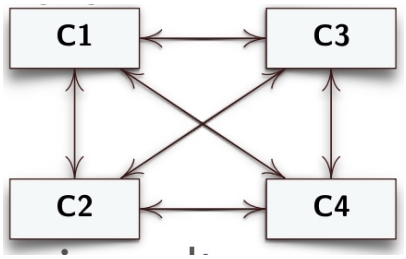
Design Pattern Mediator

Intento

Viene usato per gestire le comunicazioni tra oggetti in modo CENTRALIZZATO. Si riduce la dipendenza degli oggetti attraverso un oggetto detto MEDIATORE che si mette al centro delle comunicazioni riceve le comunicazioni e le invia all'oggetto destinatario corretto

- Gli oggetti vogliono interagire fra loro ma interagiranno tramite un oggetto che rappresenta le interazioni fra oggetti.
  Gli oggetti sono legati ma non si conoscono fra loro.
- Si promuove lo scoppiamento degli oggetti che hanno bisogno di interagire fra loro.



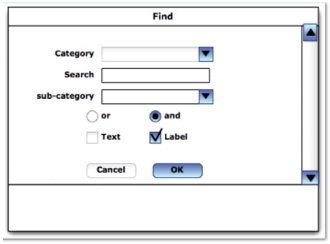
- Un oggetto `C1` comunica un risultato a `C2` e tutti gli altri oggetti. Le ***interazioni avvengono fra oggetti***.
- Ogni classe è costretta a conoscere le altre classi
- Se si deve **modificare** C3, per esempio, ***per via delle dipendenze,*** devo **modificare un po'** di `C1`, `C2` e `C4` di ***conseguenza***

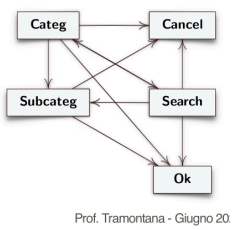
> *Quindi la modifica di una classe comporta la modifica di tutte le altre classi che dipendono da essa o da quale dipende tale classe*

- Il **ri-uso** di queste classi **diventa difficile** perchè esse sono legate fra loro e quindi vi è una ***LIMITAZIONE*** e si tratta di un **SISTEMA MONOLITICO**
- Si devono **eliminare** le interazioni/dipendenze fra le classi
- Il comportamento complessivo (di interazione) fra le classi, lo esprimo tramite un oggetto a sé stante, cioè tramite il ***MEDIATOR***
- Gli oggetti risultano più isolati e ***non più** con forti dipendenze *e quindi* più facili da usare*

Esempio che mostra la motivazione

> *Una finestra di dialogo ha dei* bottoni, caselle *per inserire del testo e altre parti..*





- Ci sono tante classi che vengono istanziate e che rappresentano ogni parte della finestra
- Le istanze sono necessarie perchè, quando l'utente interagisce, il comportamento delle parti di finestra dipendono, appunto, dall'interazione stessa
- Se l'utente sceglie *Category*, la *sub-category* deve essere **riempita opportunamente** in base alla categoria scelta dall'utente
- *Un altro esempio di dipendenza* è: il tasto OK è abilitato solo se le altre parti di finestra sono "*state riempite*"
- Ogni classe *chiama i metodi* di **TUTTE** le altre classi, quindi vi è una ***forte dipendenza fra loro***

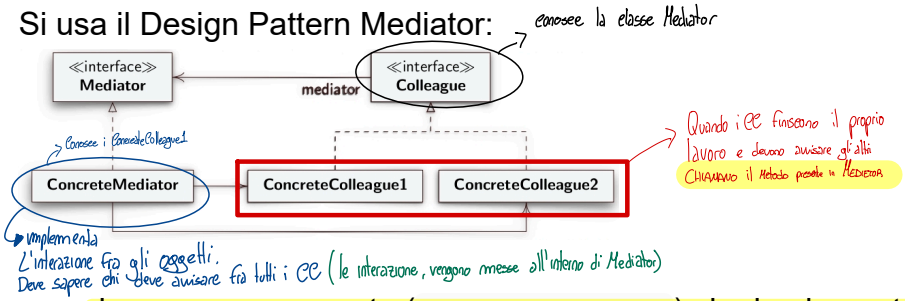
Ogni classe implementata deve ***comunicare dati a tutte le altre classi*** che servono per gestire bene la finestra e aggiornare la visualizzazione

> *In questo caso vi è un sistema monolitico: se si vuole un'altra finestra, riusare il codice risulta complicato perchè, magari, nella nuova finestra non vi sono più delle parti che nella vecchia c"erano*

Soluzione

Le classi comunicano tra di loro attraverso Mediator

Si usa il Design Pattern Mediator:



- si usa un componente (`ConcreteMediator`) che implementa le interazioni fra i vari oggetti
- Le **classi *NON* comunicano più fra loro** ma lo fanno tramite il Mediator
- ***Colleague*** (`C`) conosce l'interfaccia ***Mediator*** (`M`)
- Il ***ConcreteMediator*** (`CM`) conosce i ***ConcreteColleague*** (`CC`)
- Quando i `CC` finiscono il loro lavoro e devono avvisare gli altri, chiamano un metodo definito in `M`
- `CM` deve sapere qual è l'avviso dato dai `CC` e sapere chi deve avvisare fra tutti i `CC`
- Le interazioni, quindi, vengono messe all'interno di `M`

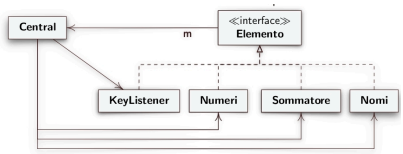
Conseguenze

- Tutte le dipendenze del gruppo di oggetti che si scambiano i vari risultati, sono gestite dal `CM`
- Si è tolto parte di codice che serviva alle interazioni fra singoli oggetti e queste parti di codice vengono messe all'interno di `CM`
- I `CC` sono più **RIUSABILI** che devono solo conoscere l'interfaccia `M`

- Ogni classe `CC` non conosce l'esistenza di altre `CC` (`CC1` *non conosce* `CC2`) quindi ogni singolo `CC` si può riusare
- Solitamente i `CM` non sono RIUSABILI perchè rappresenta le singole interazioni in *QUELLA DETERMINATA APPLICAZIONE*
- I `CC` sono meno specifici, appunto perchè non conoscono i codici degli altri `CC`

Esempio applicazione di Mediator

Si ha un'interazione con l'utente (*senza interfaccia grafica*, cioè diversamente dall'esempio di prima). Ogni cosa che l'utente ci dice, può far scatenare eventi a catena.



- KeyListener `KL` legge i dati da tastiera. Entra in azione solo quando si deve leggere dallo standard input
- Se è un dato numerico, esso serve a Numeri `Nu` e a Sommatore `S`
- Il *ConcreteMediaor* `CM` è *Central* `C`
- Se è un dato testuale, esso serve a *Nomi* `No`
- `Nu` valuta il numero fornito e decide se chiedere un altro numero in input da `KL` (tramite `C`)
- `No` valuta la stringa decide cosa fare
- `S` può tenere da parte i numeri dati e può dare un risultato dato dalla somma dei numeri forniti in input

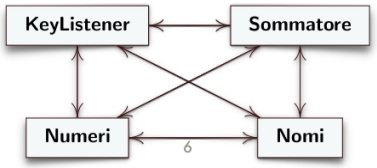
Generalmente...

- Si ha bisogno da varie classi e ognuna di esse fa qualcosa di diverso
- Le interazioni fra le classi sono gestite da *Central* `C`
- Elemento `E` rappresenta il *Colleague*
- `C` fa da *ConcreteMediator* ed è un'***ALTERNATIVA AL MEDIATOR*** *senza interfaccia*
- Questo design pattern **non suggerisce un nome dei metodi perchè** è ***molto dipendente*** da come esso si vuole usare

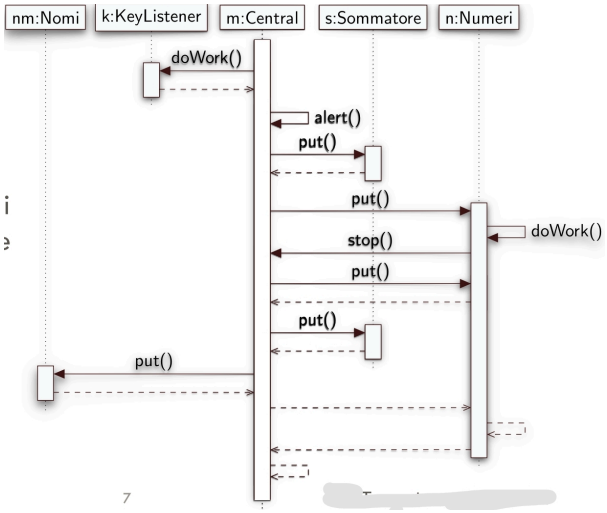
Lo **scopo dell'applicazione** è:

- L'utente deve rispondere a 2 domande: se risponde alla prima deve fornire un numero. Se risponde alla seconda deve fornire una stringa
- Il **Mediator** si occupa ***ANCHE dell'istanziazione*** degli oggetti delle varie classi
- Ogni `Colleague` hanno un costruttore che prende in ingresso un parametro di riferimento a `Central`

Se non si usasse il design pattern mediator, si avrebbe la seguente situazione:



- Il Mediator Central avvia la lettura da tastiera tramite il metodo doWork() di KeyListener e ottiene da esso il valore letto, quindi Central chiama put() sugli oggetti interessati al valore letto
- Quando un oggetto ConcreteColleague riconosce una condizione di arresto, chiama stop() su Central, che avvisa gli altri ConcreteColleague
- In figura si mostra il caso in cui Numerichiama stop() su Central
  Il *diagramma di sequenza* è il seguente:



Esempio di codice

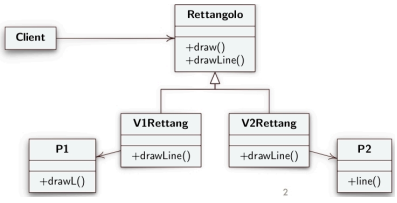
Design Pattern Bridge → STRUTTURALE

Intento

Consente di separe l'astrazione dall'implementazione in modo che entrambe possano essere modificate INDIPENDENTEMENTE l'una dall'altra

- Un'astrazione può avere diverse implementazioni e in questo caso si usa l'ereditarietà
- Si forniscono algoritmi diversi per quell'unica astrazione che si è pensato
- Si deve disaccoppiare un'astrazione dalla sua implementazione così che le due possano variare indipendentemente

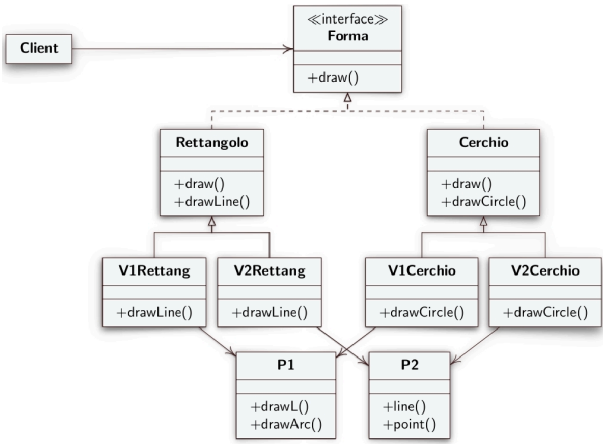
Esempio



- Si ha bisogno dell'astrazione *Rettangolo* e si vuole disegnare

- Quando si deve disegnare si può avere bisogno di scegliere fra più istruzioni diverse per il disegno
- Quando si cambia *libreria di disegno*, si ha bisogno di un pezzo di codice che chiami i metodi giusti della libreria selezionata
- Una libreria (detta anche *piattaforma P1*) mette a disposizione `drawL()` per disegnare una linea
- Un'altra libreria (*P2*) fornisce `line()` e si deve invocare su un'altra classe diversa.
- Si distingue il codice e fra le 2 librerie non si mischiano fra loro
- Le due classi `V1Rettang` e `V2Rettang` sono particolare implementazioni dell'astrazione `Rettangolo`

Ma se, all'interno dell'applicazione, si vuole anche un Cerchio, si avrà:



In questo caso, il codice è modulare in 2 classi diverse, sia in Rettangolo che in Cerchio
Se si volesse introdurre un'altra forma geometrica (Triangolo) allora:

- Si deve implementare Triangolo con le 2 possibili classi (o più).
- Per ogni asrtazione in più di cui ho bisogno, devo implementare 2 classi in più e così via...
- Se non basta interfacciarsi con 2 librerie e serve una terza libreria `P3`, allora, per ogni forma pensata, serve un'altra classe che implementa i metodi a questa nuova libreria (`V3Rettang`, `V3Cerchio`, `V3Triangolo` che implementano i metodi di `P3`)
- Si ha una ***POLIFERAZIONE DI CLASSI***

Quindi....

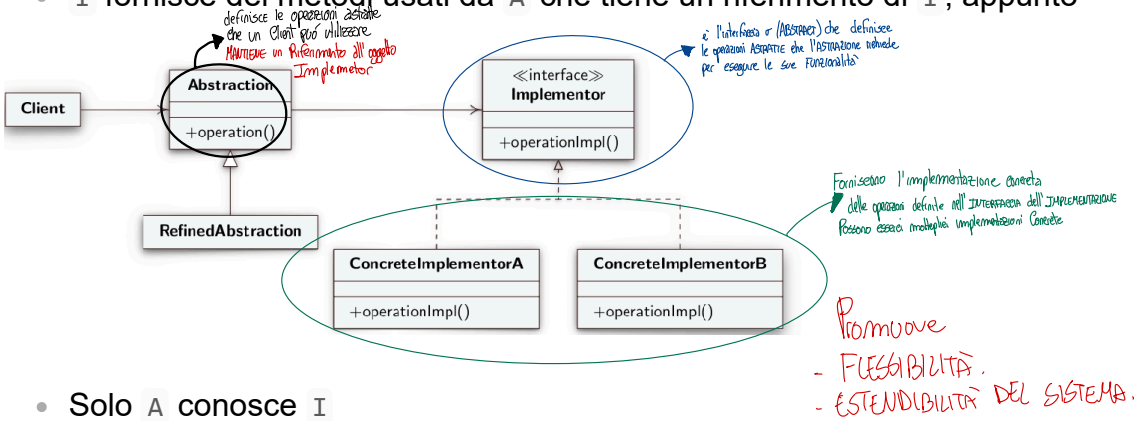
- Per ogni variazione da introdurre, ***si vorebbe un incremento lineare del numero di classi.*** (Aggiunte di nuove astrazioni o di nuove piattaforme (librerie) da usare)
- Ogni classe è legata ad una certa piattafoma in modo permanente, cioè un'istanza di V1Rettang non può usare una piattaforma diversa da `P1`

Soluzione

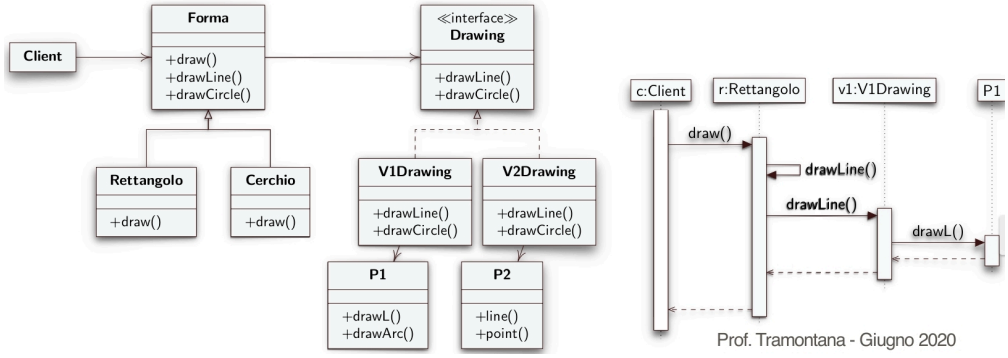
il pattern Bridge affronta questo problema introducendo due GERARCHIE SEPARATE: una gerarchia di astrazione e una di implementazione

Si usa il Design Pattern Bridge:

- Si crea un'**Abstraction** `A` generale che tiene in riferimento un oggetto **Implementor** `I` e, userà questo riferimento per essere implementata. Qui vengono inseriti i metodi che serviranno essere usati
- Si usano delle sottoclassi di `A`, chiamate **RefinedAbstraction** `RA` che usano i metodi implementati in `A`.
- Si hanno degli `I` e dei **ConcreteImplementor** `CI` e questi ultimi forniscono le operazioni concrete
- `I` è un'interfaccia che rappresenta i vari `CI`
- i `CI` è uno per ciascuna piattaforma da pilotare. Ogni `CI` sa chiamare i metodi giusti da usare a run-time
- `I` fornisce dei metodi usati da `A` che tiene un riferimento di `I`, appunto



- Solo `A` conosce `I`
- `CI` si *servono delle piattaforme* che si vogliono supportare e ognuno di loro **conoscono una sola piattaforma**
- `A` è una classe che definisce e implementa `operation()` e fa da superclasse per `RA`
- `RA` usa operation di `A`
- Il client si lega all'A ma si può legare anche a `RA`



- Le RA chiamano metodi di A
- Per implementare i draw() nelle RA mi servo dei metodi implementati nella classe Forma
- Se si deve supportare una piattaforma aggiuntiva, si deve creare solo una nuova classe V3Drawing che chiamerà metodi della piattaforma P3 e dovrà fornire i metodi utili all'Abstraction
  - Si deve solo fornire all'abstraction il riferimento alla nuova classe `V3Drawing`
- ***L'aggiunta di una piattaforma*** equivale ***all'aggiunta di una sola classe*** di *Drawing*

> *Se ci serve la classe Triangolo che avrà al suo interno `draw()` che chiamerà 3 volte `drawLine()` per le 3 linee da disegnare*

Conseguenze

- Bridge permette a una implementazione di non essere connessa permanentemente a una interfaccia, l'implementazione può essere configurata e anche cambiata a runtime
- Il **disaccoppiamento** permette di cambiare l'implementazione **senza dover ricompilare** *Abstraction* ed i *Client*
- Solo certi strati del software devono conoscere *Abstraction* e *Implementor*
- I *Client* non devono conoscere
- Le gerarchie di *Abstraction* e *Implementor* possono ***evolvere in modo indipendente***

Esempio minimale di Bridge

```
// Forma è una Abstraction
public class Forma {
  private Drawing impl;
  public void setImplementor(Drawing imp) {   this.impl = imp;   }
  public void drawLine(int x, int y, int z, int t) {
    impl.drawLine(x, y, z, t);
  }
}
```

```
// Drawing è un Implementor
public interface Drawing {
   public void drawLine(int x1, int y1, int x2, int y2);
}
```

```
// Rettangolo è una RefinedAbstraction
public class Rettangolo extends Forma {
  private int a, b, c, d;
  public Rettangolo(int xi, int yi, int xf, int yf) {
    a = xi; b = yi; c = xf; d = yf;
  }
  public void draw() {
    drawLine(a, b, c, b); drawLine(a, b, a, d);
    drawLine(c, b, c, d); drawLine(a, d, c, d);
  }
}
```


Prof.

Versione più completa