Design pattern – II

Stefano Mizzaro

Dipartimento di matematica e informatica
Università di Udine
http://www.dimi.uniud.it/~mizzaro
mizzaro@dimi.uniud.it

PAOO, Lezione 10
19/4/2004

Scaletta

- Riassunto
  - Inizio pattern di progetto
  - I 4 pattern visti: Adapter, Façade, Composite, Decorator
- Fine Decorator
- Discussione esercizio
- Altri pattern:
  - Bridge (complicato, ci mettiamo un po'…)
  - Proxy
  - Flyweight

Diagramma Class Adapter

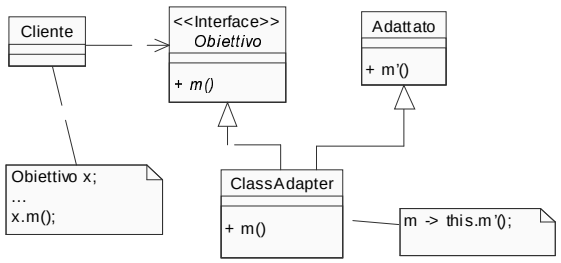
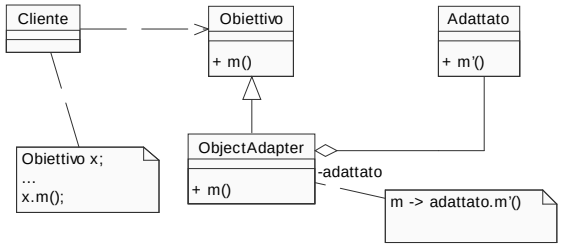


Diagramma Object Adapter



- Nota: Object Adapter non eredita…

Diagramma Façade

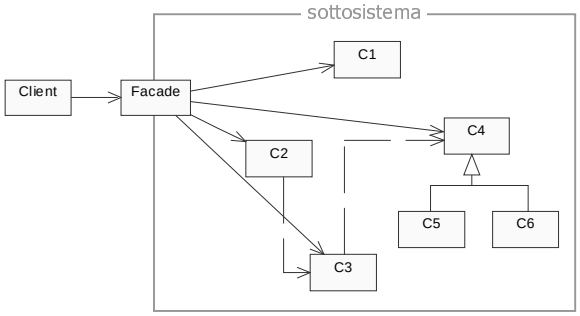


Diagramma Composite

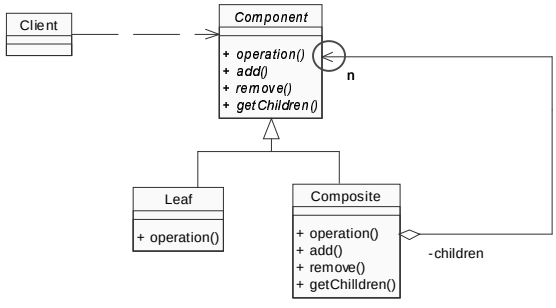
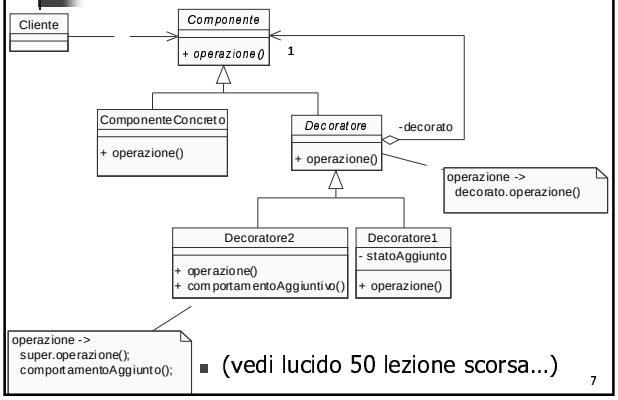


Diagramma Decorator



- (vedi lucido 50 lezione scorsa…)

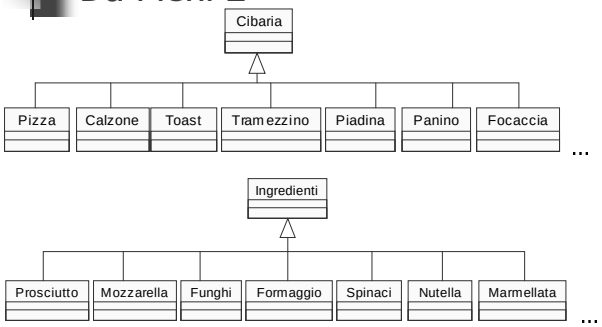
Esercizio

- Chiosco Da Meni
- Vende "cibarie"
  - Pizze, calzoni, toast, tramezzini, piadine, panini, focacce, …
- Con ingredienti a scelta fra:
  - Prosciutto, formaggio, mozzarella, funghi, spinaci, speck, gorgonzola, nutella, marmellata, …
- Eventualmente "doppi" o "tripli" o …
- Non modellate tutta la gestione, solo le classi che consentano la creazione di cibarie

Da Meni 1

- Tutte le combinazioni, una classe per ognuna
  - **`PizzaAlProsciutto`**, **`PiadinaNutella`**, **`PizzaMozzarellaERucola`**, **`PizzaNutellaEAsparagi`**, …
- Eventualmente usiamo l'ereditarietà
- Problemi:
  - Argh! Con N cibi e M ingredienti ci sono N*M combinazioni. Servono proprio tutte ste classi?
  - Eliminiamo **`PizzaNutellaEAsparagi`** e simili? E se arriva Toni che vuole proprio quella/e?

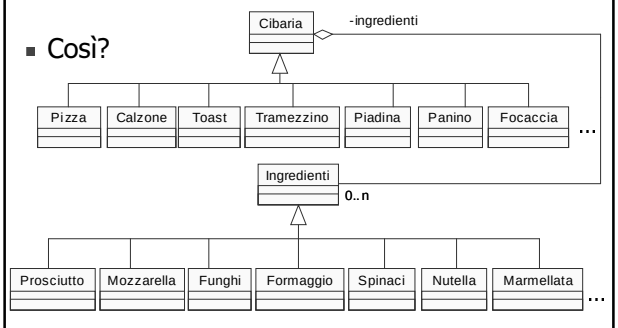
Da Meni 2



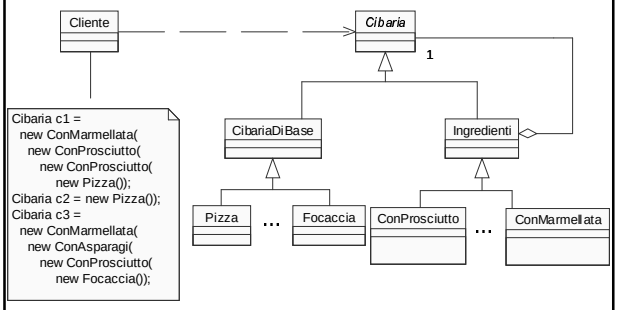
- Come li combiniamo?

Da Meni 3

- Così?



Da Meni 4



Da Meni 5

- È un Decoratore?
  - Che responsabilità/funzionalità aggiungo?
- È un Composite?
  - Però aggiungo un solo ingrediente alla volta, non raggruppo…
- Uhm, un po' Decoratore senza aggiunta di funzionalità, un po' Composite "degenere"…

5. Bridge (Ponte)

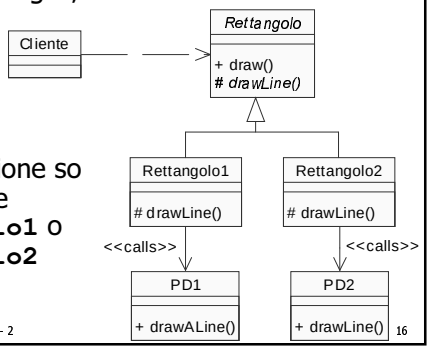
- Scopo
  - Separare un'astrazione dalla sua implementazione
  - per far sì che possano variare indipendentemente
- EH?!
- Bel pattern, ma difficile (si capisce subito)
- Spiegato male sul GoF…
- Ricaviamolo, lavorando su un esempio

Esempio

- Dobbiamo scrivere un programma che disegna rettangoli
- usando, in alternativa, uno di due programmi di disegno
  - **PD1**: **`drawALine(x1, y1, x2, y2)`**
  - **PD2**: **`drawLine(x1, x2, y1, y2)`**
- So quale dei due programmi usare al momento dell'istanziazione dei rettangoli
  - Il cliente non deve preoccuparsene

Prima soluzione (ingenua…)

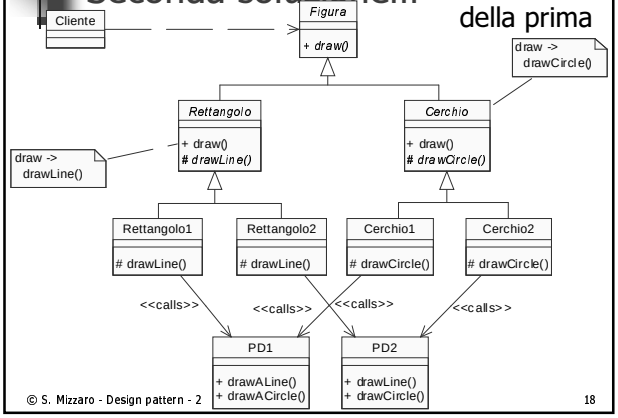
- 2 tipi di rettangoli, uno usa **`PD1`** e l'altro **`PD2`**
- All'istanziazione so se istanziare **`Rettangolo1`** o **`Rettangolo2`**



Mi ero dimenticato…

- … che voglio disegnare anche cerchi,
  - Per fortuna DP1 e DP2 hanno
    - **PD1**: **`drawACircle(x, y, r)`**
    - **PD2**: **`drawCircle(x, y, r)`**
- e sempre consentendo al cliente di astrarre
  - Cerchi come rettangoli → **`Figura`**

Seconda soluzione… … revisione della prima

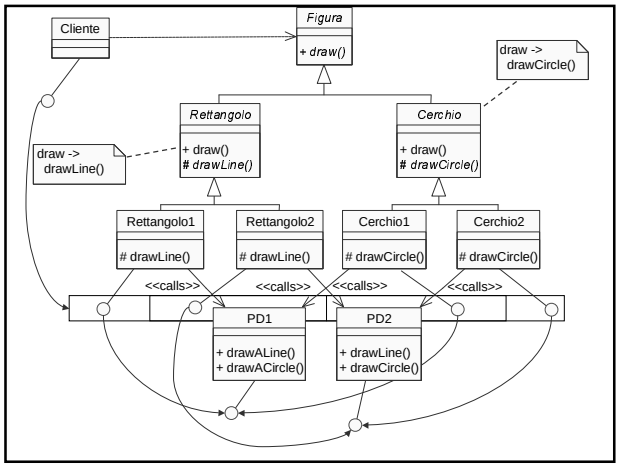


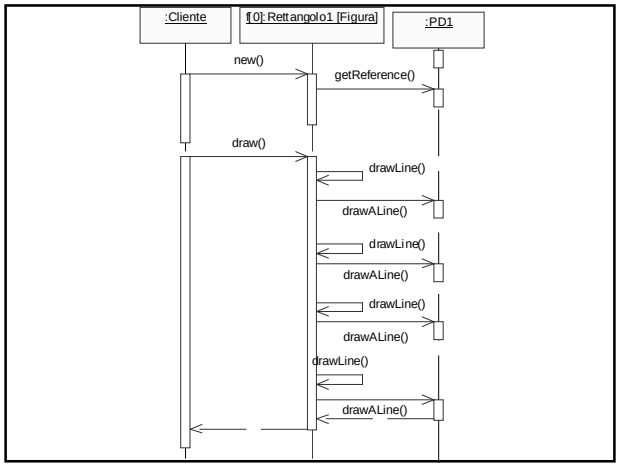
Come funziona

- Codice del **`Cliente`**:

```
Figura[] f = new Figura[4];
f[0] = new Rettangolo1(...);
f[1] = new Rettangolo2(...);
f[2] = new Cerchio1(...);
f[3] = new Cerchio2(...);
for (i = ...)
  f[i].draw();
```

- Vediamo le istanze
- Vediamo il diagramma di sequenza

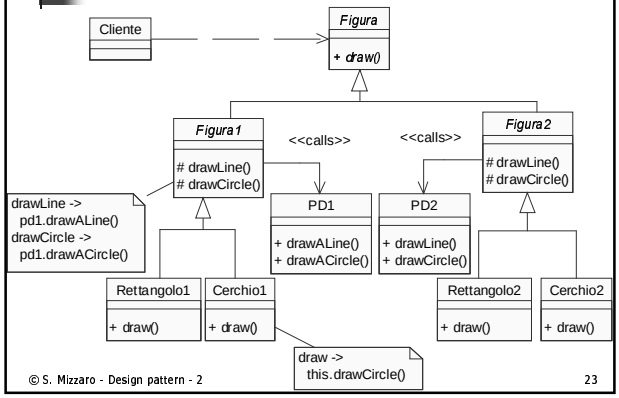




Commenti

- E se devo gestire altre figure?
  - Per ogni ulteriore figura aggiungo una piccola gerarchia (3 classi)
    - Classe astratta
    - Una sottoclasse per ogni PD
- E se per caso devo gestire/usare anche altri programmi di disegno (**`PD3`**, **`PD4`**, …)?
  - Per ogni ulteriore PD aggiungo una classe **`PDi`** e una classe in ogni gerarchia
- Sembra funzionare… ma non ci piace molto… Con N figure e M PD ho N*M classi a "livello 3"

Alternativa



Commenti

- Ho scambiato "l'ordine":
  - Da: Prima tipo di figura poi versione di PD
  - A: Prima versione poi tipo di figura
- Mah, non è cambiato molto
- Sempre N * M…

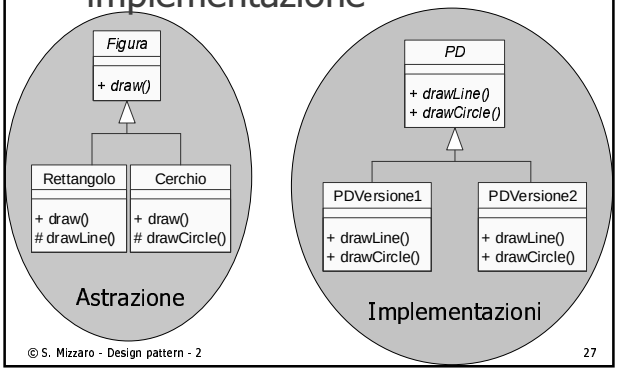
Uhm…

- Scopo del Bridge
  - Separare un'astrazione dalla sua implementazione
  - per far sì che possano variare indipendentemente
- Adesso capiamo! (?)
  - Astrazione: le figure
  - Implementazione: i programmi di disegno
- Con la nostra soluzione
  - Alto accoppiamento fra figure (astrazione) e PD (implementazioni) ⇒
  - Non riesco a farle variare indipendentemente
  - C'è anche ridondanza…

È di qualità?

- Insomma, non ci piace
- Separiamo astrazione e implementazioni
  - Astrazione: **`Figura`**
    - Ci sono 2 tipi di **`Figura`**
  - Implementazioni: Programmi di disegno (**`PD`**)
    - Anche qui ne abbiamo 2 versioni

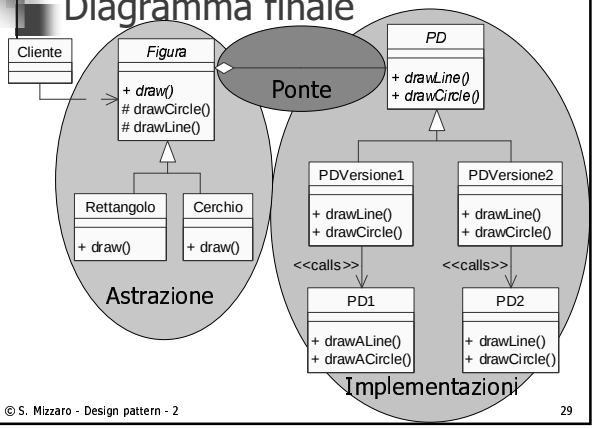
Separiamo astrazione e implementazione



Come li colleghiamo?

- Con un ponte ("bridge")…
- Una **`Figura`** delega la responsabilità del disegno (**`draw`**) a un (**`PD`**)
- Già che ci siamo, un dettaglio:
  - Rifattorizziamo **`#drawLine`** e **`#drawCircle`** nella sopraclasse per evitare duplicazioni…
  - Sono **`protected`** ⇒
    - Ereditati
    - Non visibili nell'interfaccia di **`Figura`**

Diagramma finale



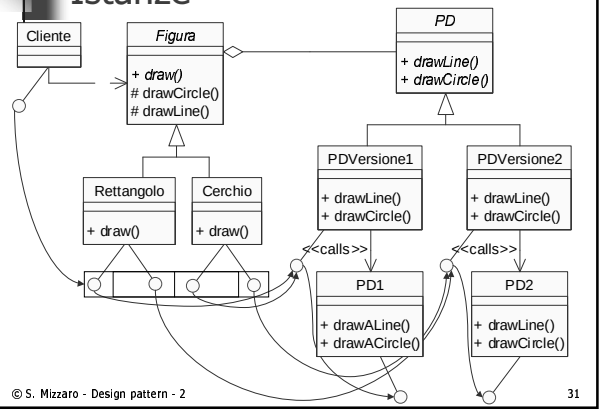
Come funziona

- Codice del **`Cliente`**:

```
Figura[] f = new Figura[4];
PD p1 = new PDVersione1();
PD p2 = new PDVersione2();
f[0] = new Rettangolo(p1);
f[1] = new Rettangolo(p2);
f[2] = new Cerchio(p1);
f[3] = new Cerchio(p2);
for (i = ...)
  f[i].draw();
```

- Vediamo le istanze
- (Diagramma di sequenza: Ex.)

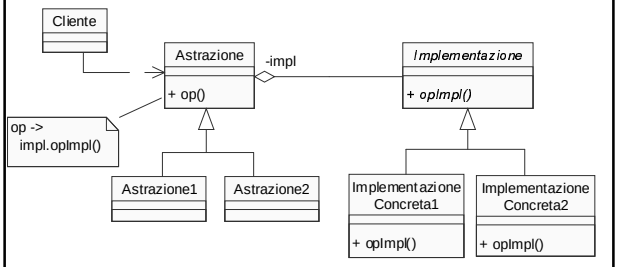
Istanze



Commenti

- Abbiamo
  - eliminato un po' di ereditarietà
  - aggiunto composizione/delega
- Non c'è più esplosione combinatoria
  - N + M classi, non N * M
- Come ci siamo arrivati?
  - Scopo del Bridge (finalmente chiaro…)
    - Separare un'astrazione dalla sua implementazione
    - per far sì che possano variare indipendentemente

Diagramma del Bridge



Commenti

- Spesso interfaccia e implementazione
  - sono nella stessa classe,
  - o vengono separate in sopraclasse e sottoclasse
- Bridge le separa "di più"
- Evita anche ricompilazioni
- Morale più generale:
  - Mai accontentarsi della prima versione…
  - … però evitare la "paralysis by analysis"…

Il Bridge secondo Eckel

- Bridge è uno strumento per il programmatore:
  - gli permette di organizzarsi il codice
  - per poter aggiungere nuove astrazioni (front-end) e nuove implementazioni (back-end) in modo semplice
- Si potrebbe usare delegazione invece di ereditarietà nella variazione di astrazioni
- Pro
  - Astrazioni con interfacce diverse
- Contro
  - Cliente non può usare il polimorfismo

Bridge vs. Adapter

- Obiettivi diversi
- Adapter adatta, Bridge separa interfaccia e implementazione per farle variare indipendentemente
- Bridge si usa tipicamente all'inizio di un progetto,
- Adapter viene usato dopo che le classi coinvolte sono state progettate/implementate e vengono riusate
- Comunque le classi **`PDVersione1`** e **`PDVersione2`** sono due adattatori oggetto…
  - Soluzione frequente

Bridge vs. Decoratore

- Obiettivo in comune: evitare la proliferazione (esplosione combinatoria) di classi
- Struttura diversa

6. Proxy

- Scopo
  - Fornire un surrogato o un segnaposto per un oggetto
  - Per controllarne l'accesso
- Esempi
  - Per efficienza: In un word processor, caricare le figure solo al bisogno evitando di caricarle tutte all'inizio
  - Per sicurezza: Segnaposto che controlla e gestisce i privilegi d'accesso
  - ...

Segnaposto in word processor

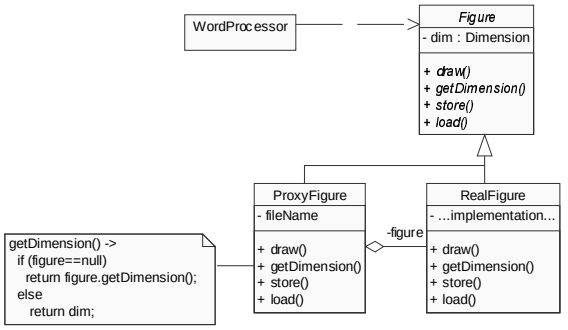
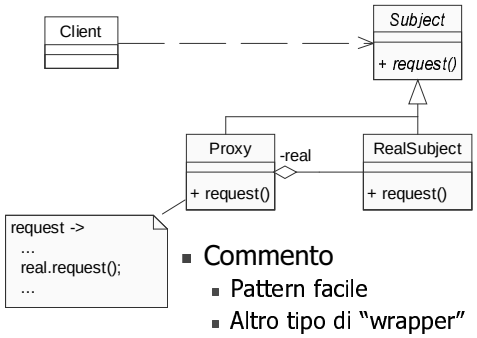


Diagramma del Proxy



- Commento
  - Pattern facile
  - Altro tipo di "wrapper"

4 tipi di Proxy

- Proxy remoto
  - Rappresentante locale per oggetto in spazio ≠ (Es.: RMI, Remote Method Invocation)
- Proxy virtuale
  - Gestisce la creazione su richiesta di oggetti "costosi" (Es.: immagine in word processor)
- Proxy di protezione
  - Gestisce diritti di accesso ≠ per oggetti ≠
- Riferimento intelligente
  - Sostituisce un puntatore (Es.: #riferimenti per gestire la distruzione)

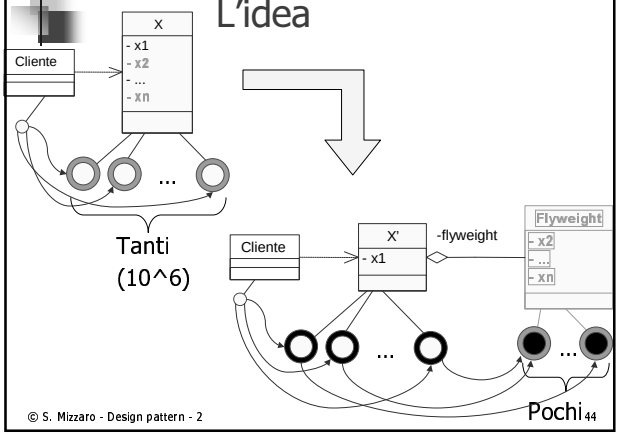
Copy-on-write

- Uso tipico del Proxy
- Copia di un oggetto grande/complicato
  - Costosa
  - E magari inutile, se non modifico...
- Soluzione:
  - Non copio, metto un Proxy che controlla l'accesso
  - E copia solo quando l'oggetto viene modificato

7. Flyweight (Peso piuma)

- Scopo
  - Usare la condivisione di oggetti/istanze
  - per rappresentare in modo efficiente un gran numero di oggetti
  - [a granularità fine (?)]
- Migliorare le prestazioni (usare meno memoria)
- "Performance hack" [Eckel]
- Lo vediamo un po' "di corsa"

L'idea

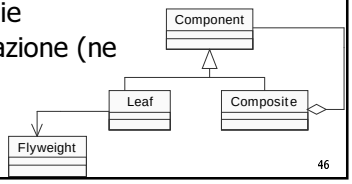


Stati "interno" ed "esterno"

- "Interno" (al flyweight)
  - Indipendenti dal contesto, condiviso
- "Esterno"
  - Dipendenti dal contesto, individuale
- Esempio
  - Caratteri in un word processor
    - Condiviso: rappresentazione font true type, funzioni per scalare, …
    - Individuale: dimensione, stile, posizione

Commenti

- Più utile se tanti oggetti con duplicazione dati
- Attenzione ai confronti con "**==**"!!
- Flyweight vs. Decorator e Bridge
  - Decorator&Bridge: esplosione combinatoria classi
  - Flyweight: esplosione combinatoria oggetti
- Spesso usato insieme al Composite per condividere le foglie
- Factory per la creazione (ne parleremo)



Riassunto

1. Adapter (Adattatore)
2. Façade (Facciata)
3. Composite (Composto)
4. Decorator (Decoratore)
5. Bridge (Ponte)
6. Proxy (Proxy)
7. Flyweight (Peso piuma)

Esercizio

- Programma di grafica
- "Palette" figure di base
- "Palette" figure create dall'utente raggruppando
- Gestione gruppi