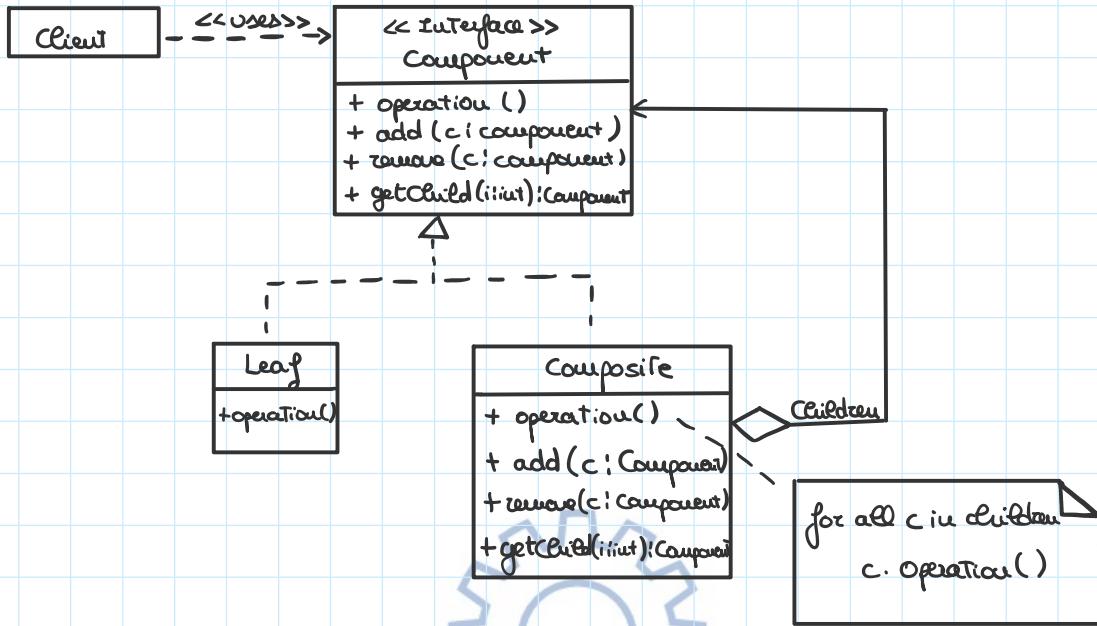


- **Composite** (object-based) \* composite / composite / leaf

Usato per rappresentare gli oggetti come composizione di altri in maniera gerarchica e per potervi applicare tutti gli stessi metodi.



es. Una interfaccia di grafica permette di accedere sia a componenti elementari come i punti che ad oggetti composti come un poligono e di poterli gestire alla stessa maniera.

## APPUNTI DI INGEGNERIA

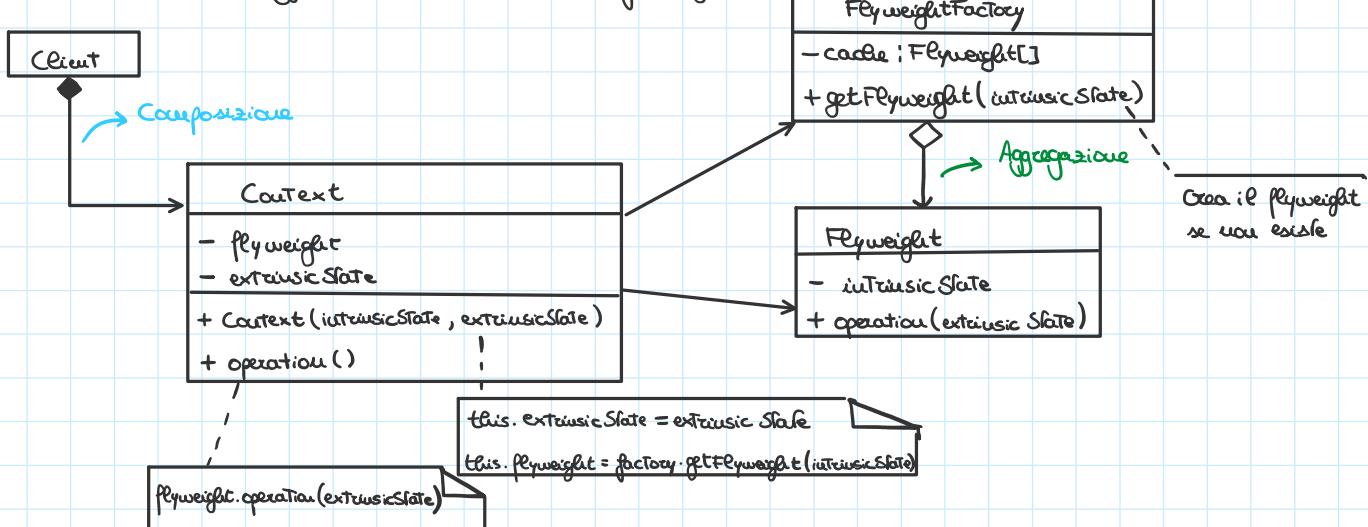
**CONTRO:** I metodi di gestione dei figli sono utili solo per i composite. Metterli solo nei composite elimina l'uniformità mentre introdurre delle eccezioni nelle leaves va contro il principio di Liskov in quanto sono eredi di component ma non possono essere usate al suo posto.

- **Facade** (object-based)

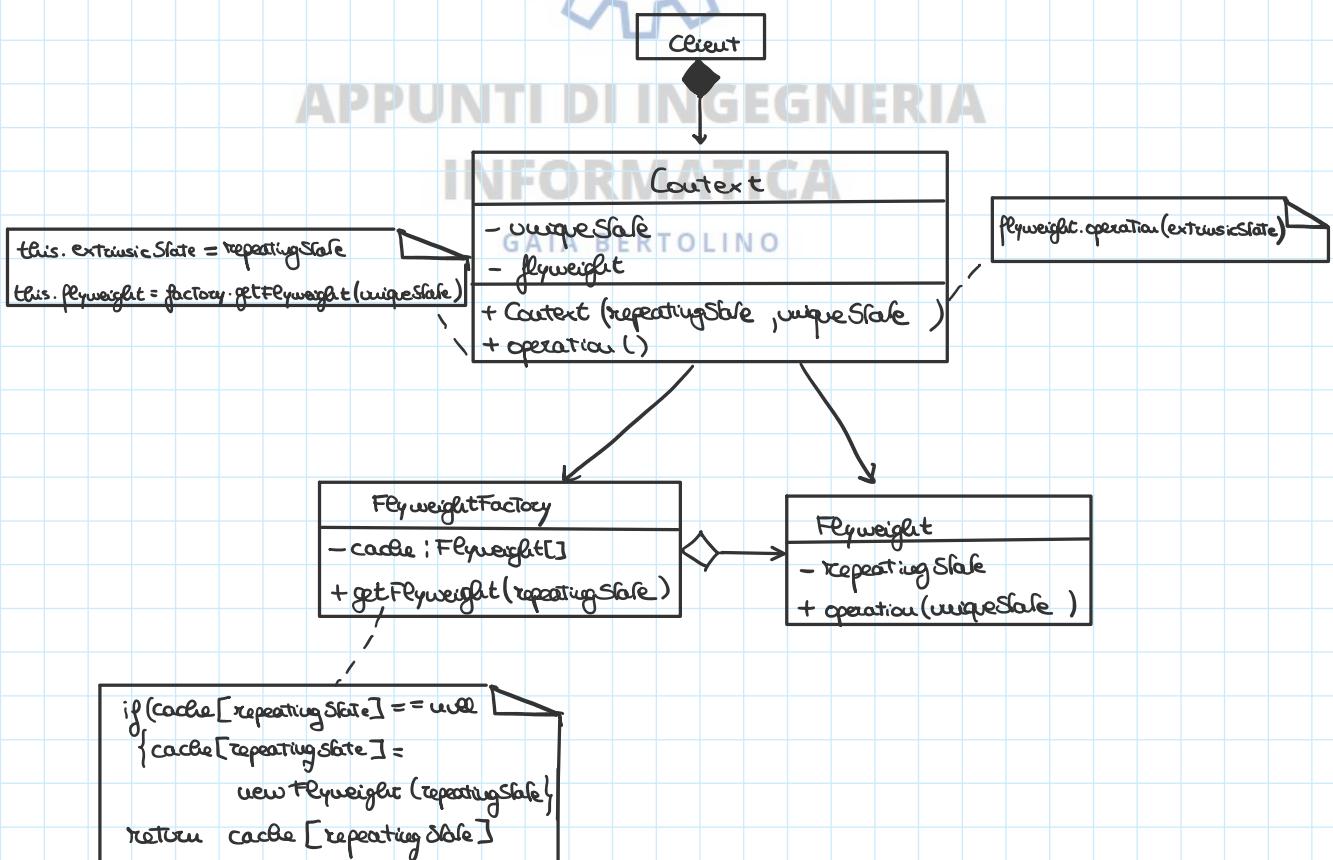
Usata per avere una interfaccia unificata che è l'unica ad interagire con il client

### • Flyweight (object-based) \* Context / flyweightFactory

Usato per una più efficiente memorizzazione degli oggetti che presentano uno stesso stato interno attraverso un oggetto condiviso chiamato flyweight

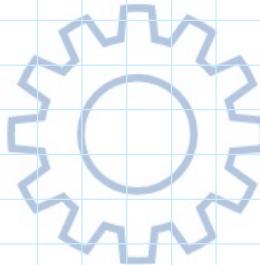
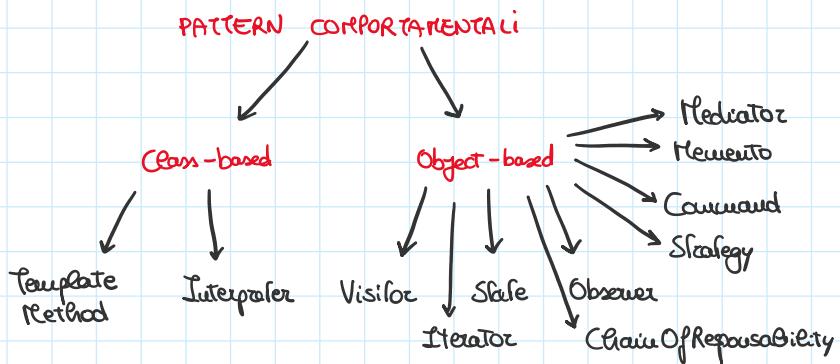


es. In un videogioco, i proiettili sparati in una partita hanno tutti le stesse caratteristiche di base (ovvero lo stato interno) che sono condivisibili: una diversa velocità (ovvero stato esterno) che non sono condivisibili fra oggetti.



**CONTRO:** se l'individuazione dello stato esterno è difficoltosa e la sua riunione crea vantaggi nel costo di memorizzazione applicare il pattern è inutile

## PATTERN CORPORATENALE



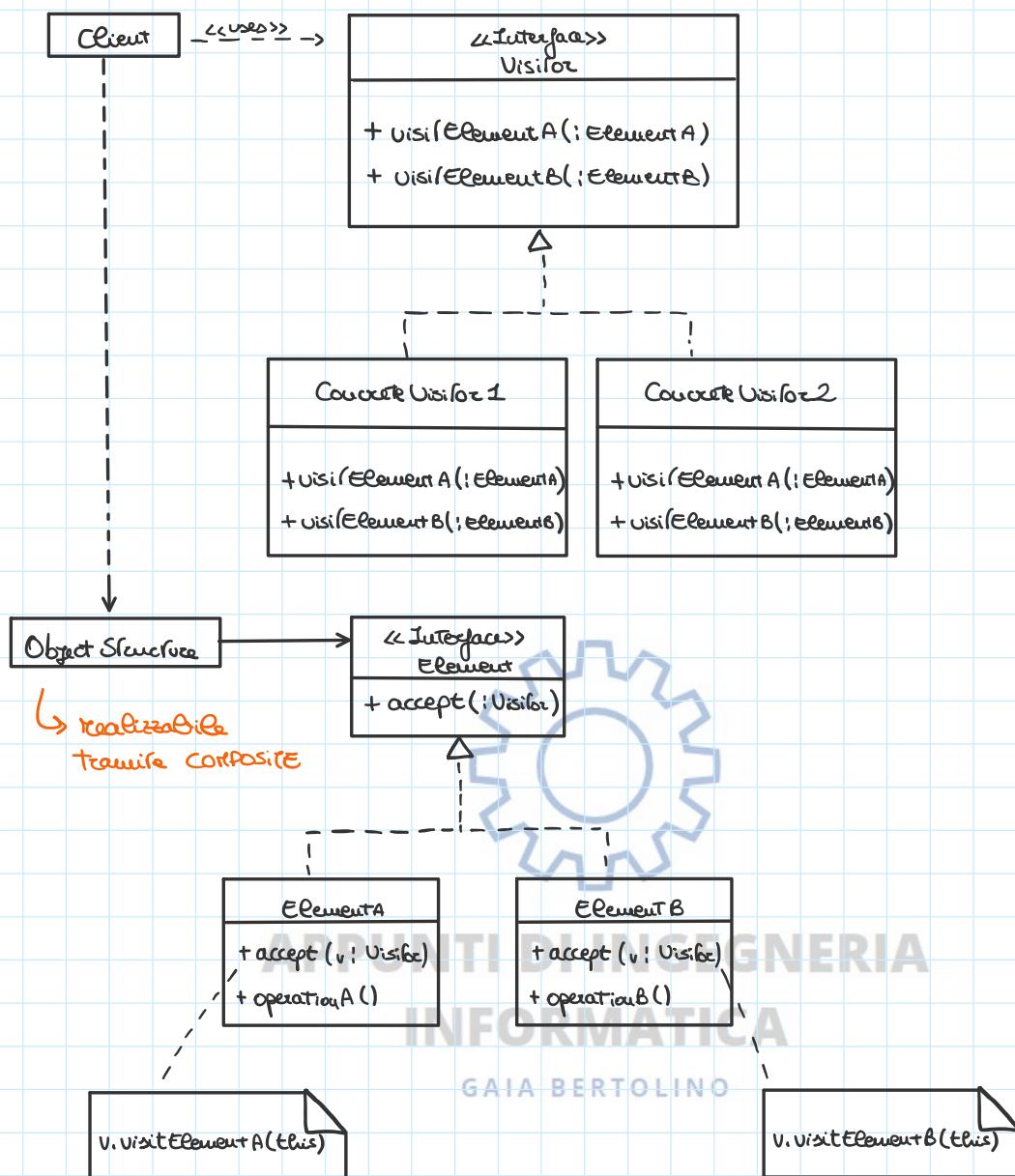
# APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

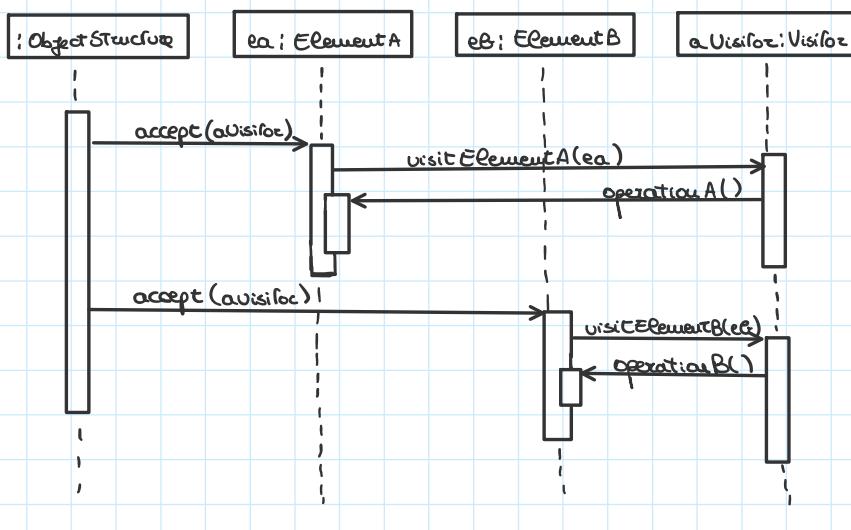
- **Visitor** (object-based) \* Element / Visitor

Utilizzato per uniformare l'accesso ad oggetti esterni e per introdurre comportamenti comuni senza dover alterare le classi

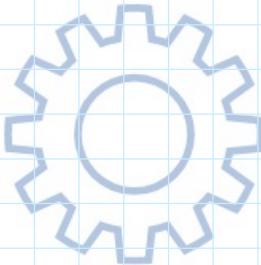
## Class diagram



## Sequence diagram



- PRO:** **Inoltro doppio** (es. `accept(:aVisitor)`)  
il pattern permette di aggiungere delle operazioni ad un gruppo di oggetti quando questi vengono visitati senza dover alterare le sue classi. Inoltre, i tipi di elementi appartenenti ad un visitor possono essere diversi (e non uguali come in Iterator).
- CONTRO:** L'aggiunta di un elemento concreto richiede l'aggiunta del metodo di visita associato all'interno di visitor e delle sue classi concrete.  
Inoltre, per operare le sue attività richiede che le informazioni degli elementi siano pubbliche.  
L'attraversamento degli oggetti può essere implementata con un Iterator o attraverso la classe stessa.

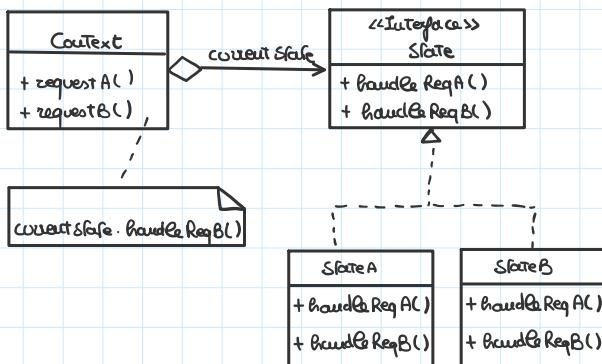


# APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

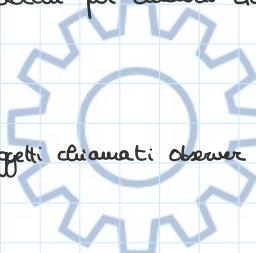
### • State (object-based) \*Context / State

Usato per far assumere ad un oggetto un comportamento diverso in base al suo stato



**PRO:** Le transazioni fra stati sono atomiche. Col pattern Flyweight si possono definire le parti di stato che si ripetono

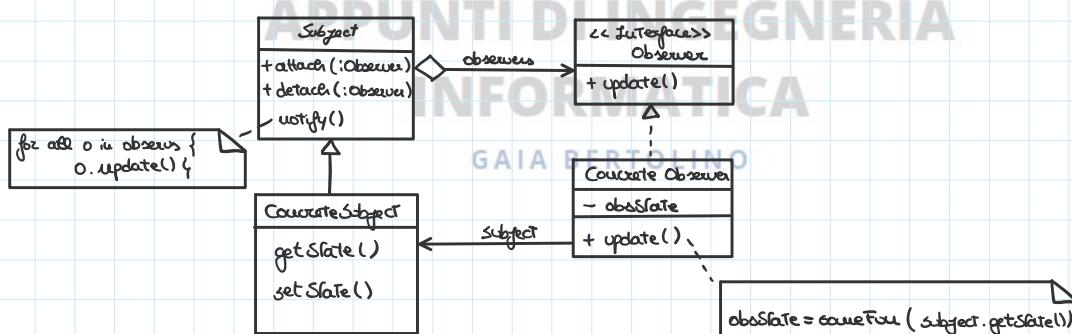
**CONTRO:** Crea la necessità di avere molti blocchi per ciascuno stato.



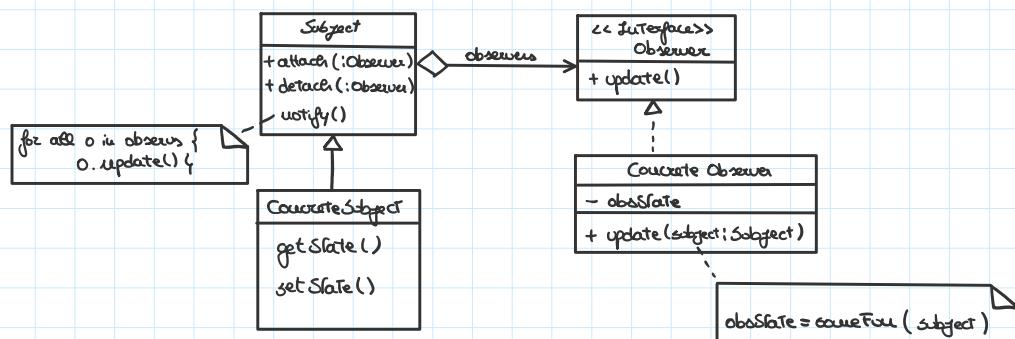
### • Observer (object-based) \*Subject / Observer

Usato per creare una dipendenza fra molti oggetti chiamati observer e uno chiamato subject

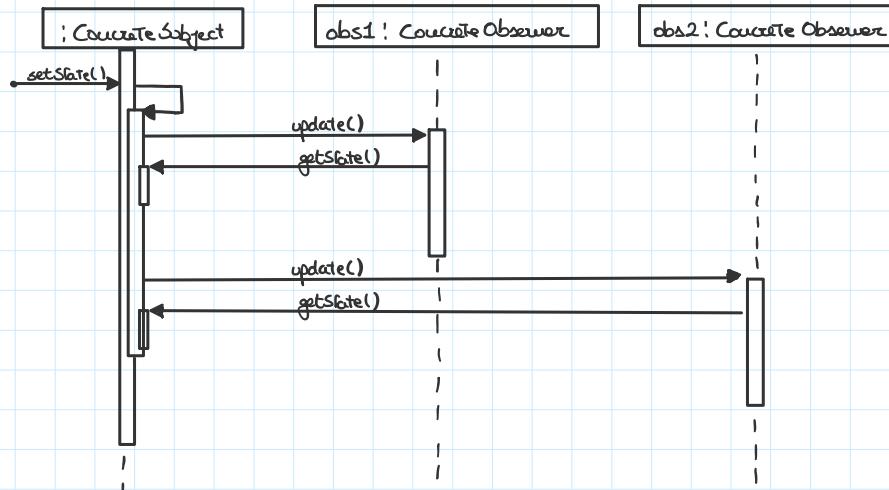
Modello ad un solo object



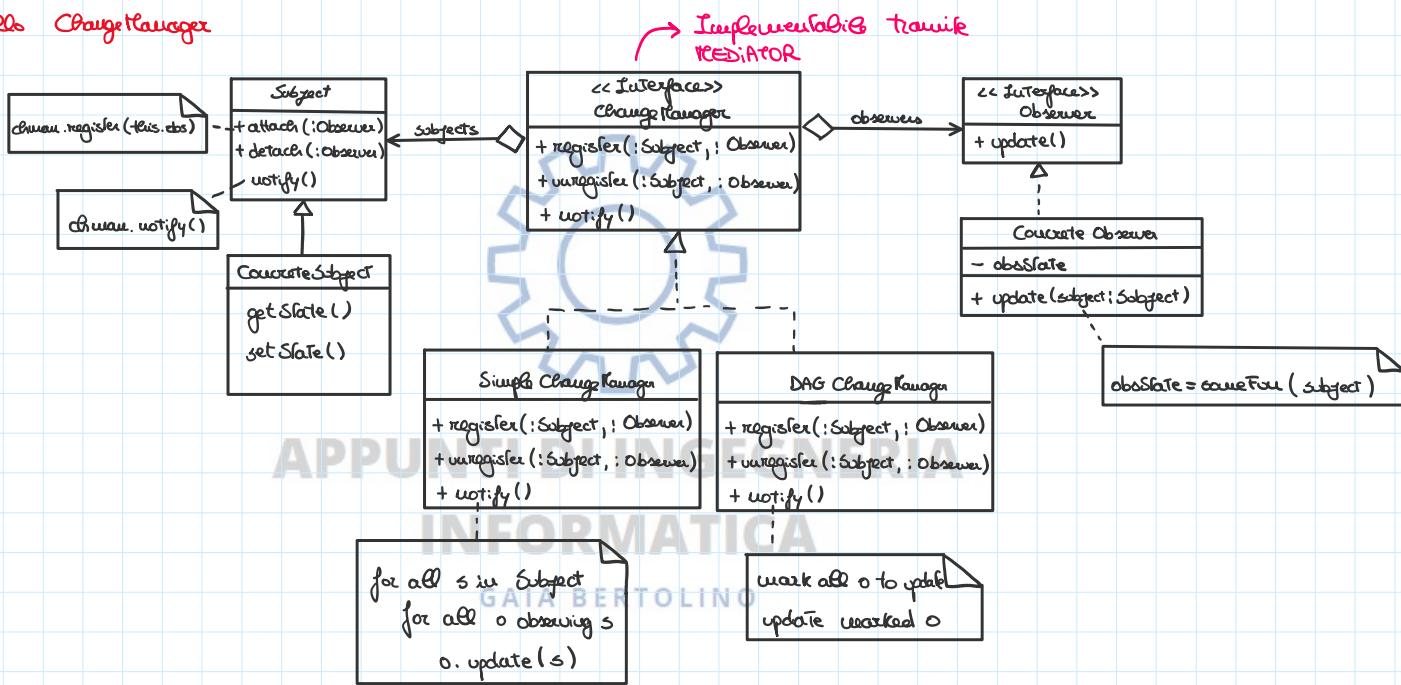
Modello a più subject



## Sequence diagram



## Modello ChangeManager



**Tipologie:**

- **Pull model**: dopo la notifica, l'observer richiede al subject le informazioni da modificare
- **Push model**: il subject invia le informazioni da modificare con la notifica

es. Esistono diversi tipi di grafici e la modifica dei dati li influenzano tutti

**PRO:** È possibile introdurre più observers senza riferire al subject le informazioni da modificare

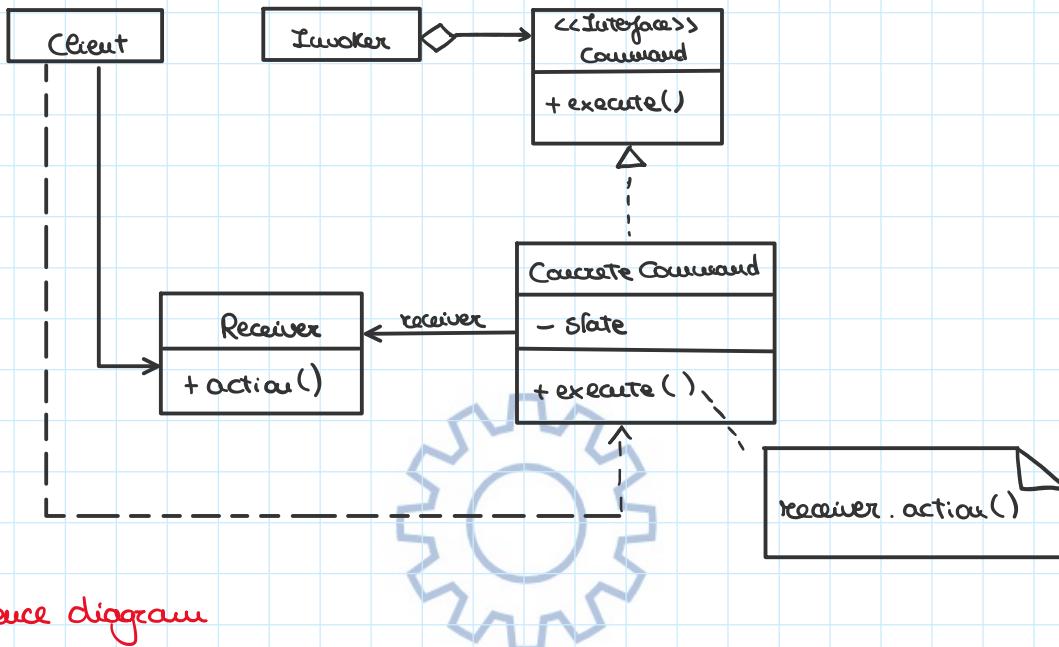
La comunicazione avviene in broadcast quindi è compito dell'observer decidere se recepirla o meno

**CONTRO:** I cambiamenti notificati non tengono conto in alcun modo del costo di implementazione

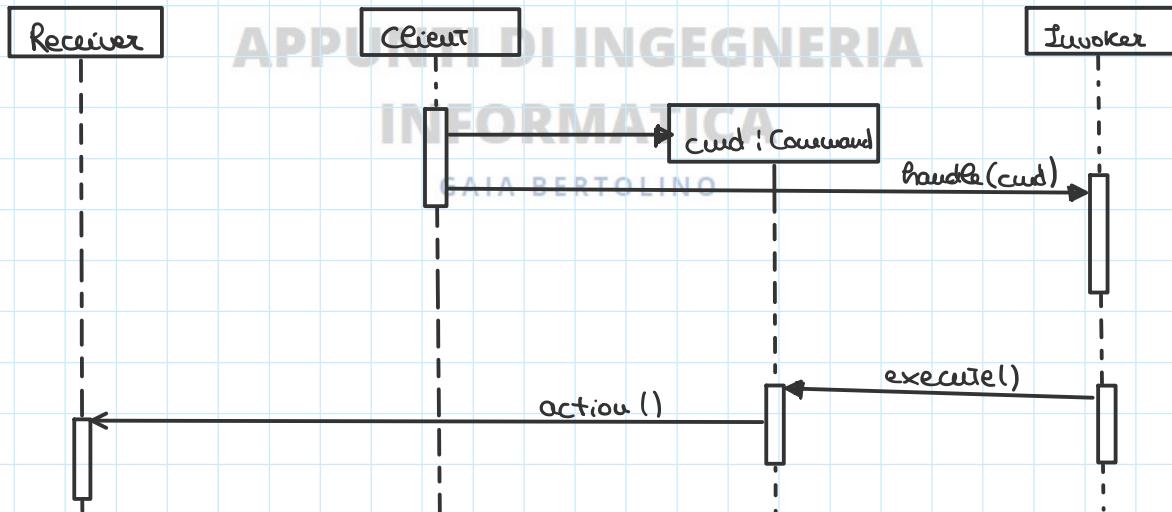
- **Command (object-based)**

Usato per separare un comando e la sua esecuzione.

### Class diagram



### Sequence diagram



**PRO:** Vi è un disaccoppiamento fra un comando e la sua implementazione.

Inoltre, un comando è implementato come oggetto e ha dunque la possibilità di essere eseguito.

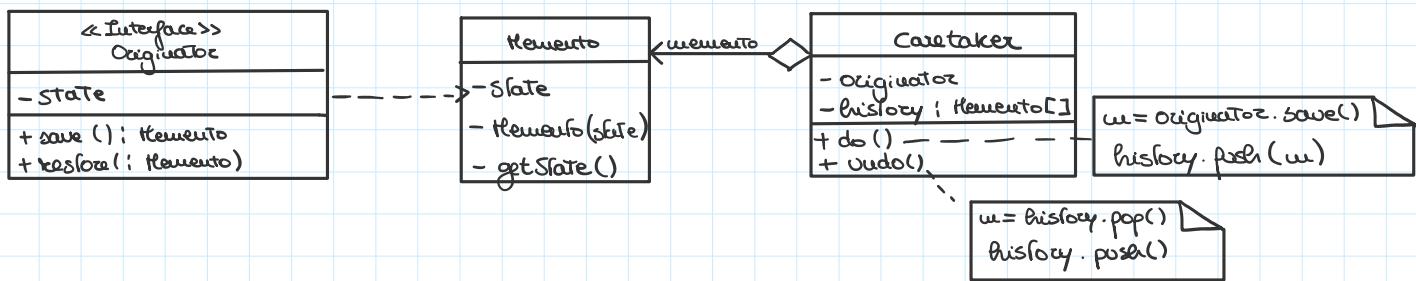
È possibile aggiungere nuovi comandi senza modificare le classi esistenti.

Memorizzando lo stato, tramite opportune funzioni di redo( ) e undo( ) è possibile ripetere o annullare un comando.

La memorizzazione può avvenire tramite elemento

• **Memento** (object-based) \* Originator

Usato per riportare lo stato di un oggetto in modo da poterlo ripristinare in seguito

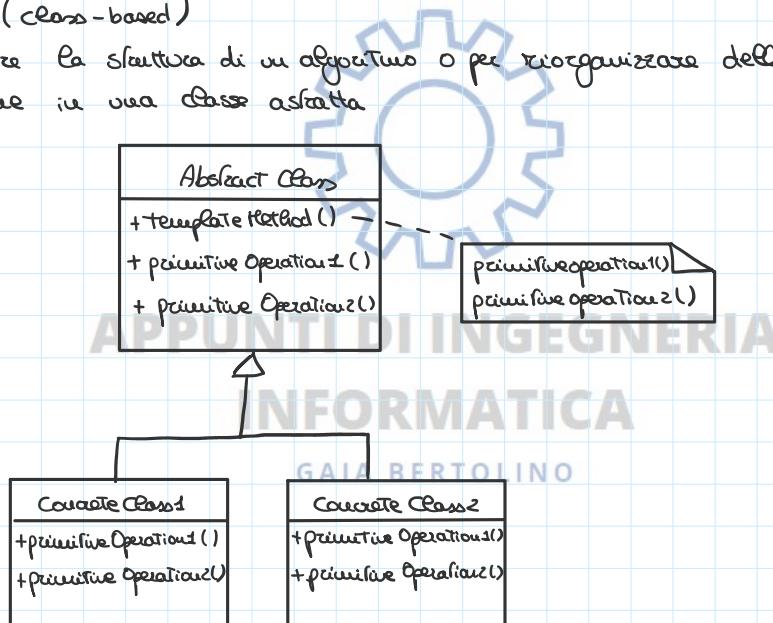


**PRO:** I dati rimangono incapsulati, protetti e accessibili solo all'Originator

**CONTRO:** Un elemento potrebbe essere oneroso da memorizzazione a causa della quantità di dati al suo interno

• **Template Method** (class-based)

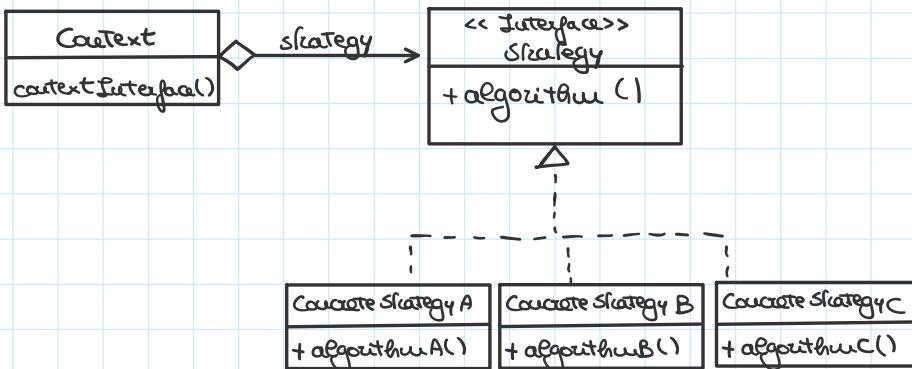
Usato per definire la struttura di un algoritmo o per riorganizzare delle classi spostando la parte comune in una classe astratta



Il pattern ribalta la situazione classica attraverso il principio Hollywood  
ovvero è il genitore a invocare i metodi concreti delle sottoclassi

- **Strategy** (object-based)

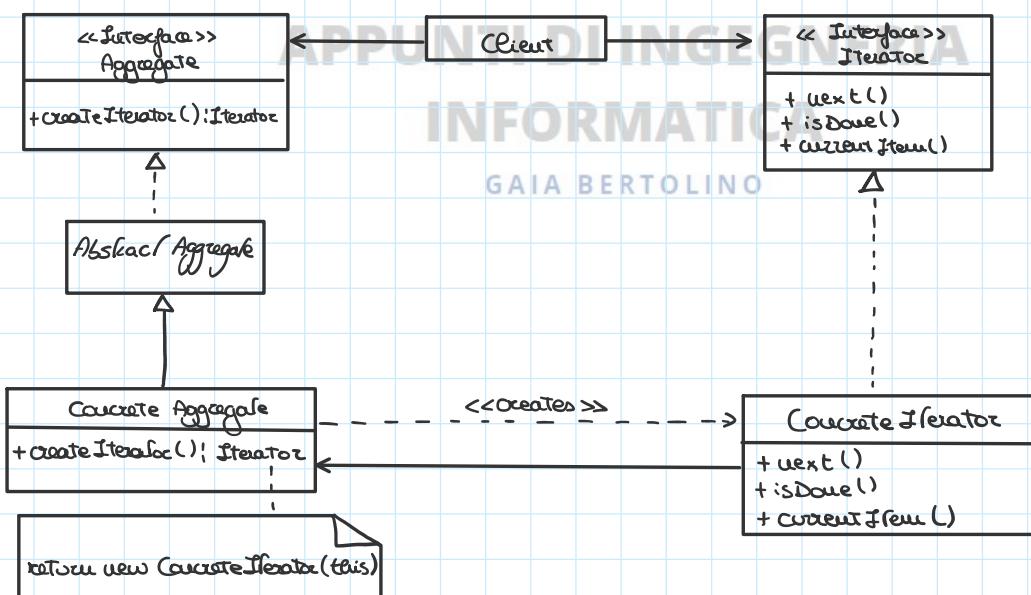
Usato per uniformare sotto un'unica interfaccia diversi algoritmi che risolvono lo stesso problema



**CONTRO:** Si può creare overhead di comunicazione per quegli algoritmi più semplici

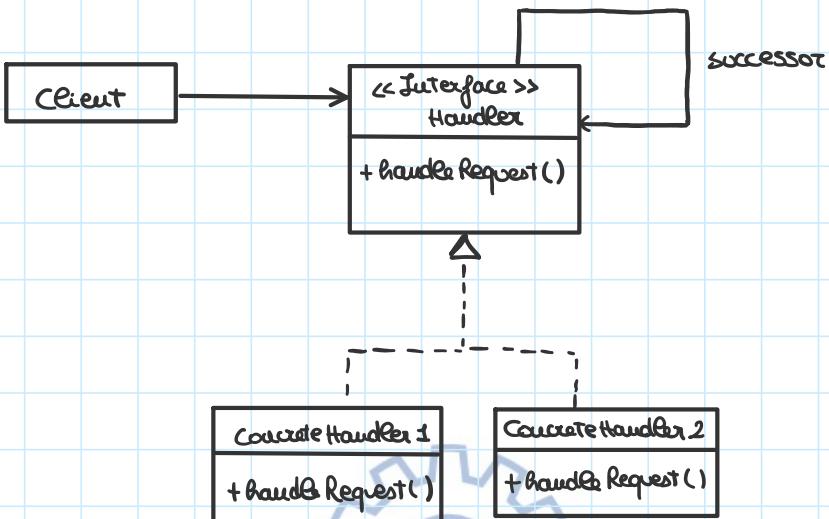
- **Iterator** (object-based)

Usato per fornire un accesso iterativo e sequenziale agli elementi di una collezione senza esporre la struttura interna



- **ChainOfResponsibility** (object-based)

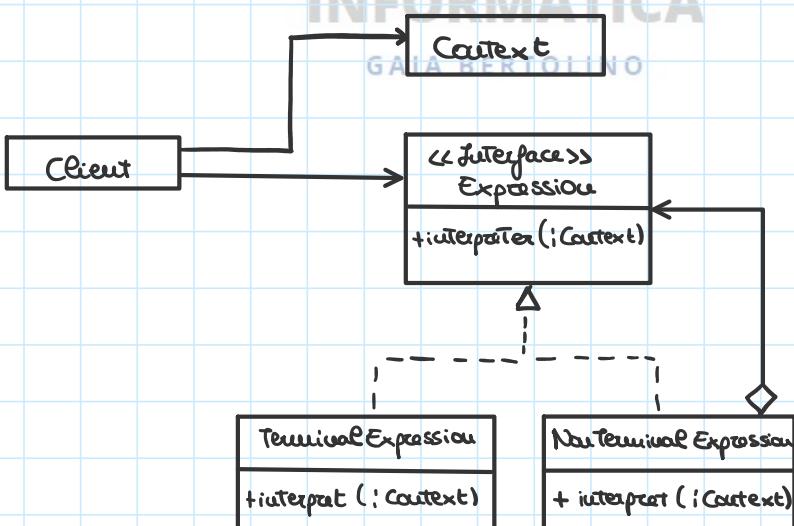
Usato per definire una catena di oggetti in grado di soddisfare una richiesta.  
Il comando viene inviato da un oggetto all'altro



**CONTRO:** Il riempimento delle responsabilità non porta alla risoluzione della richiesta

- **Interpreter** (class-based)

Usato per definire una rappresentazione della grammatica di un linguaggio

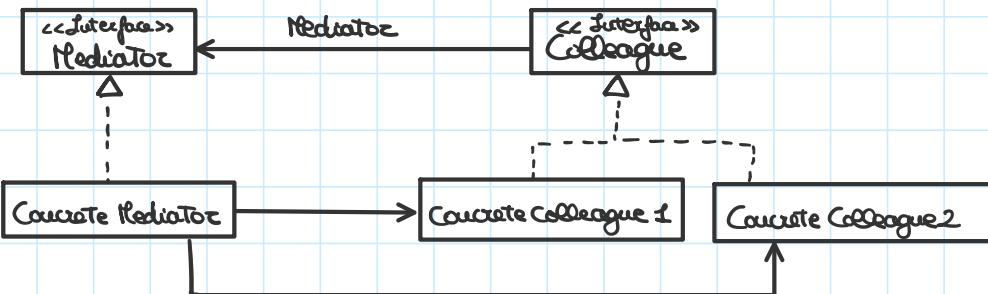


**PRO:** Facile implementazione

**CONTRO:** Non adatto a linguaggi: troppo complessi

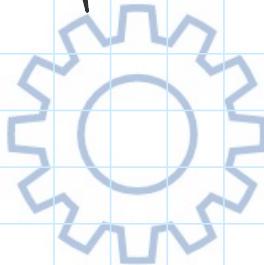
- **Mediator** (object-based)

Usato per ridurre i collegamenti diretti fra classi e dunque le loro dipendenze.



**PRO:** Permette l'implementazione di nuovi comportamenti uno-a-molti

**CONTRARIO:** Il corpo del Mediator può diventare molto pesante

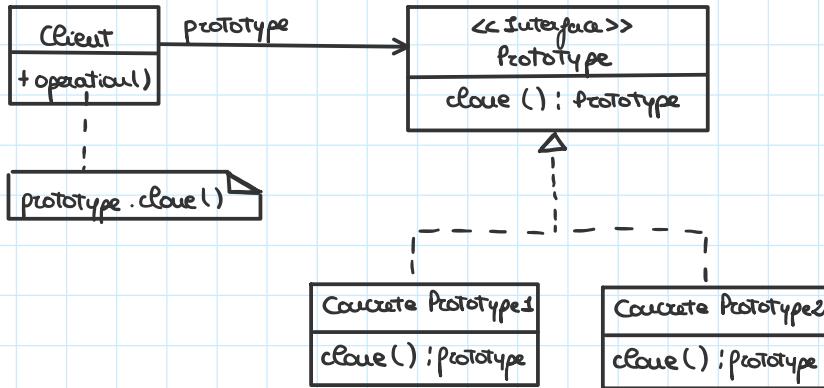


**APPUNTI DI INGEGNERIA  
INFORMATICA**

GAIA BERTOLINO

- **Prototype** (object-based) \* close

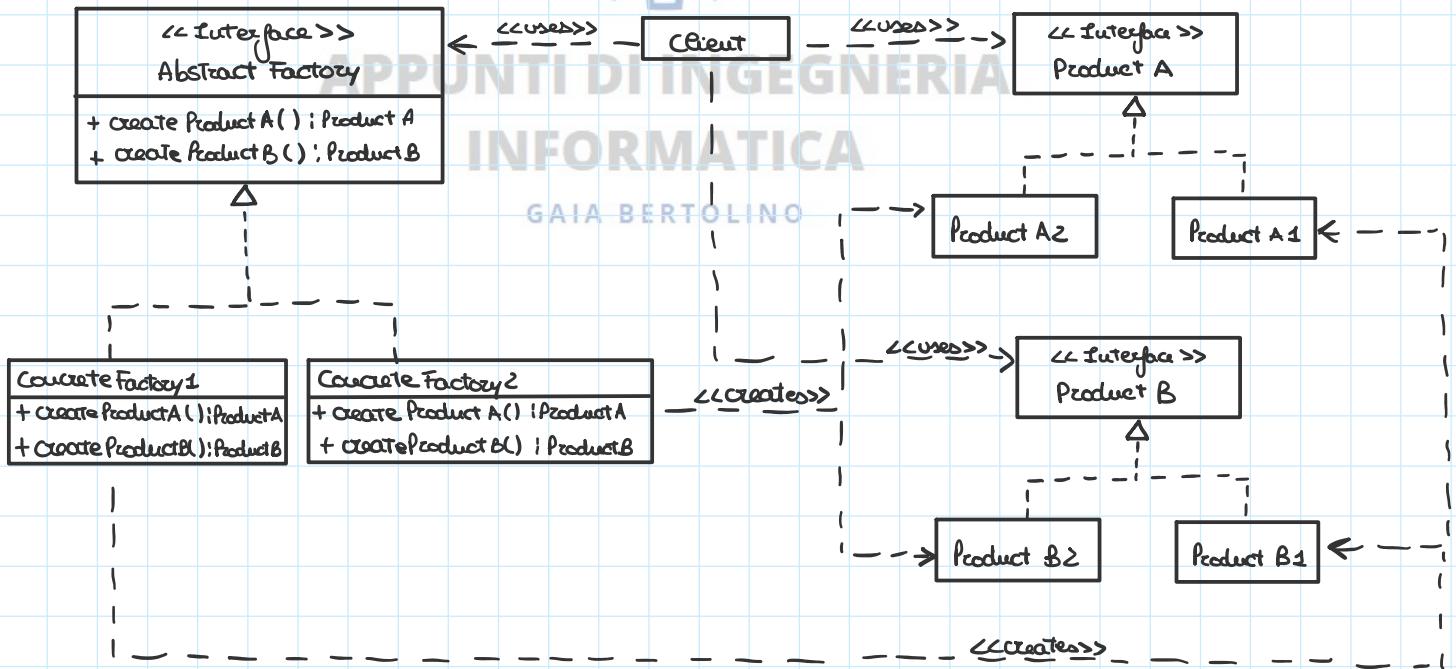
Usato per creare nuovi oggetti utilizzando alcuni prototipi già presenti.



**CONTRO:** Bisogna rispettare attenzione alla funzione di **clone** affinché esegua una copia profonda

- **Abstract Factory** (object-based) \* ConcreteFactory / Product A/B

Usato per creare oggetti senza conoscere l'implementazione e ritardandola



**CONTRO:** L'introduzione di un componente con un comportamento diverso richiede la modifica della classe **Abstract Factory** e di tutte le classi eredi.