

Introduzione

lunedì 28 febbraio 2022 15:33

INGEGNERIA DEL SOFTWARE

- Applicazione di un approccio sistematico, disciplinato e quantificabile nello sviluppo, funzionamento e manutenzione del software. [IEEE 610.12-1990 *Glossario standard della terminologia dell'ingegneria del software*]
- Creazione di software multiversione da parte di più operatori. [Parnas, 1978]

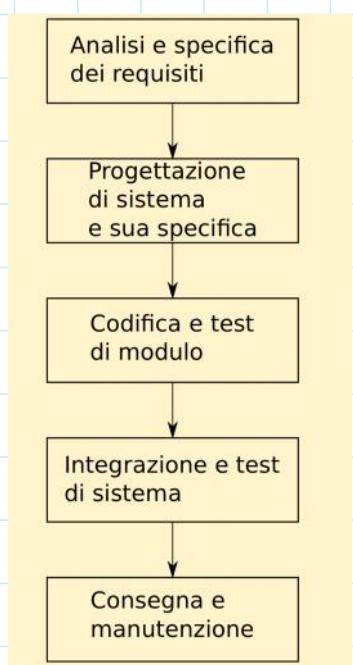
Rispetto alla programmazione, l'ingegneria del software richiede un approccio di gruppo mentre la programmazione è un atto individuale.

Un esempio di ingegneria del software sono le collections di Java che vengono utilizzate per costruire altri sistemi.

Un buon ingegnere deve essere in grado di rispettare le richieste dei cosiddetti **STAKEHOLDERS** (poterai d'interesse) ovvero i clienti / persone interessate del lavoro.

CICLO DI VITA

È dello **modello a cascata** e ogni fase restituisce un prodotto che viene passato al livello successivo.



{ Eseguire a valle di uno studio di fattibilità

progettazione ↗ architettonica → componentistica ad alto
livello
di dettaglio

{ La manutenzione dura in base all'uso / diffusione

SOFTWARE

Il prodotto dell'ingegneria del software è un sistema software che è utile ovvero è possibile modificare il prodotto anche il progetto, la quale qualità è però un'arma a doppio taglio.

Il costo di un software è determinato non dalla fabbricazione ma dalla progettazione e implementazione.

Le qualità di un software sono di due tipi:

- esterne ovvero quelle visibili agli utenti
- interne ovvero quelle che riguardano gli sviluppatori. Esse determinano il conseguimento di quelle esterne.

I software sono sottoposti a test come ad esempio ai **Test di regressione** i quali vanno a verificare che un software a seguito dell'aggiunta di nuove funzionalità ripresenti dei bug precedentemente fissati (ovvero retrodisca).

La **correttezza** di un software richiede il soddisfacimento di requisiti di due tipologie:

- funzionali ovvero esprimibili tramite una funzione
- non funzionali come ad esempio prestazioni e scalabilità.

Le specifiche tuttavia possono essere espresse in maniera ambigua a causa della mancanza di un linguaggio formale.

La correttezza si può verificare con diversi metodi:

- Testing ovvero verificare l'output a determinati input
- approcci analitici: ispezione del codice, verifica formale

L'**affidabilità** è un concetto che ingloba la correttezza. Un software si dice affidabile se si comporta come ci si attende che esso faccia.

La correttezza si considera come una proprietà assoluta mentre l'affidabilità è un concetto relativo in quanto un sistema non corretto può essere considerato affidabile.

Tuttavia un software non potrà funzionare i requisiti i comuni a un pa-

affidabile.

Tuttavia, un software può essere corretto (avendo verificato i requisiti) ma ha delle specifiche incorrecte allora il risultato prodotto può non essere quello desiderato.

Per **robustezza** si intende la capacità di comportarsi in modo accettabile anche in circostanze non previste dalle specifiche.

Un requisito desiderabile deve essere sempre opportunamente inserito per evitare che ci sia un problema di robustezza.

Le **prestazioni** sono influenzate dall'efficienza di un sistema ad esempio se utilizza troppe risorse non ha un buon livello di prestazioni.

Le prestazioni influenzano la **scattabilità** ovvero la capacità di funzionare correttamente con input più grandi.

La valutazione delle prestazioni può avvenire nei seguenti modi:

- misura
- analisi
- simulazioni

APPUNTI DI INGEGNERIA INFORMATICA GAIA BERTOLINO

Con **usabilità** si intende una qualità soggettiva in quanto essa riguarda il livello di proprietà user-friendly che un software possiede. Dunque è molto legata all'interfaccia messa a disposizione per accedere al software stesso.

La **verificabilità** riguarda la verifica della correttezza delle prestazioni che è facilitata dall'utilizzo di metodi di progettazione modulare e linguaggi di programmazione appropriati.

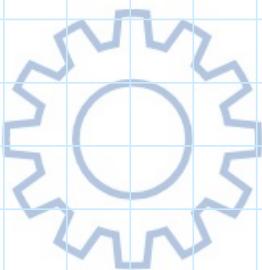
La verificabilità è una qualità interna ma nel caso di sistemi critici (es. centrali nucleari) diventa esterna in quanto va ad influenzare chi è esterno.

Altre qualità sono:

- manutenzione

Altre qualità sono:

- manutenibilità
- riparabilità
- evolvibilità ovvero un approccio modulare permette una evoluzione maggiormente isolata (con minore possibilità di regressione o introduzione di bugs)
- riutilizzabilità ovvero



APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

Processo di produzione del software

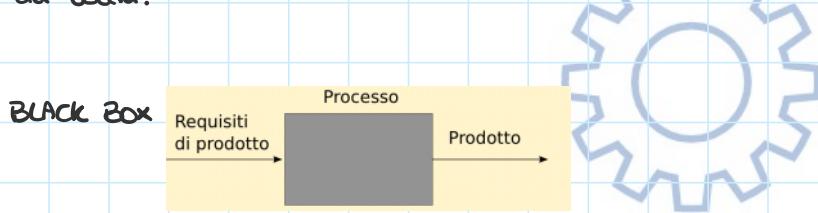
giovedì 3 marzo 2022 12:12

Agli inizi utente e programmatore corrispondevano. L'approccio era di tipo **Code & Fix** ovvero di scrittura e fixing degli errori. Tuttavia tale approccio rende sempre più difficili le modifiche e il codice diventa ingessato per i seguenti motivi:

- mancanza di documentazione
- scarsa anticipazione del cambiamento ovvero utilizzare la modifica per rendere il codice più flessibile per i cambiamenti.

Da questo approccio fallimentare nasce poi l'ingessatura del software.

Si introduce quindi il concetto di **modello di processo di sviluppo** che è una serie di meccanismi che seguono seguiti affinati, tenendo sotto controllo il processo ed eseguito delle istruzioni ordinate, è più facile avere controllo sulla qualità ottenuta. Se non si usa un modello specifico si parla di **black box** dove i requisiti fanno da ingresso e il prodotto da uscita.



Le parti principali di un processo di sviluppo sono:

- studio di fattibilità che viene fatto preliminarmente per valutare se procedere con lo sviluppo. Esso viene eseguito attraverso una simulazione della realizzazione del progetto.
- acquisizione, analisi e specifica dei requisiti ovvero descrive quali sono le qualità da assicurare. Tuttavia NON si deve specificare come queste verranno implementate (che è compito di un'altra fase).

In questa fase è importante comprendere la necessità dello stakeholder e le circostanze di applicazione.

Per quanto riguarda il documento di specifica esso è caratterizzato dalle seguenti parti:

- Dominio**
Breve descrizione del dominio applicativo e degli obiettivi da raggiungere.
 - Chi sono gli utenti, quali sono le loro aspettative ed i loro obiettivi?
 - Quali sono le principali entità che caratterizzano il dominio?
 - Quali sono le loro relazioni? Che influssi avrà il sistema su di esse?
- Requisiti funzionali**
 - Descrivono cosa dovrà fare il prodotto.
 - Devono essere espressi in una notazione opportuna (formale o semiformale)
- Requisiti non funzionali**
 - Affidabilità, accuratezza, prestazioni
 - Interfaccia tra sistema e utente
 - Limiti operativi, limiti fisici, portabilità
- Requisiti del processo di sviluppo e manutenzione**
 - Procedure per il controllo della qualità
 - Priorità di sviluppo tra le funzioni
 - Possibili cambiamenti del sistema

La documentazione può seguire una specifica data dalla compagnia come l'UML.

- I requisiti sono raggruppabili, in rapporto alla loro priorità/criticità, tra:
- **MUST**: insieme minimo di requisiti per considerare "accettabile" il comportamento del sistema da realizzare
 - **SHOULD**: requisiti desiderabili in quanto rendono più completo il sistema, ma la cui omissione non compromette nessuna funzionalità di base. La loro implementazione/integrazione non dovrebbe richiedere modifiche profonde alla struttura del progetto
 - **MAY**: sono requisiti la cui presenza nel progetto è facoltativa: in caso di mancanza di risorse (tempo) possono essere traslati inizialmente. Tuttavia, la loro integrazione successiva potrebbe comportare modifiche significative al sistema, come il riprogetto o l'aggiunta di nuove classi

- **integrazione e testing** del sistema che prevede allo stadio finale l'**alpha testing** (interno) che precede il **beta testing** (riservato all'utente).

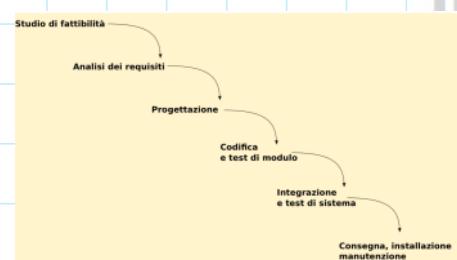
:



Modelli

Tali modelli sono alternativi al code & fix!

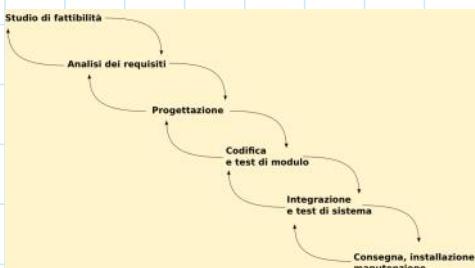
- **Modello a cascata**



GALA BERTOLINO

Prevede che ci sia un solo errore. È rigido e manca il feedback fino al rilascio della beta. Inoltre, manca parallelismo fra le diverse parti di sviluppo.

- **Modello a cascata con feedback**



Si inserisce un feedback continuo per poter utilizzare i feedback.

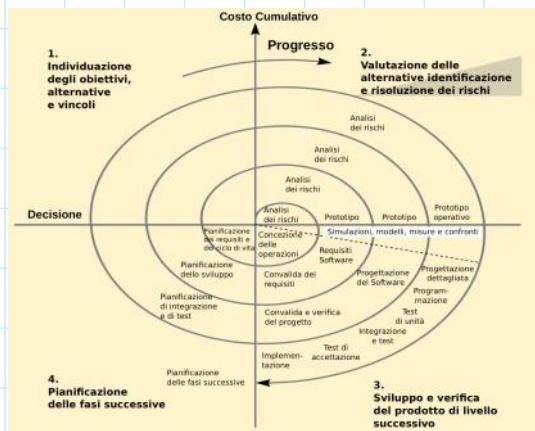
Modelli evolutivi

Detti anche **INCREMENTALI**, si basano sul concetto **do it twice** avendo la prima

Verifica del prodotto è da continuare in quanto viene vista come di test.

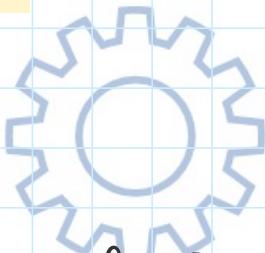
Si parla di **fasi incrementali** quando di un prodotto vengono rilasciate le parti ottenute in fasi intermedie.

• Modello a spicce



Vi sono quattro quadranti. Lungo i quali si svolge la spicce.

Il raggio rappresenta il costo accumulato mentre la dimensione angolare il progresso che si svolge dal primo quadrante al quarto.



TIPOLOGIE DI SOFTWARE

• Software open source

I software open source hanno una regolamentazione delle licenze tali che

- l'uso non è limitato
- si possono fare copie liberamente
- possono essere introdotte nuove modifiche
- non ci sono limitazioni sulla distribuzione sul software o sui derivati

Queste licenze open source specificano inoltre la condizione di copyleft tale che versioni modificate vengono rilasciate negli stessi termini della licenza originale.

Alla base di un progetto open source vi è un piccolo gruppo di sviluppatori e da un grande gruppo di contributori che, utilizzando il software, contribuiscono a migliorarlo e a risolvere i bug.

Elenco pattern

giovedì 23 giugno 2022 15:18

Pattern creazionali

I pattern creazionali risolvono problematiche inerenti all'istanziazione degli oggetti

- L'[Abstract factory](#) (letteralmente, "fabbrica astratta") fornisce un'interfaccia per creare famiglie di oggetti connessi o dipendenti tra loro, in modo che non ci sia necessità da parte degli utilizzatori di specificare i nomi delle classi concrete all'interno del proprio codice.
- Il [Builder](#) ("costruttore") separa la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che il processo di costruzione stesso possa creare diverse rappresentazioni.
- Il [Factory method](#) ("metodo fabbrica") fornisce un'interfaccia per creare un oggetto, ma lascia che le sottoclassi decidano quale oggetto istanziare.
- La [Lazy initialization](#) ("inizializzazione pigra") è la tattica di istanziare un oggetto solo nel momento in cui deve essere usato per la prima volta. È utilizzato spesso insieme al pattern *factory method*.
- Il [Prototype pattern](#) ("prototipo") permette di creare nuovi oggetti clonando un oggetto iniziale, o prototipo.
- Il [Singleton](#) ("singoletto") ha lo scopo di assicurare che di una classe possa essere creata una sola istanza in sistemi con un unico thread.
- Il [Double-checked locking](#) ("Interblocco ricontrallato") ha lo scopo di assicurare che di una classe possa essere creata una sola istanza in sistemi multithread.

Pattern strutturali

I pattern strutturali risolvono problematiche inerenti alla struttura delle classi e degli oggetti

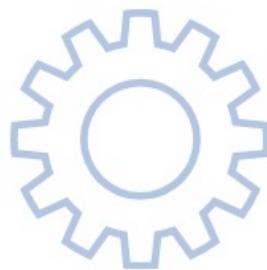
- L'[Adapter](#) ("adattatore") converte l'interfaccia di una classe in una interfaccia diversa.
- [Bridge](#) ("ponte") permette di separare l'astrazione di una classe dalla sua implementazione, per permettere loro di variare indipendentemente.
- Il [Composite](#) ("composto"), utilizzato per dare la possibilità all'utilizzatore di manipolare gli oggetti in modo uniforme, organizza gli oggetti in una struttura ad albero.
- Il [Container](#) ("contenitore") offre una soluzione alla rottura dell'incapsulamento per via dell'uso dell'ereditarietà.
- Il [Decorator](#) ("decoratore") consente di aggiungere metodi a classi esistenti durante il *run-time* (cioè durante lo svolgimento del programma), permettendo una maggior flessibilità nell'aggiungere delle funzionalità agli oggetti.
- [Extensibility](#) ("estendibilità")
- Il [Façade](#) ("facciata") permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse e diverse tra loro.
- [Flyweight](#) ("peso piuma"), che permette di separare la parte variabile di una classe dalla parte che può essere riutilizzata.
- [Proxy](#) fornisce una rappresentazione di un oggetto di accesso difficile o che richiede un tempo importante per l'accesso o creazione. Il Proxy consente di posticipare l'accesso o creazione al momento in cui sia davvero richiesto.
- [Pipes and filters](#) ("condotti e filtri")
- [Private class data](#) ("dati di classe privati")

Pattern comportamentali

I pattern comportamentali forniscono soluzioni alle più comuni tipologie di interazione tra gli oggetti.

- [Chain of Responsibility](#) ("catena di responsabilità") diminuisce l'accoppiamento fra l'oggetto che effettua una richiesta e quello che la soddisfa, dando a più oggetti la possibilità di soddisfarla
- Il [Command](#) ("comando") permette di isolare la porzione di codice che effettua un'azione dal codice che ne richiede l'esecuzione.
- [Event listener](#) ("ascoltatore di eventi")
- [Hierarchical Visitor](#) ("visitatore di gerarchia")
- [Interpreter](#) ("interprete") dato un linguaggio, definisce una rappresentazione della sua grammatica insieme ad un interprete che utilizza questa rappresentazione per l'interpretazione delle espressioni in quel determinato linguaggio.
- L'[Iterator](#) ("iteratore") risolve diversi problemi connessi all'accesso e alla navigazione attraverso gli elementi di una struttura dati, senza esporre i dettagli dell'implementazione e della struttura interna del contenitore.
- Il [Mediator](#) ("mediatore") si interpone nelle comunicazioni tra oggetti, allo scopo di aggiornare lo stato del sistema quando uno qualunque di essi comunica un cambiamento del proprio stato.
- Il [Memento](#) ("promemoria") è l'operazione di estrarre lo stato interno di un oggetto, senza violarne l'incapsulazione, e memorizzarlo per poterlo ripristinare in un momento successivo.

- L'[Observer](#) ("osservatore") definisce una dipendenza uno a molti fra oggetti diversi, in maniera tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti vengono notificati del cambiamento avvenuto e possono aggiornarsi.
- [Single-serving Visitor](#)
- [State](#) ("stato") permette ad un oggetto di cambiare il suo comportamento al cambiare di un suo stato interno.
- Lo [Strategy](#) ("strategia") è utile in quelle situazioni dove è necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione.
- Il [Template method](#) ("metodo schema") permette di definire la struttura di un algoritmo lasciando alle sottoclassi il compito di implementarne alcuni passi come preferiscono.
- Il [Visitor](#) ("visitatore") permette di separare un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da poter aggiungere nuovi comportamenti senza dover modificare la struttura stessa.
- [Null object](#) ("oggetto nullo") permette di sostituire un riferimento nullo con un oggetto che non fa nulla.

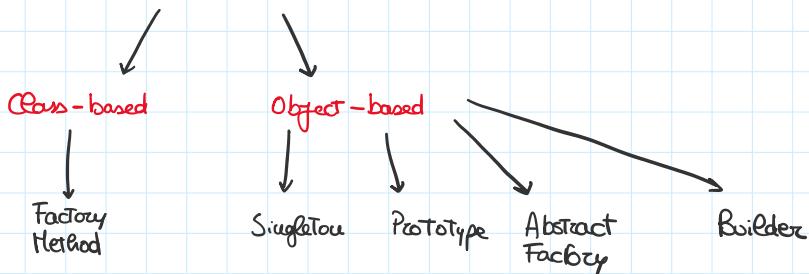


APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

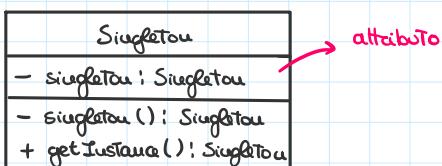
CREAZIONALI

PATTERN CREAZIONALI



• Singleton (object-based)

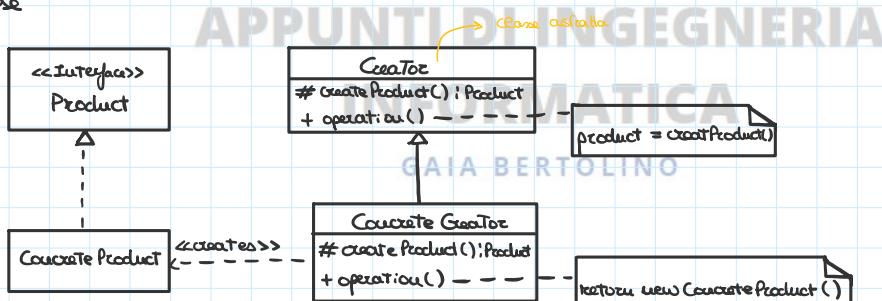
Usato per limitare la creazione per il tipo singleton ad un solo oggetto



CONTRO! bisogna realizzare la sincronizzazione per l'accesso e il metodo readResolve() se la classe è serializzabile per evitare copie.

• Factory method (class-based) * Creaore (prodotto)

Usato per realizzare la creazione di un oggetto al di fuori della sua classe

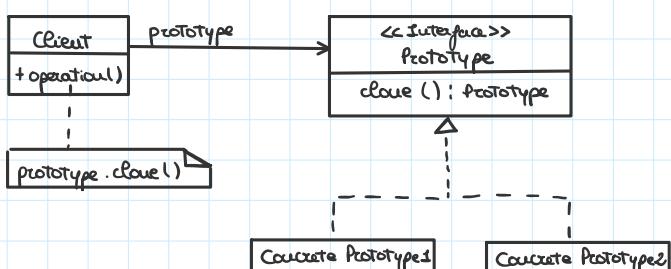


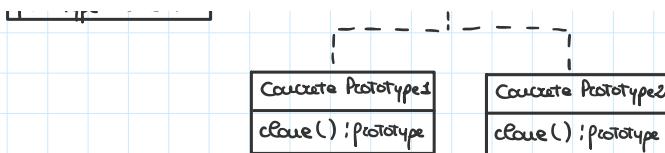
es. I prodotti sono linee + poligoni (ovvero sottiuscimenti di figura) e ciascuno di essi necessita di una classe concreta che implementi la realizzazione

CONTRO! Per ciascun oggetto è necessario istanziare una classe concreta ConcreteCreator oltre che a creare una estensione della classe Product

• Prototype (object-based) * clone

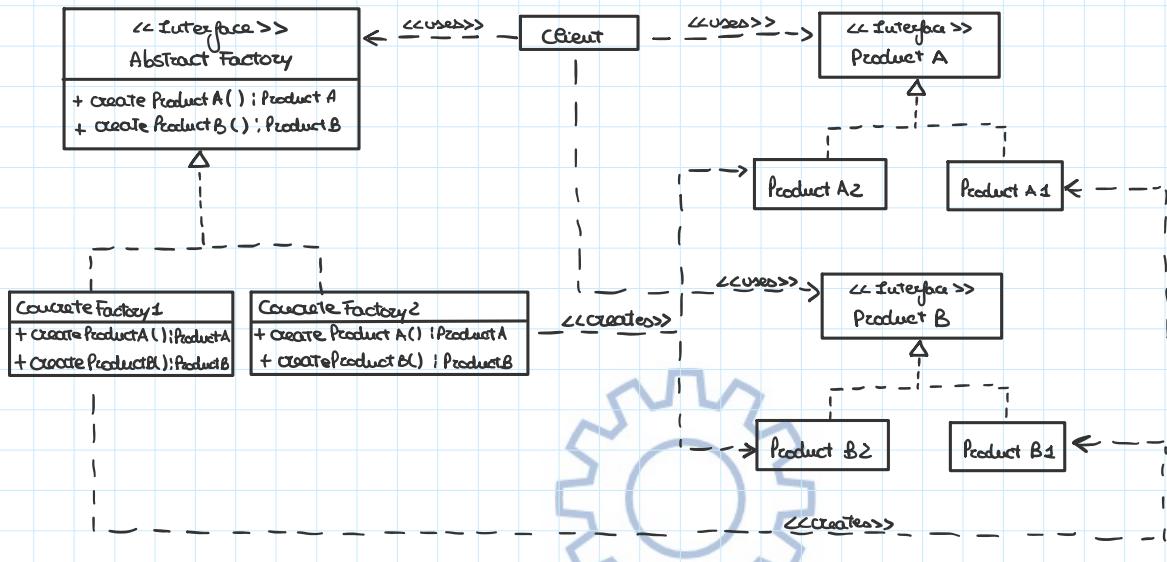
Usato per creare nuovi oggetti utilizzando alcuni prototipi già presenti





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

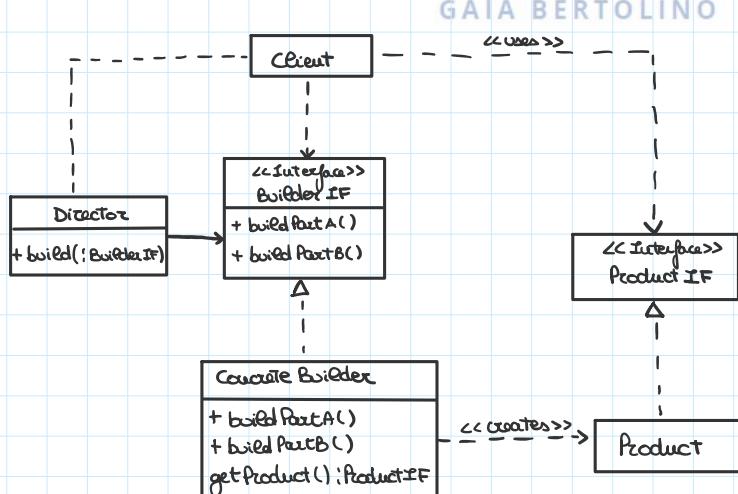
- **Abstract Factory (object-based)** * ConcreteFactory / Product A/B
Usato per creare oggetti senza conoscere l'implementazione e ritardandola



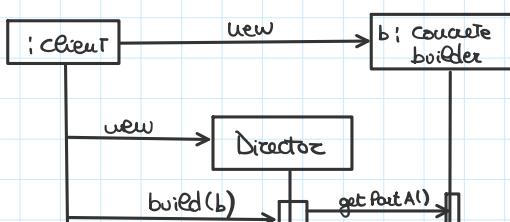
CONTRO: L'introduzione di un componente con un comportamento diverso richiede la modifica della classe Abstract Factory e di tutte le classi eredi.

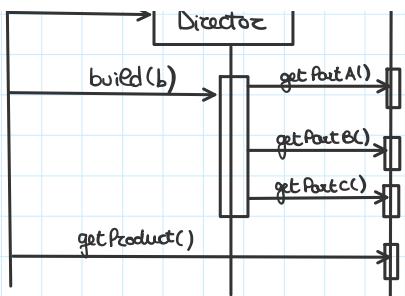
- **Builder (object-based)** * Director / Builder IF

Usato per definire e separare il processo di costruzione di un oggetto dalla realizzazione specifica dei passi di costruzione



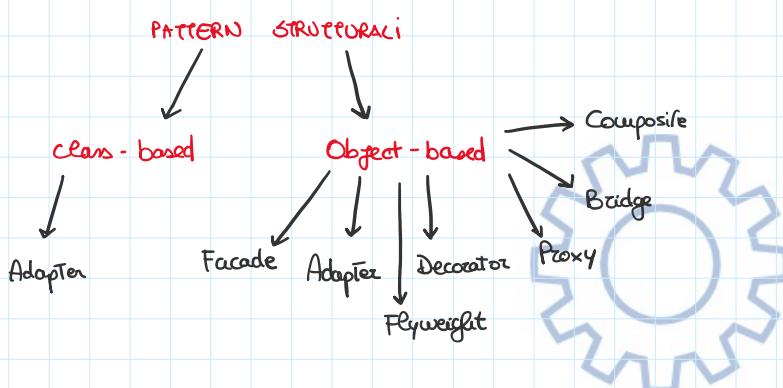
Sequence diagram





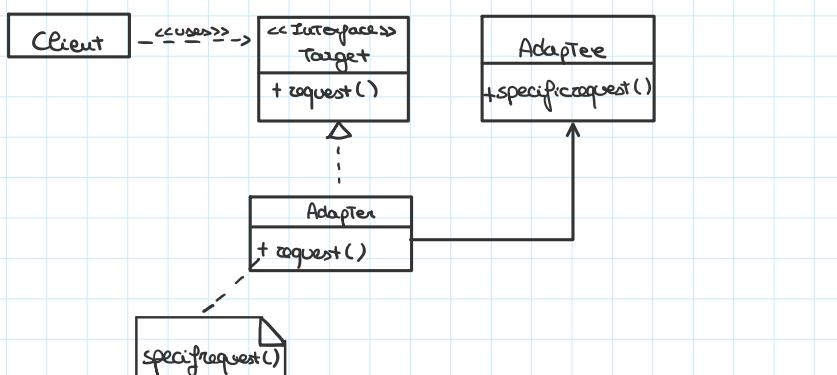
PRO: È una valida alternativa ai metodi costruttori telescopici e alla tecnica Java Beans che prevede la modifica dei campi attraverso dei metodi setta in quanto prevede un metodo per ogni parametro specificato e restituisce il build stesso.

PATTERN STRUTTURALI



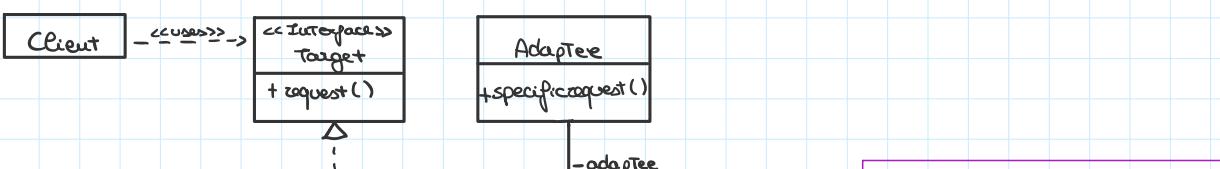
- **Adapter** (class-oriented / object-oriented) * Target
Usato per adattare una classe esistente ai meccanismi di una interfaccia conosciuta dal cliente

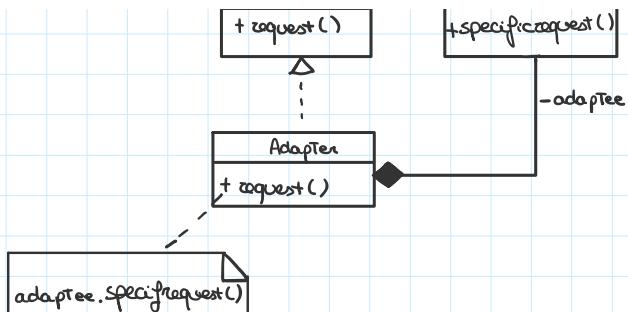
Versione class-based



CONTRO: ha limiti di applicabilità se Target è una classe astratta
o se Adaptee è final, interfaccia o astratta

Versione object-based

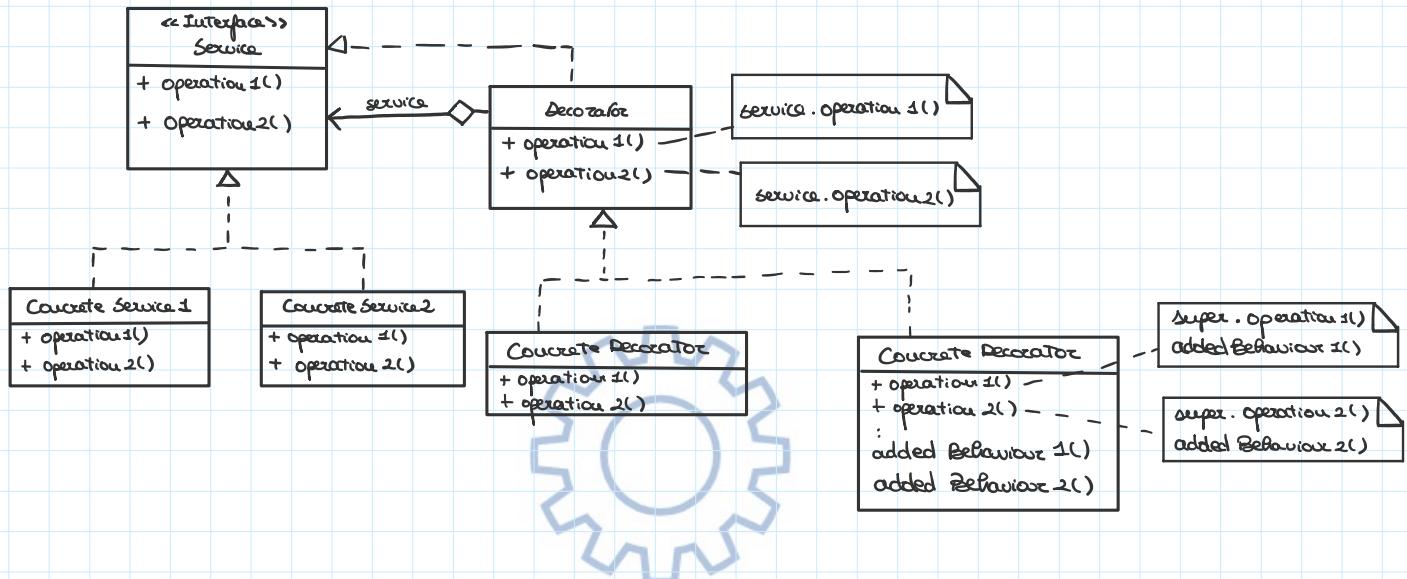




L'esecuzione si compone a tempo
di compilazione mentre la
composizione a runtime.

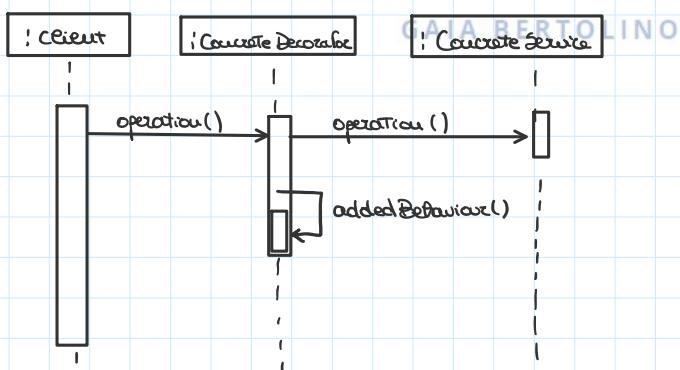
• Decorator (object-based) * Decorator (source)

Usato per poter aggiungere in runtime delle caratteristiche ad un oggetto



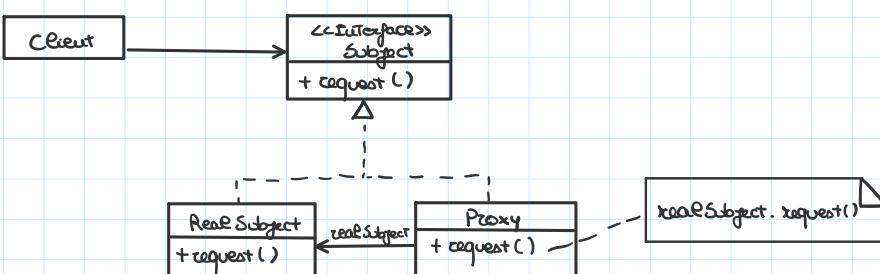
es. Nel framework di Java Swing, dei componenti sono combinati secondo decorator come ad esempio
JTextArea che viene decorato da JScrollPane

Sequence diagram



• Proxy (object-based) * Subject

Usato per introdurre una classe di filtro fra cliente e oggetto

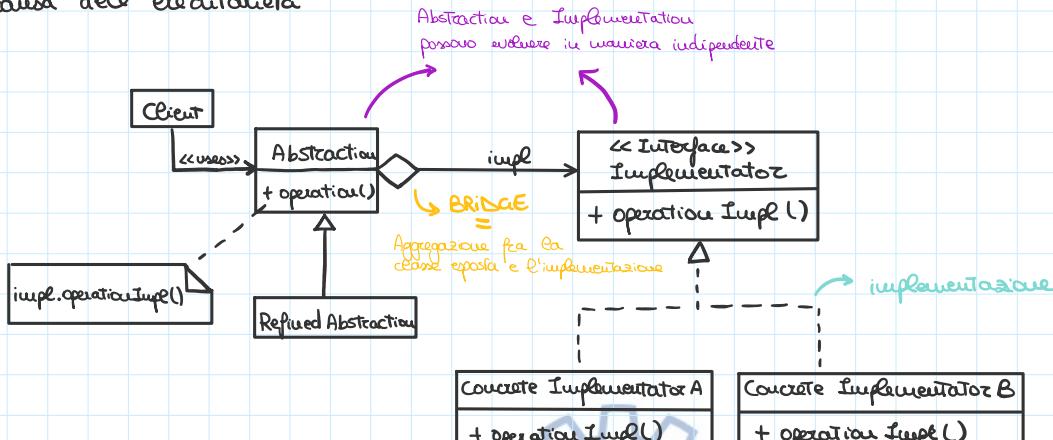


- APPLICAZIONI :
- Proxy remoto
 - Proxy virtuale
 - Proxy di protezione
 - Smart reference

• **Bridge** (object-based) * Abstraction / Implementation / operation Impl

Usato quando si vuole creare un disaccoppiamento fra un concetto e la sua implementazione.

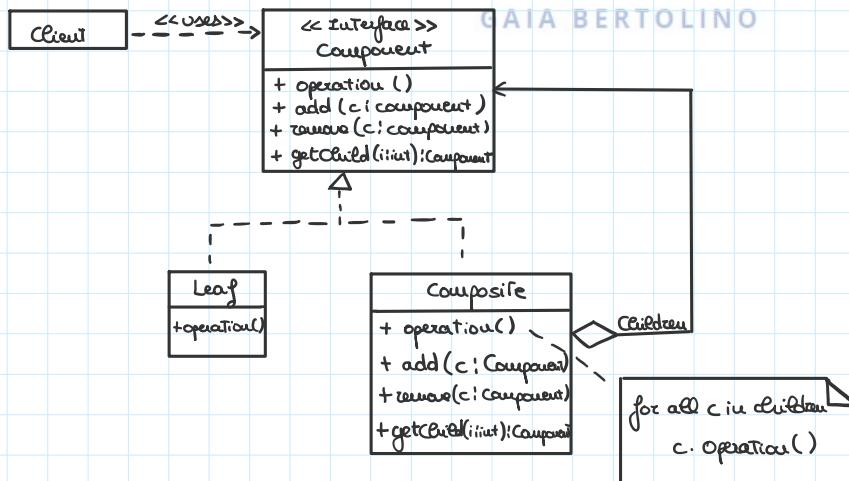
Usare una classe astratta o una interfaccia vi è una forte dipendenza a causa dell'ereditarietà



es. Per realizzare i concetti di finestra e suoi sottotipi (gli eredi) e le possibili implementazioni in base all'ambiente grafico

• **Composite** (object-based) * composite / composite / Leaf

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



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

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

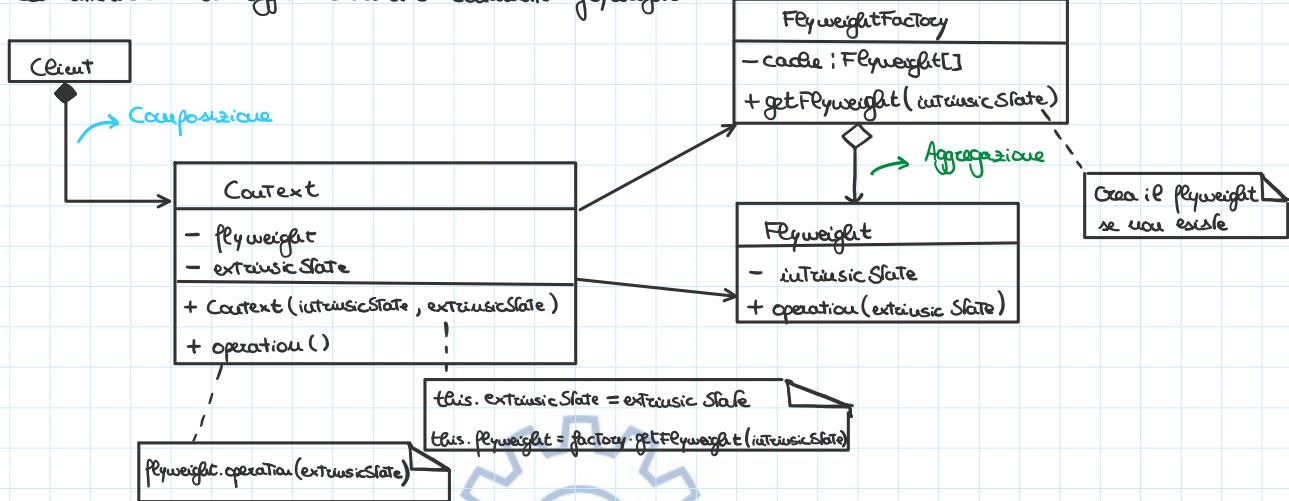
nel composite esistono tante interfacce ma non tutte sono necessarie
nelle frane va contro il principio di Liskov in quanto sono eredi di componenti
ma non possono essere usate al suo posto.

• Facade (object-based)

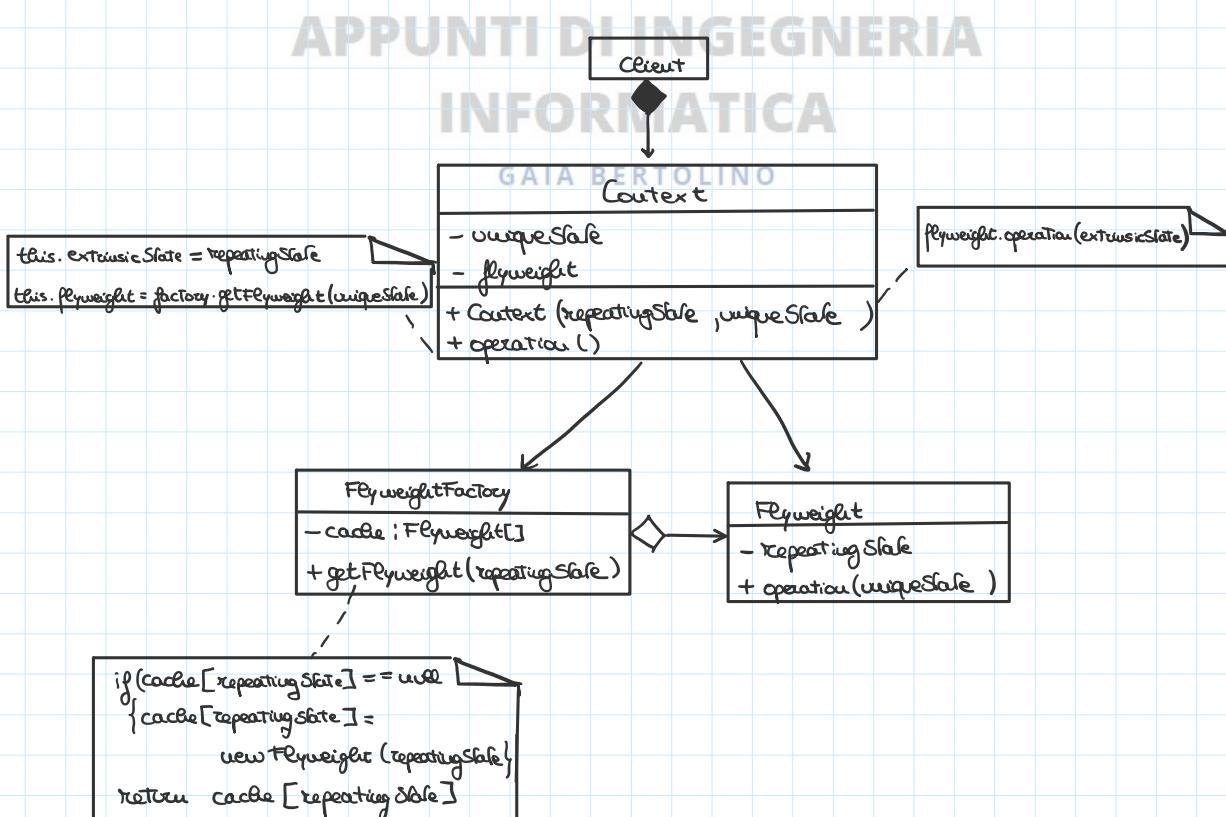
Usata per avere una interfaccia unificata che è l'unica ad interagire con il client

• Flyweight (object-based) * Context / flyweightFactory

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



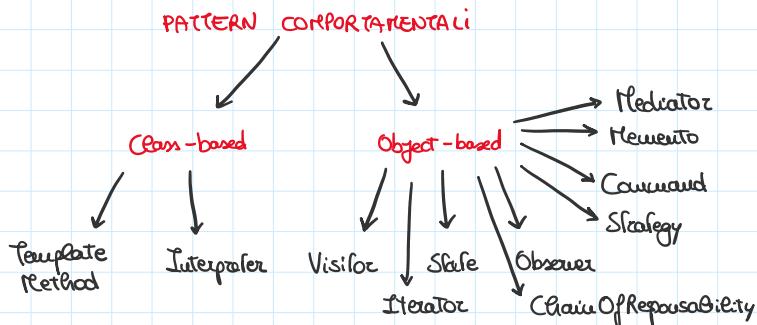
es. In un videogioco, i proiettili sparati in una partita hanno tutti le stesse caratteristiche di base (ovvero lo stato interno) che sono condivisibili: una diversa velocità (ovvero stato esterno) che non sono condivisibili fra oggetti.



CONTRO: se l'individuazione dello stato esterno è difficoltosa e la sua memorizzazione
non offre vantaggi nel caso di memorizzazione applicare il pattern è inutile

CONTRO: se l'individuazione dello stato esterno è difficoltosa e la sua rimozione non crea vantaggi nel caso di memorizzazione applicare il pattern è inutile

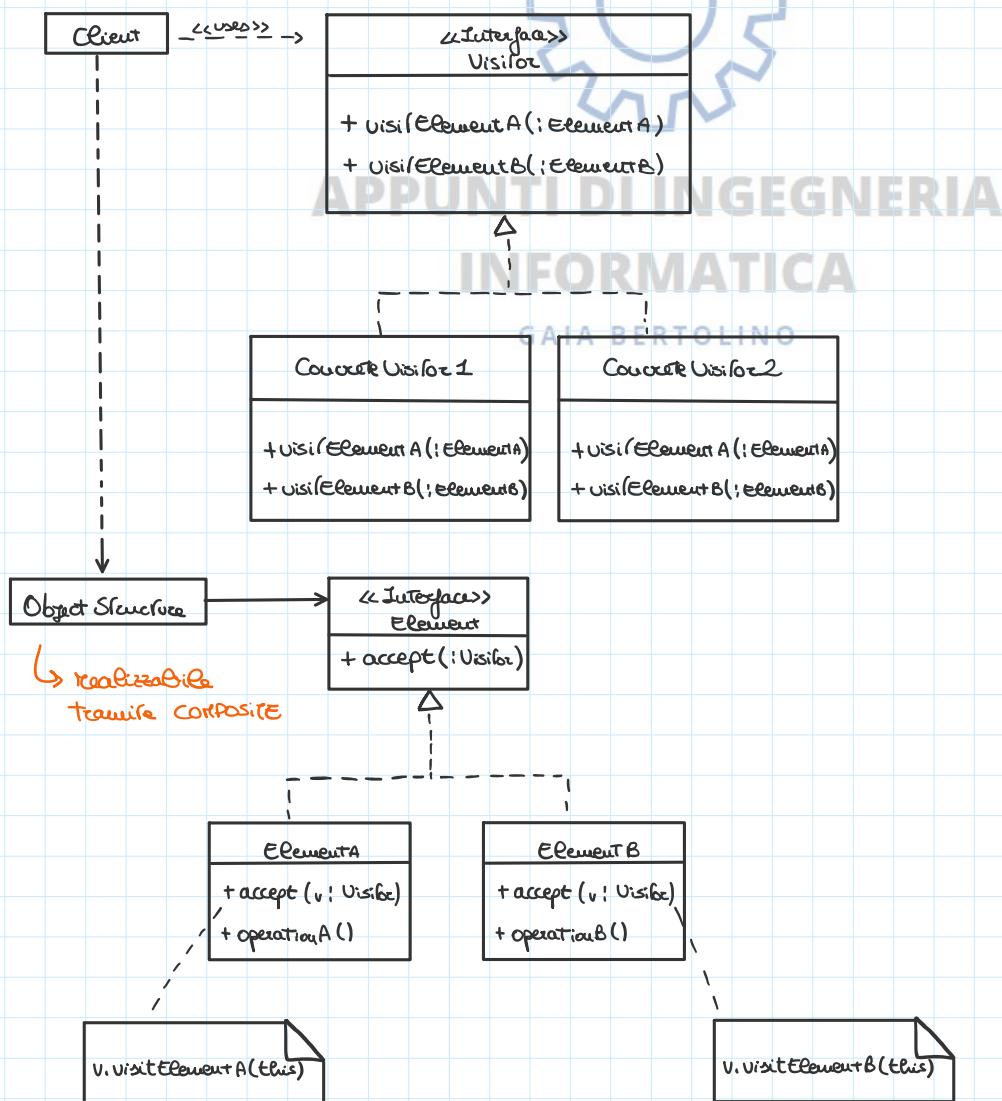
PATTERN COMPORTAMENTALE



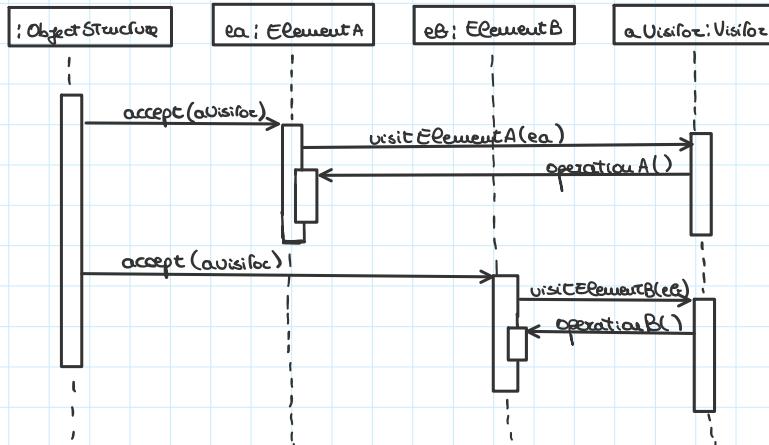
- **Visitor** (object-based) * Element / Visitor

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

Class diagram



Sequence diagram



INATRO DOPPIO (es. accept(aVisitor))

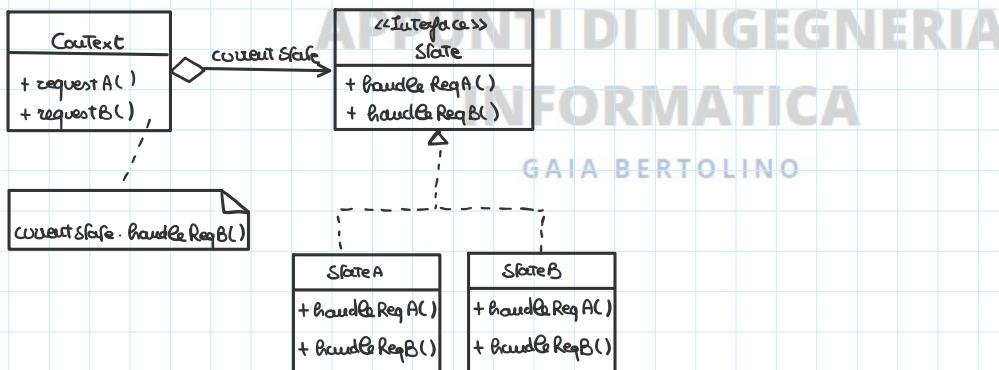
PRO: il pattern permette di aggiungere delle operazioni ad un gruppo di oggetti quando questi vengono visitati senza dover alterare le sue classi. Inoltre, i tipi di elementi appartenenti ad un visitor possono essere diversi (e non uguali come in Iterator).

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.

• State (object-based) *Context / State

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



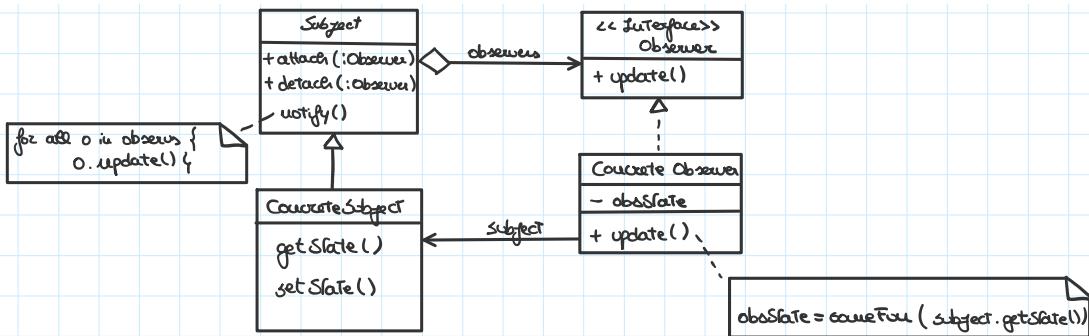
PRO: Le transizioni fra stati sono atomiche. Con pattern Flyweight si possono definire le parti di stato che si ripetono

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

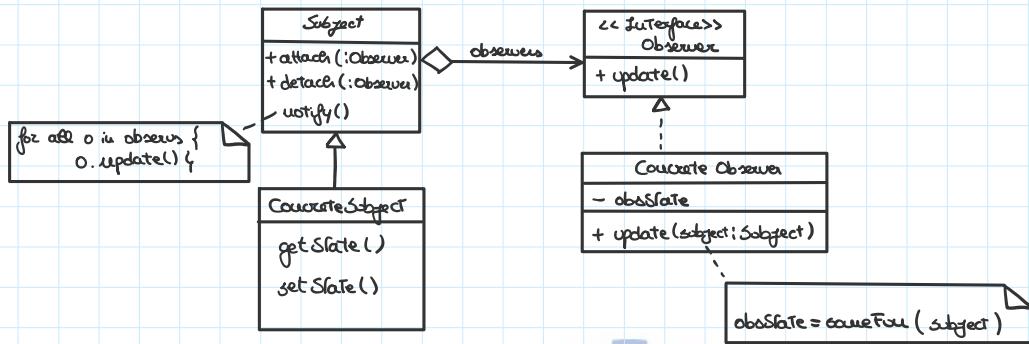
• Observer (object-based) * Subject / Observer

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

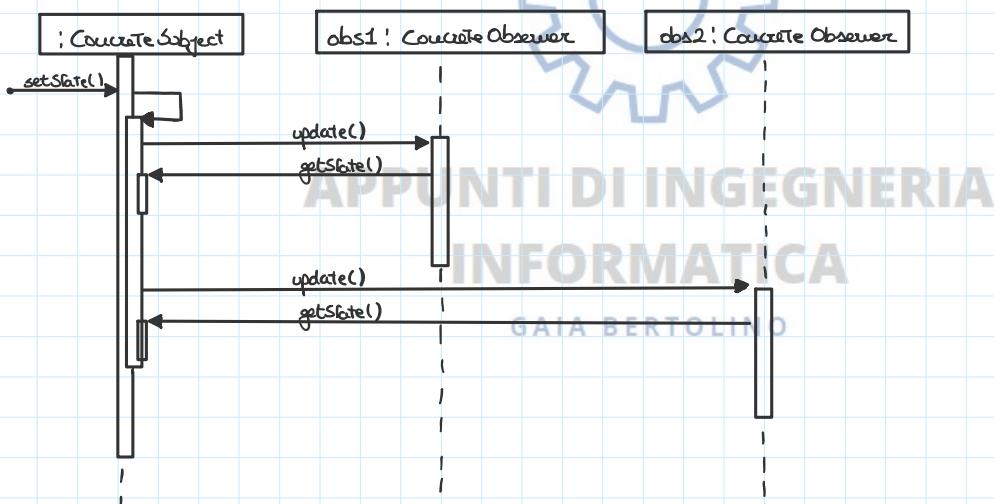
Modello ad un solo object



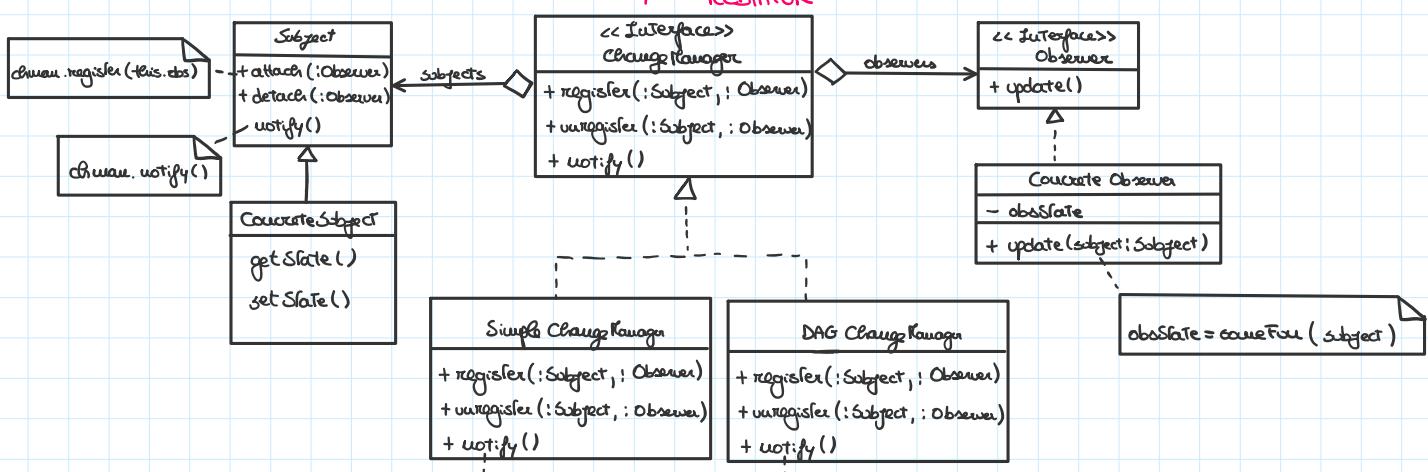
Modello a più subject

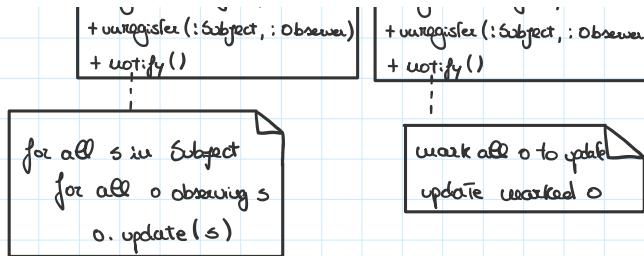


Sequenza diagramm



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 influenza tutti

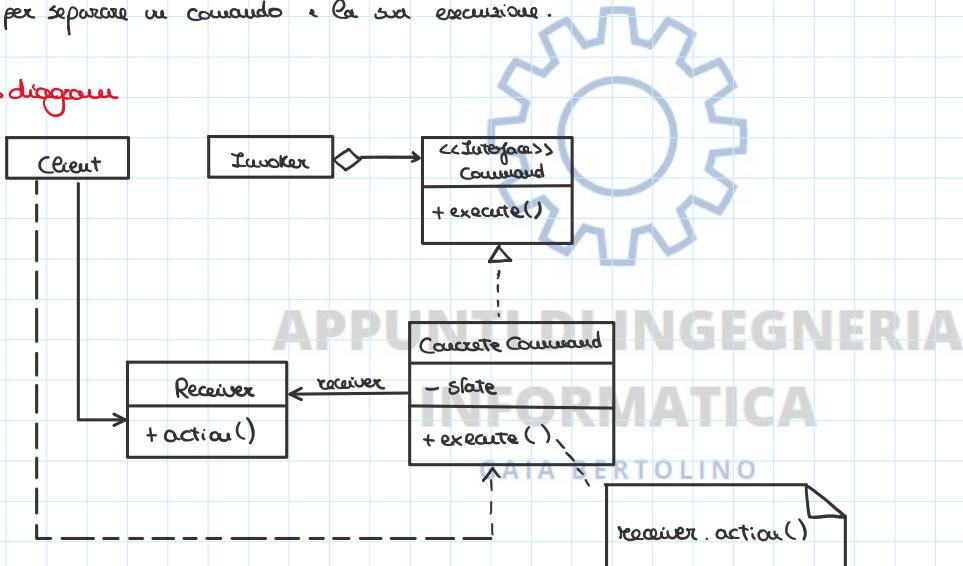
PRO: È possibile introdurre più observers senza alterare il subject
La comunicazione avviene in broadcast quindi è compito dell'observer decidere se ricepirlo o meno

CONTRO: I cambiamenti notificati耗費 tempo costato in alcun modo del costo di implementazione

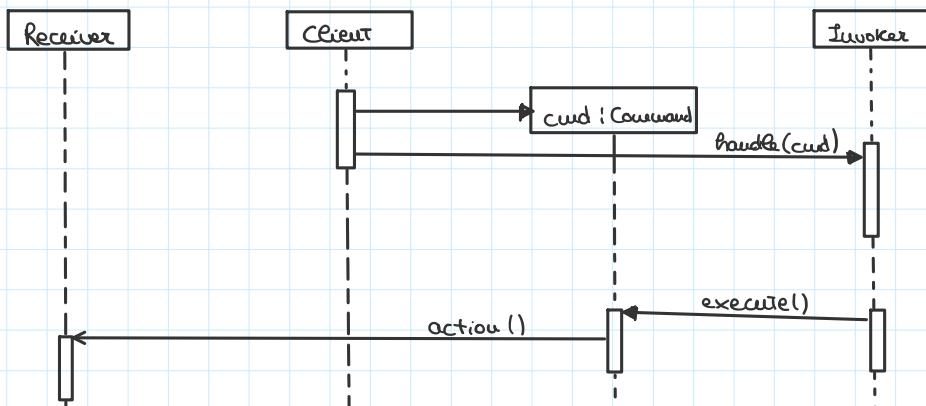
• Command (object-based)

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

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

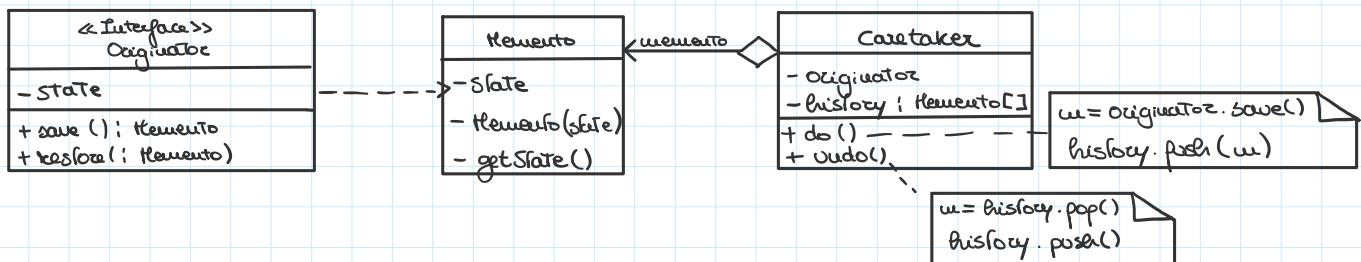
È possibile aggiungere nuovi comandi senza modificare le classi esistenti.

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

La memorizzazione può avvenire tramite Memento.

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

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

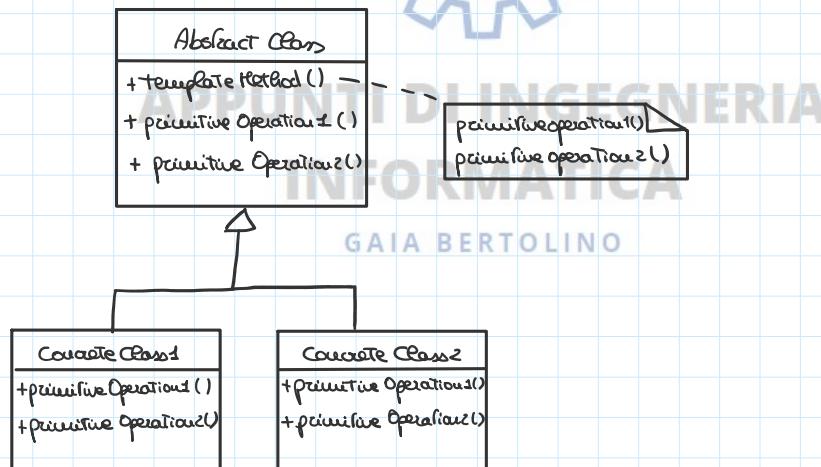


PRO: I dati rimangono incapsulati, protetti e accessibili solo all'Originator.

CONTRO: Un memento potrebbe essere oneroso da memorizzare a causa della quantità di dati al suo interno.

• **Template Method** (class-based)

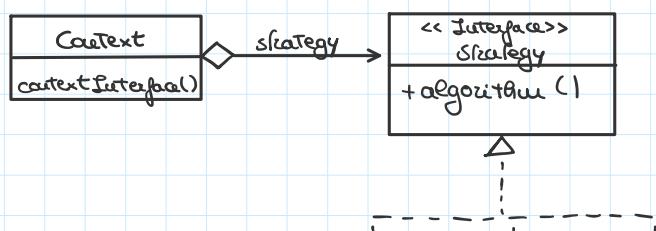
Usato per definire la struttura di un algoritmo o per riorganizzare delle classi spostando la parte comune in una classe astratta.

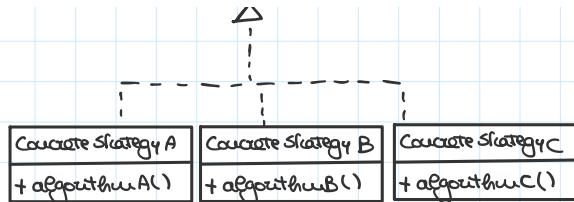


Il pattern ribalta la situazione classica attraverso il principio Hollywood ovvero è il genitore a invocare i metodi concreti delle sottoclassi.

• **Strategy** (object-based)

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

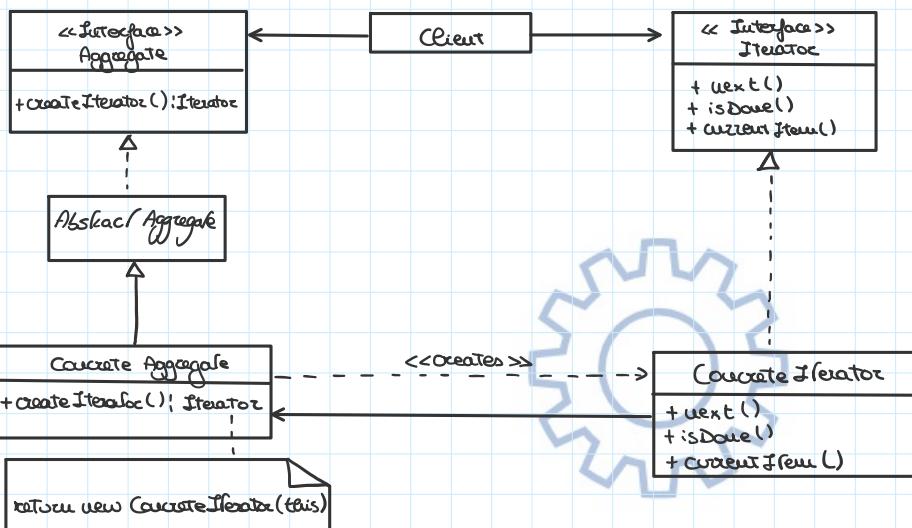




CONTRO: Si può avere overhead di comunicazione per quegli algoritmi più semplici

- **Iterator** (object-based)

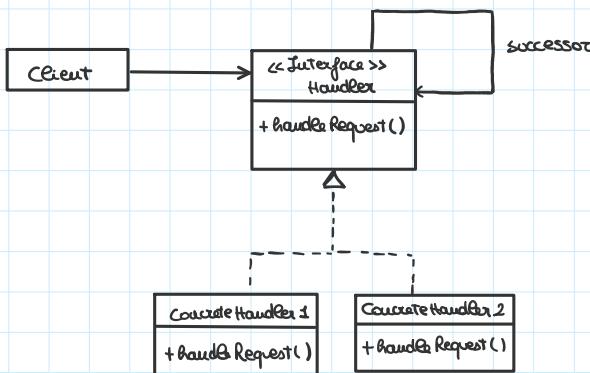
Usato per fornire un accesso iterativo e sequenziale agli elementi di una collezione senza esporre la struttura interna.



APPUNTI DI INGEGNERIA INFORMATICA

- **ChainOfResponsibility** (object-based)

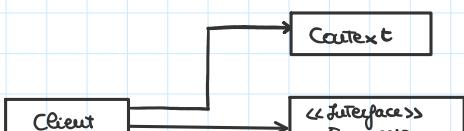
Usato per definire una catena di oggetti in grado di soddisfare una richiesta.
Il comando viene inviato da un oggetto all'altro

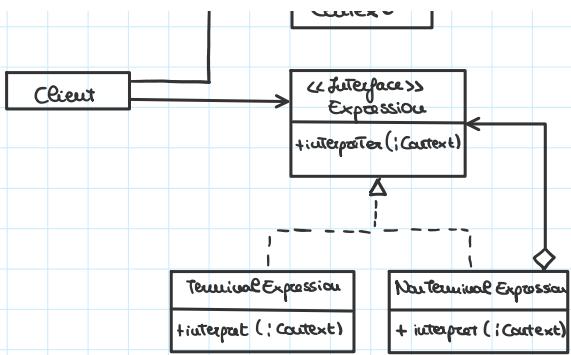


CONTRO: Il ricupero delle responsabilità non porta alla risoluzione della richiesta

- **Interpreter** (class-based)

Usato per definire una rappresentazione della grammatica di un linguaggio



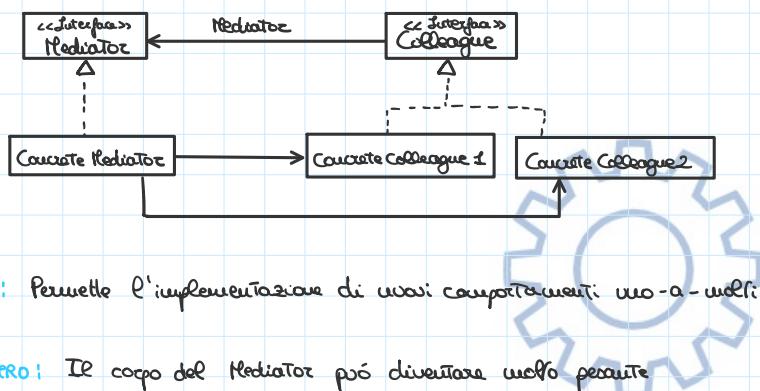


PRO: Facile implementazione

CONTRO: Non adatto a linguaggi molto 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

CONTRO: Il corpo del Mediator può diventare molto pesante

APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

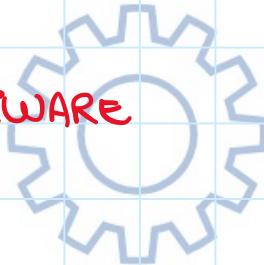
Concetti chiave

domenica 3 luglio 2022 00:04

FASI DI SVILUPPO DEL SOFTWARE

1. Studio di fattibilità
2. Analisi e specifica dei requisiti
3. Progettazione e specifica di sistema
4. Codifica e test di modulo
5. Integrazione e test di sistema
6. Conseguenza e manutenzione

QUALITÀ DEL SOFTWARE



1. Correttezza
2. Affidabilità
3. Robustezza
4. Verificabilità
5. Manutenibilità
6. Riparabilità
7. Evolvibilità
8. Prestazioni
9. Usabilità
10. Risabilità
11. Portabilità
12. Interoperabilità
13. Comprensibilità

APPUNTI DI INGEGNERIA
INFORMATICA

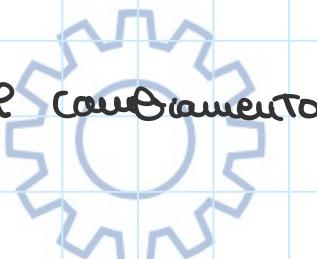
GAIA BERTOLINO

QUALITÀ DEI MODELLI DI PRODUZIONE

1. Risabilità
2. Produttività
3. Visibilità
4. Tempestività

PRINCIPI DELL' INGEGNERIA

1. Rigorosità
2. Separazione degli interessi
3. Modularità
4. Astrazione
5. Anticipazione del cambiamento
6. Generalità
7. Incrementalità

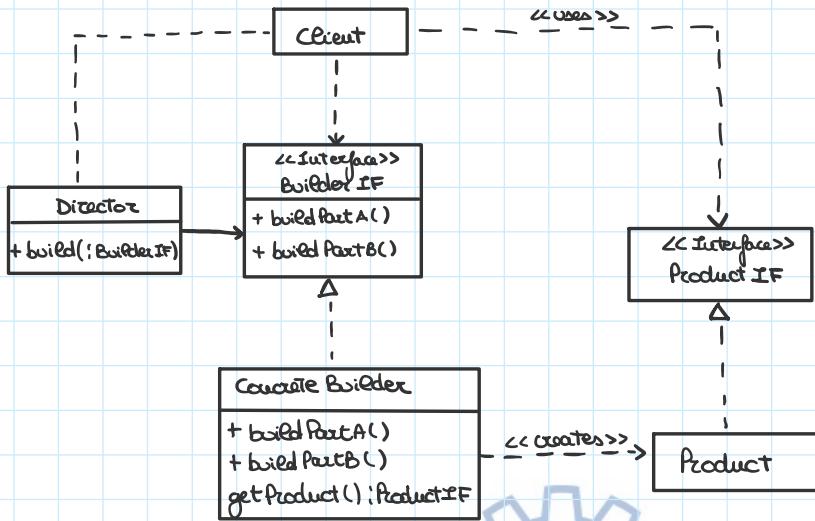


APPUNTI DI INGEGNERIA INFORMATICA

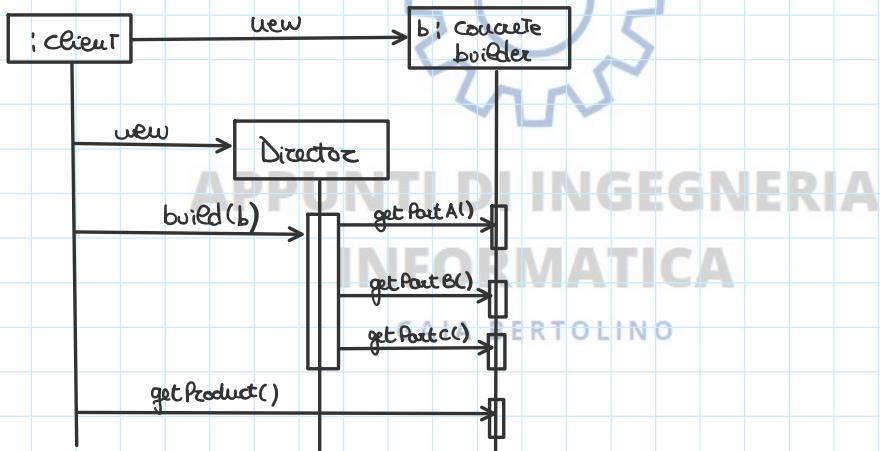
GAIA BERTOLINO

- **Builder (object-based)** * Director / Builder IF

Usato per definire e separare il processo di costruzione di un oggetto dalla realizzazione specifica dei passi di costruzione



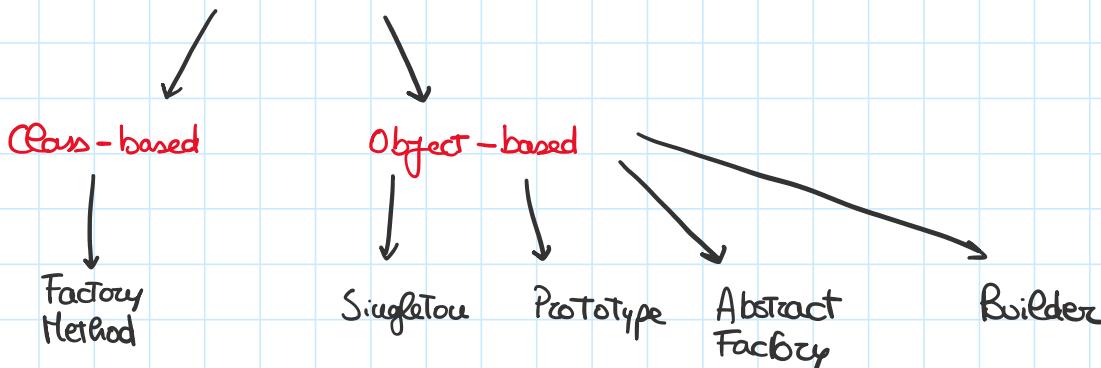
Sequence diagram



PRO: È una valida alternativa ai metodi costruttori telescopici e alla tecnica Java Beans che prevede la modifica dei campi attraverso dei metodi `setter` in quanto prevede un metodo per ogni parametro specificato e restituisce il `build` stesso.

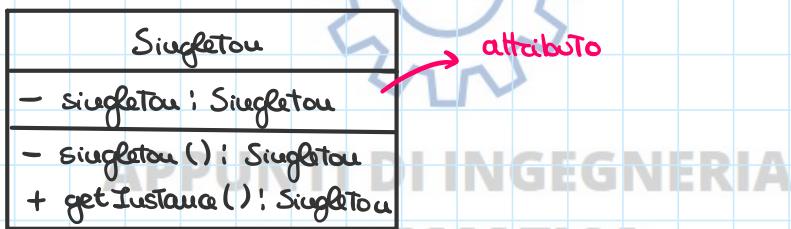
CREAZIONALI

PATTERN CREAZIONALI



- **Singleton** (object-based)

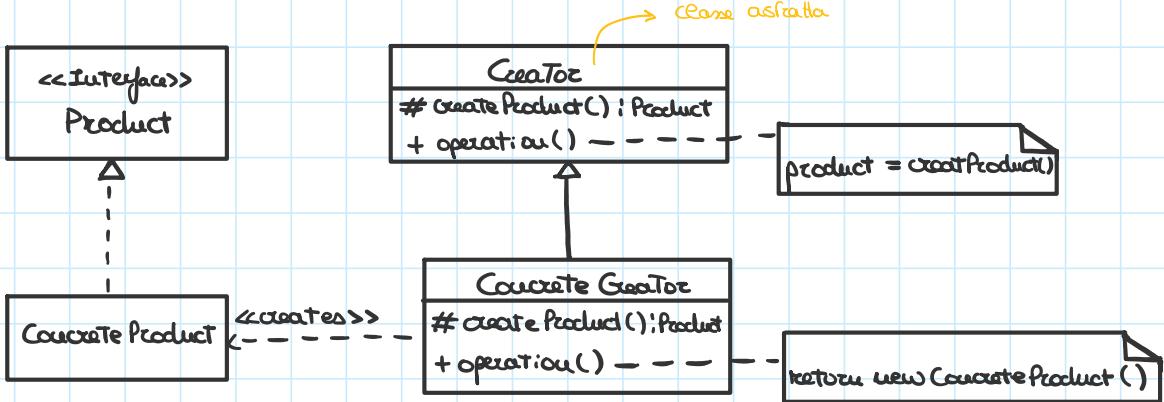
Usato per limitare la creazione per il tipo singleton ad un solo oggetto



CONTRO: bisogna realizzare la sincronizzazione per l'accesso e il metodo readResolve() se la classe è serializzabile per evitare copie.

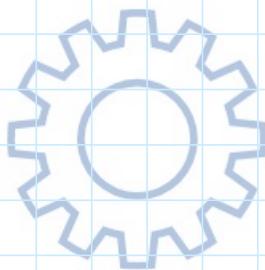
- **Factory method** (class-based) *creatore / prodotto

Usato per realizzare la creazione di un oggetto al di fuori della sua classe



es. I prodotti sono linee e poligoni (ovvero sottospecie di figure) e ciascuno di essi necessita di una classe concreta che implementi la realizzazione

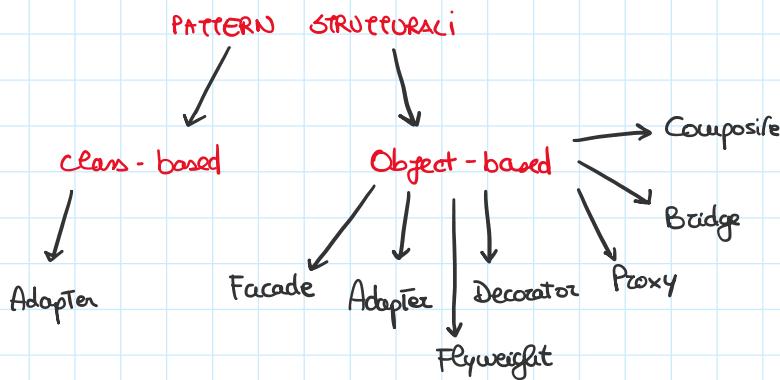
CONTRO: Per ciascun oggetto è necessario istanziare una classe concreta. Concrete Creator oltre che a creare una estensione della classe Product



APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

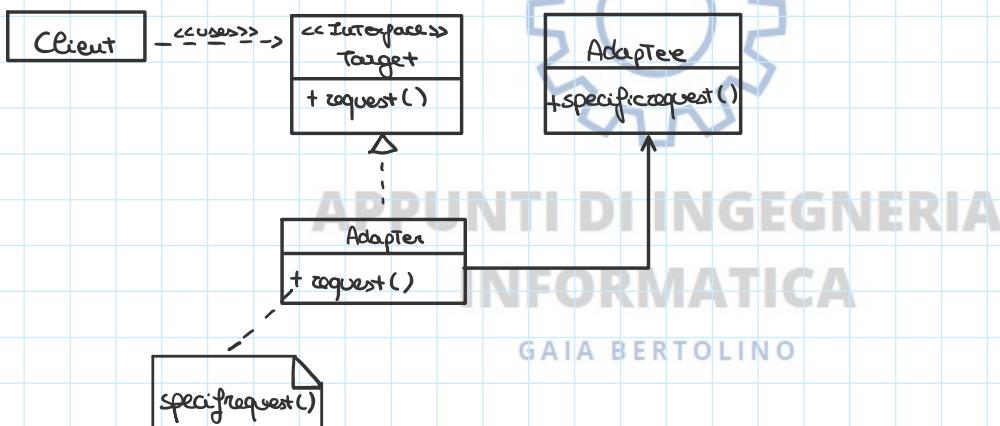
PATTERN STRUTTURALI



- **Adapter** (class-oriented / object-oriented) * Target

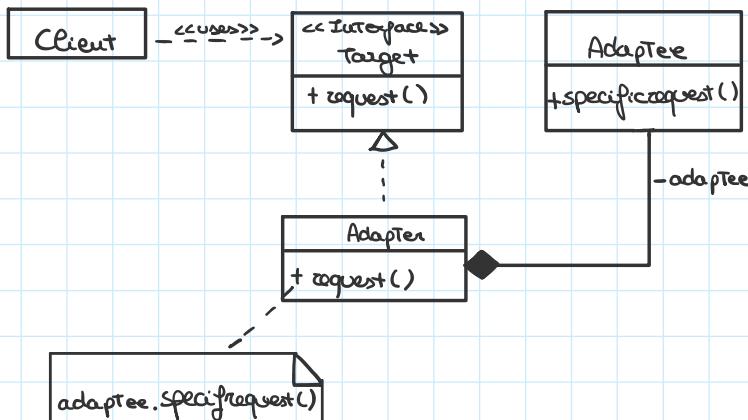
Usato per adattare una classe esistente ai meccanismi di una interfaccia conosciuta dal cliente

Versione class-based



CONTRO: ha limiti di applicabilità → Target è una classe astratta
o se Adaptee è final, interfaccia o astratta

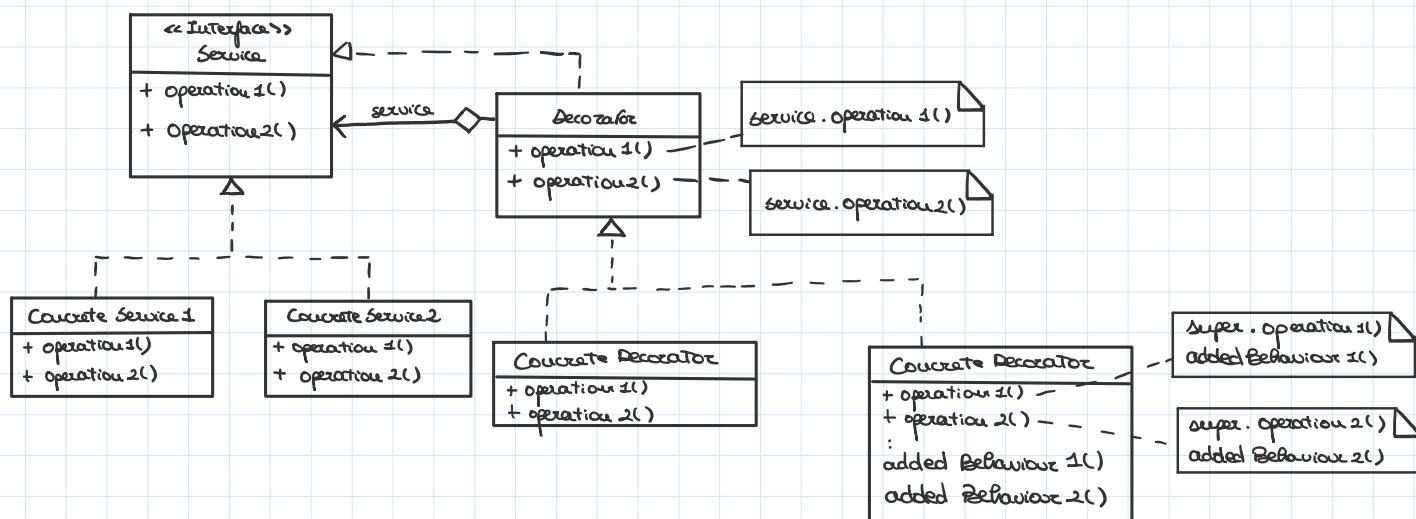
Versione object-based



L'eredità si compone a tempo
di compilazione mentre la
composizione è runtime.

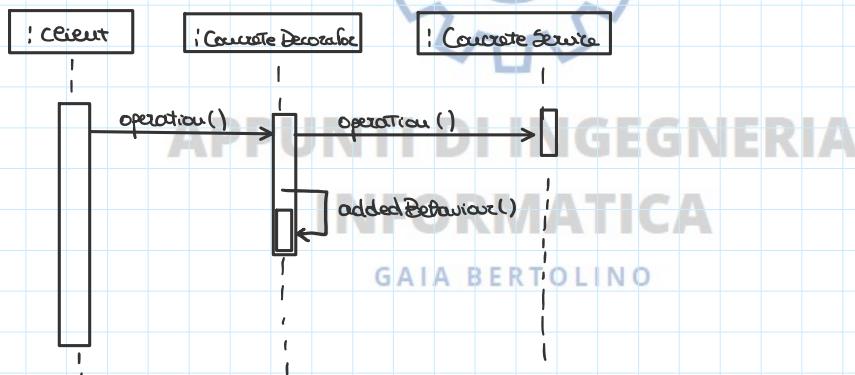
• **Decorator** (object-based) * Decorator / source

Usato per poter aggiungere in runtime delle caratteristiche ad un oggetto



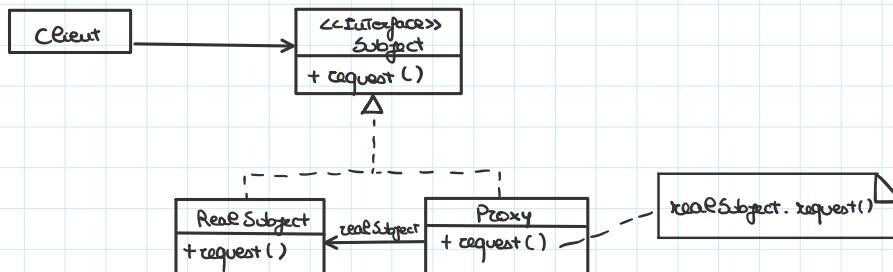
es. Nel framework di Java Swing, dei JComponent sono combinati secondo decorator come ad esempio JTextArea che viene decorato da JScrollPane

Sequence diagram



• **Proxy** (object-based) * Subject

Usato per introdurre una classe di filtro fra cliente e oggetto



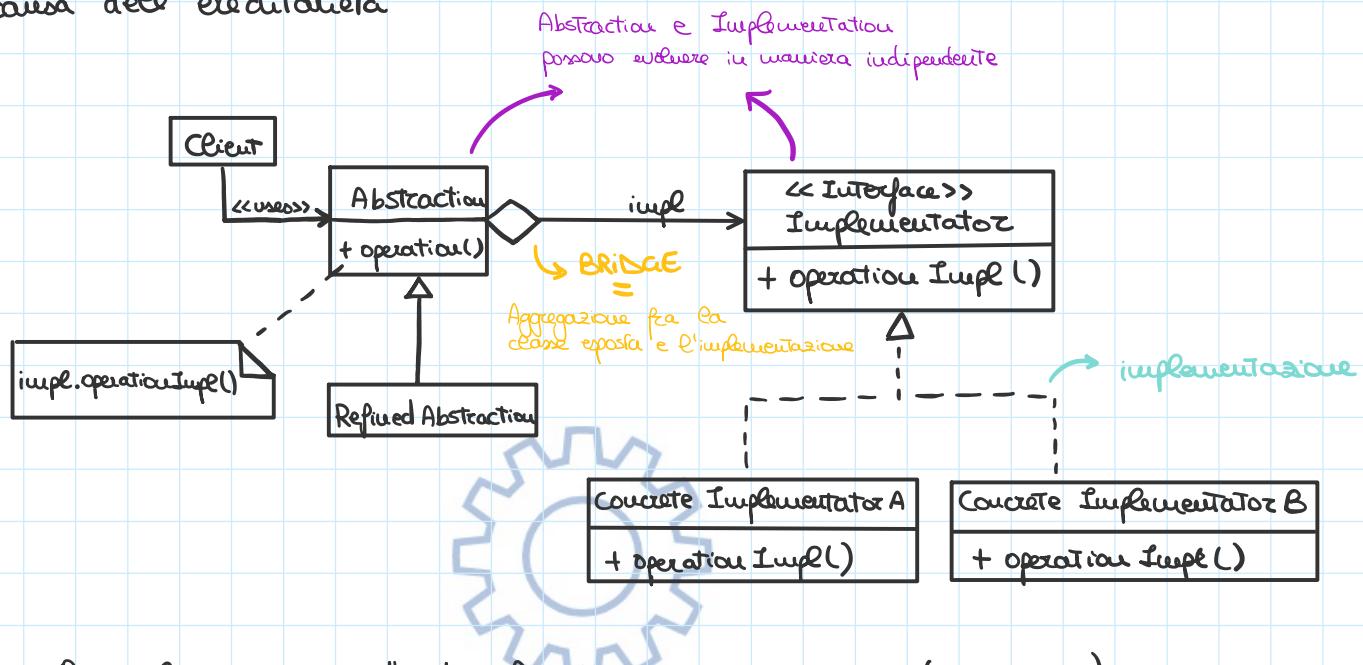
APPLICAZIONI:

- Proxy remoto
- Proxy virtuale
- Proxy di protezione
- Smart reference

- **Bridge** (object-based) * Abstraction / Implementation / operation Impl

Usato quando si vuole creare un disaccoppiamento fra un concetto e la sua implementazione.

Usare una classe astratta o una interfaccia vi è una forte dipendenza a causa dell'ereditarietà



es. Per realizzare i concetti di finestra e suoi sottotipi (gli eredi) e le possibili implementazioni in base all'ambiente grafico

INFORMATICA

GAIA BERTOLINO