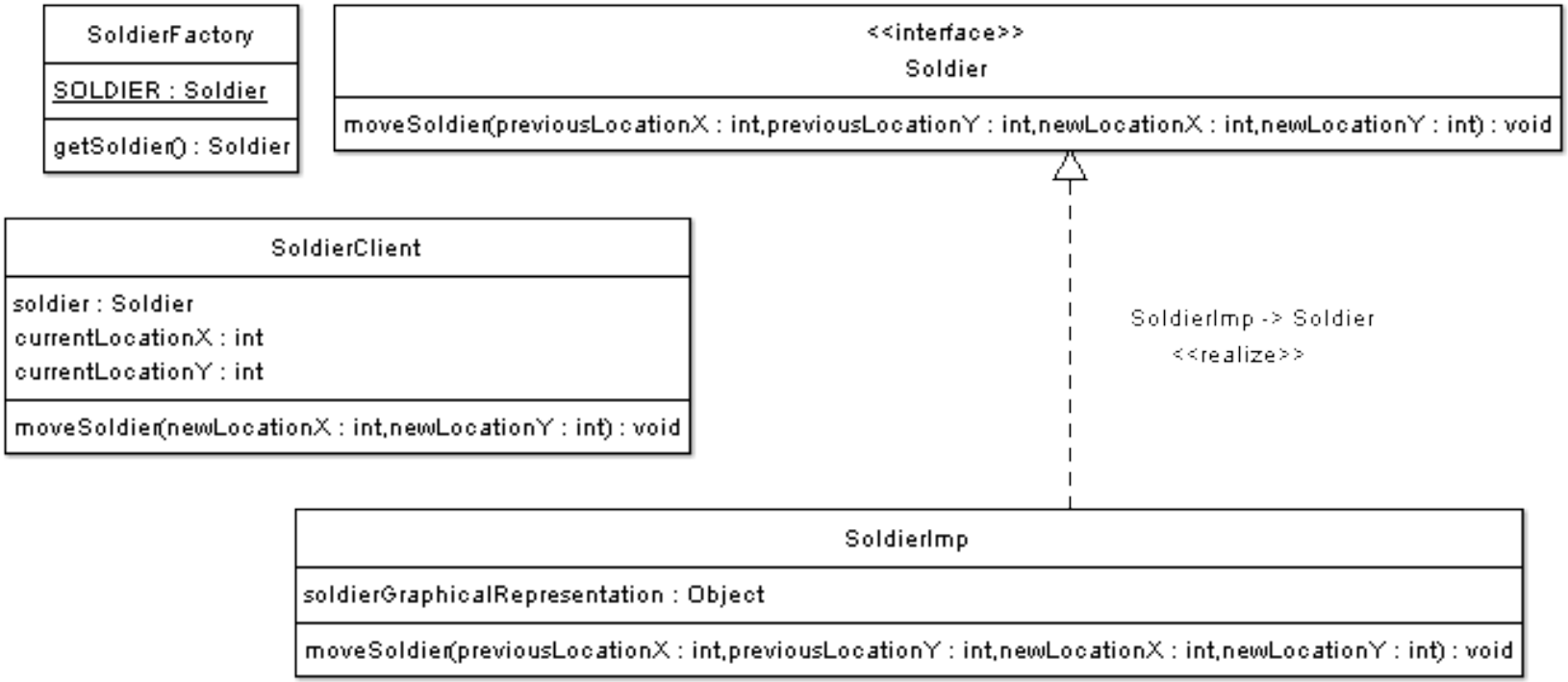
Esemplificazione caricamento di pagine Web (pattern `Flyweight`)

Facade - Struttura



Struttura del pattern `Flyweight`

Esempio



Esempio di utilizzo del pattern `Flyweight`

Codice di riferimento

`War_Game (directory)`

Considerazioni

- E' usato in (JDK)
  - tutte le classi wrapper `valueOf()` usano `Fleyweight`
  - `String Pool` in `Java String`

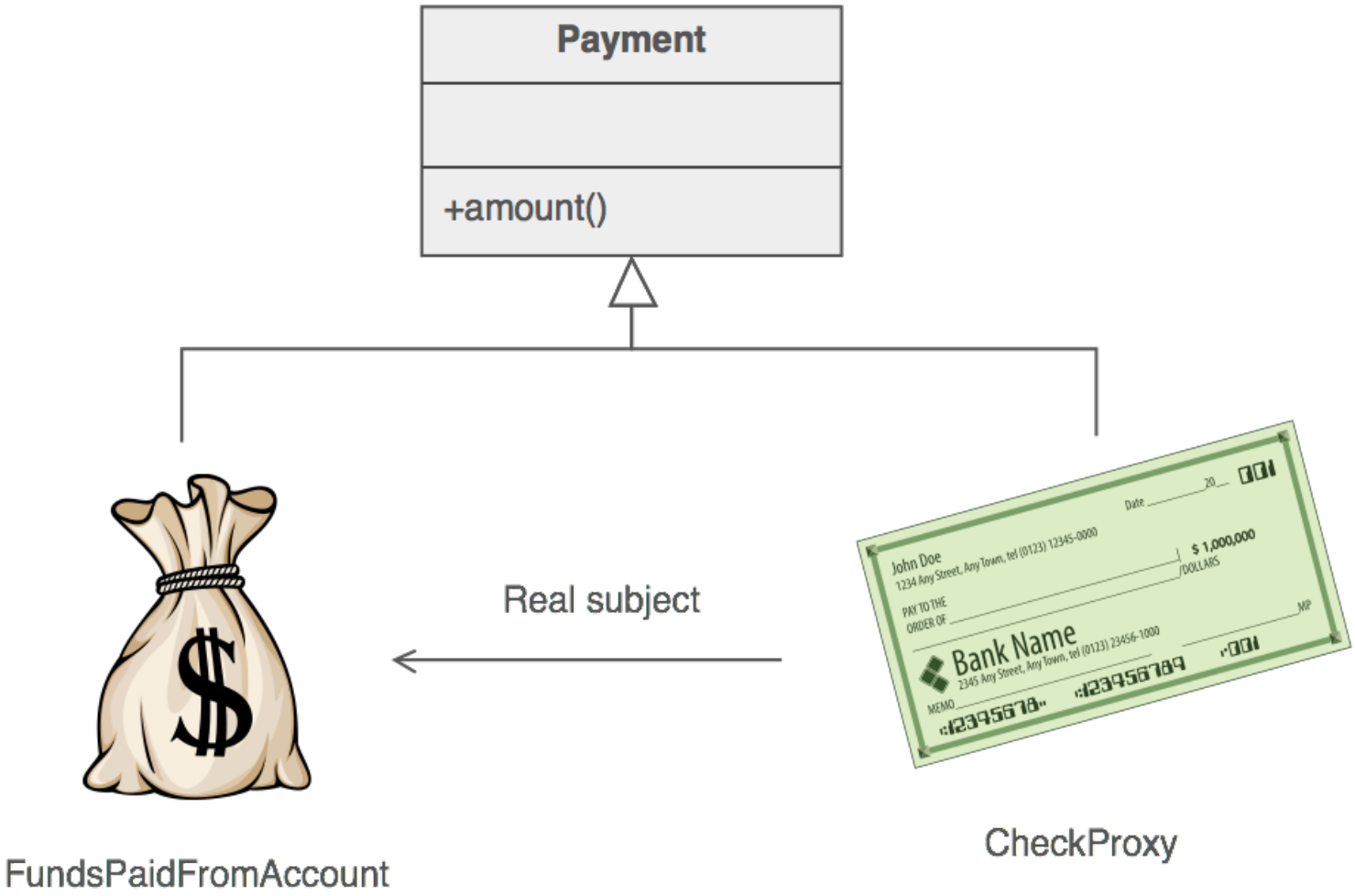
Proxy

- Scopo
  - *Fornire un surrogato (o placehoder) per un altro oggetto per controllare l'accesso ad esso.*
- Motivazione
  - classe che funziona come interfaccia per qualcos'altro
    - *connessione di rete, un grosso oggetto in memoria, un file e altre risorse che sono costose o impossibili da duplicare*

Proxy

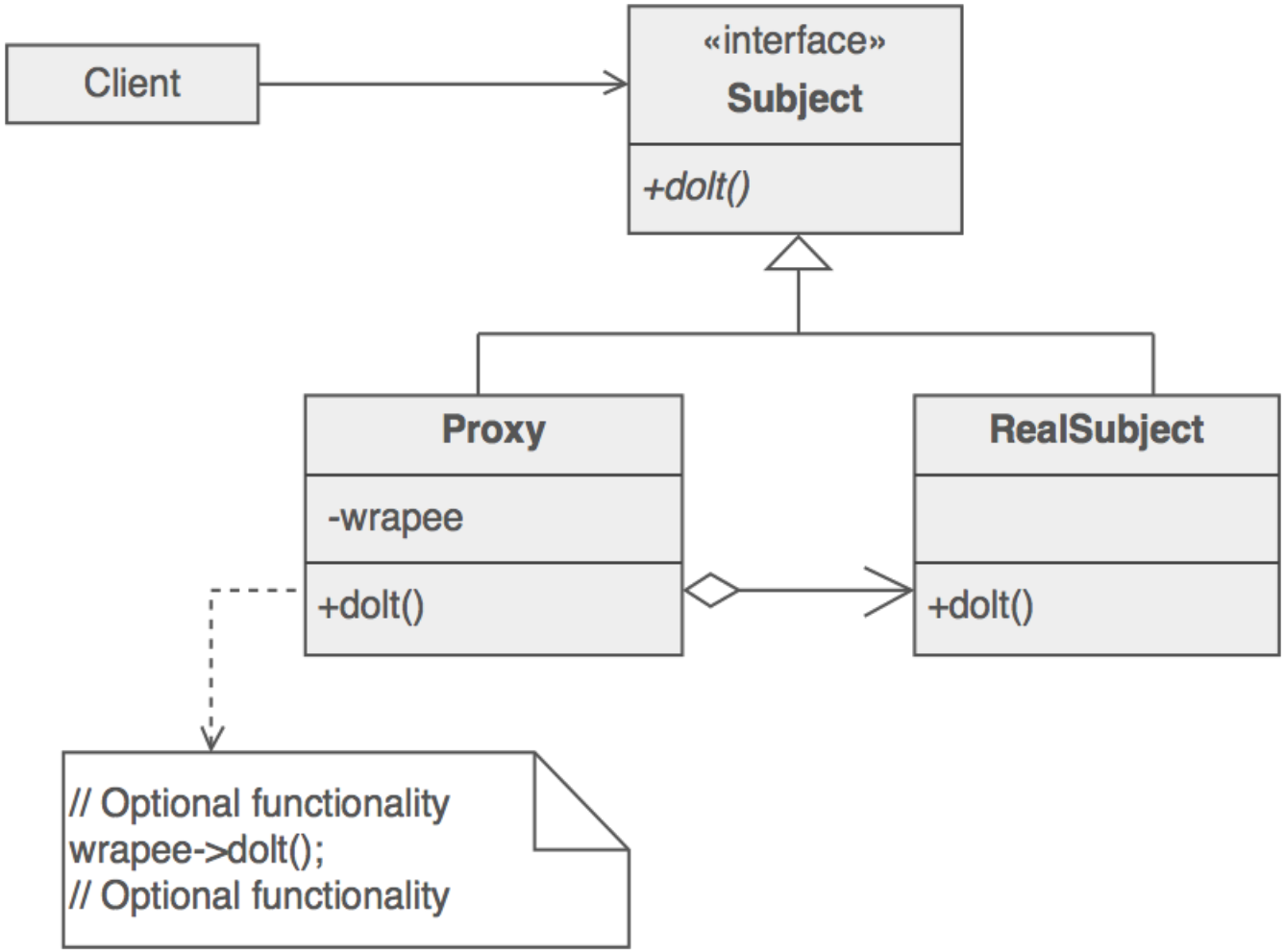
- Applicabilità
  - Remote proxy
    - fornisce una rappresentazione locale per un oggetto in una differenti spazio di indirizzi
  - Virtual Proxy
    - crea oggetti costosi su richiesta
  - Protection proxy
    - controlla l'accesso all'oggetto originale
  - Smart proxy
    - interpone azioni addizionali quando si fa accesso ad un oggetto
      - conteggio del numero di referenze
      - caricamento dell'oggetto in memoria al primo riferimento
      - controllo che l'oggetto reale non sia accessibile

Proxy



Esemplificazione pattern `Proxy`

Proxy - Struttura



Struttura del pattern `Proxy`

Esempio

- Comunicazione tramite Socket
- Implementare la lettura della stringa nella classe Demo

Codice di riferimento

```
Socket (directory)
```


Considerazioni

- E' usato in
  - Java Remote Method Invocation (RMI)
  - Security Proxies