

Design Pattern GOF

Si dividono in: (in rosso quelli che tratteremo)

- **CREAZIONALI** - risolvono problematiche inerenti l'istanziazione di oggetti
 - ABSTRACT FACTORY
 - SINGLETON
- **STRUTTURALI** - risolvono problematiche inerenti la struttura delle classi e degli oggetti
 - ADAPTER
 - COMPOSITE
 - DECORATOR
- **COMPORTAMENTALI** - forniscono soluzioni alle più comuni tipologie di interazione fra gli oggetti
 - OBSERVER
 - STATE
 - STRATEGY
 - VISITOR

I design pattern, a differenza dei pattern GRASP, sono degli schemi di progettazione, non principi

COMPOSIZIONE > EREDITARIETÀ

può essere reference via interface, o runtime
o può essere qualunque classe

rispetta l'incapsulamento, solo una modifica all'interfaccia può portare a ripercussioni

può essere **COMPOSIZIONE** via delega

- funzionalità ottenute assemblando o componendo gli oggetti
- i dettagli interni non sono conosciuti (viss white-box)

definita staticamente, non può cambiare a runtime

EREDITARIETÀ

- Definisce un oggetto in termini di un altro
- La sottoclass può accedere ai dettagli implementativi delle superclassse (viss white-box)

non rispetta l'incapsulamento, una modifica alla superclasse può rompere la sottoclasse

Il meccanismo di ereditarietà può essere utilizzato in due modi:

- Polimorfismo: sottoclassi eredate in base al tipo
- Specializzazioni: sottoclassi che guadagnano elementi e proprietà

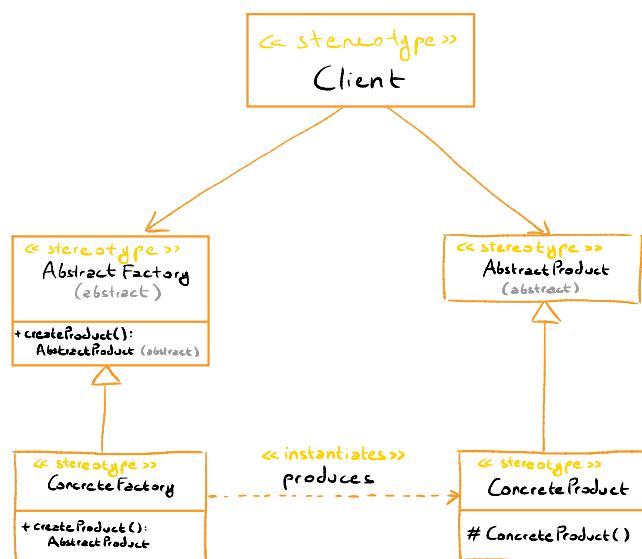
ABSTRACT FACTORY

Problema: Come creare famiglie di classi concrete che implementano un'interfaccia comune?

Soluzione: Definire un'interfaccia factory. Definire una classe factory concreta per ciascuna famiglia di elementi da creare. Optionalmente definire una vera classe astratta che implementa l'interfaccia factory e fornisce servizi comuni alle classi concrete che la estendono.

Presenta un'interfaccia per la creazione di famiglie di prodotti, in modo tale che chi le utilizza non abbia conoscenza delle loro concrete classi.

Una variante consiste nel creare una classe Factory comune alla quale si si accede tramite pattern Singleton.

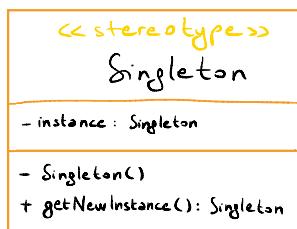


SINGLETON

Problema: È consentire una sola istanza di una classe.
Gli altri effetti hanno bisogno di un punto
d'accesso globale e unico è questo effetto

Soluzione: Definisci un metodo statico della classe che
restituisce l'effetto singleton

In UML un singleton viene illustrato con "1" nella sezione
del nome, in alto a destra



Implementabile in 3 modi in Java:

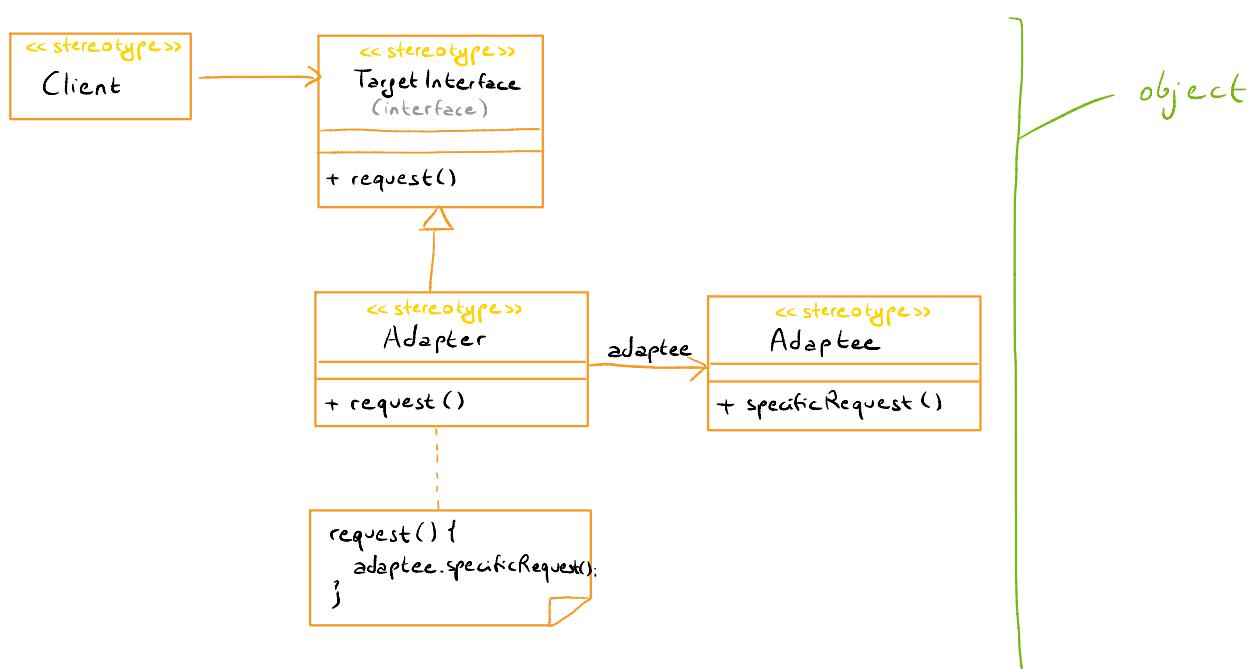
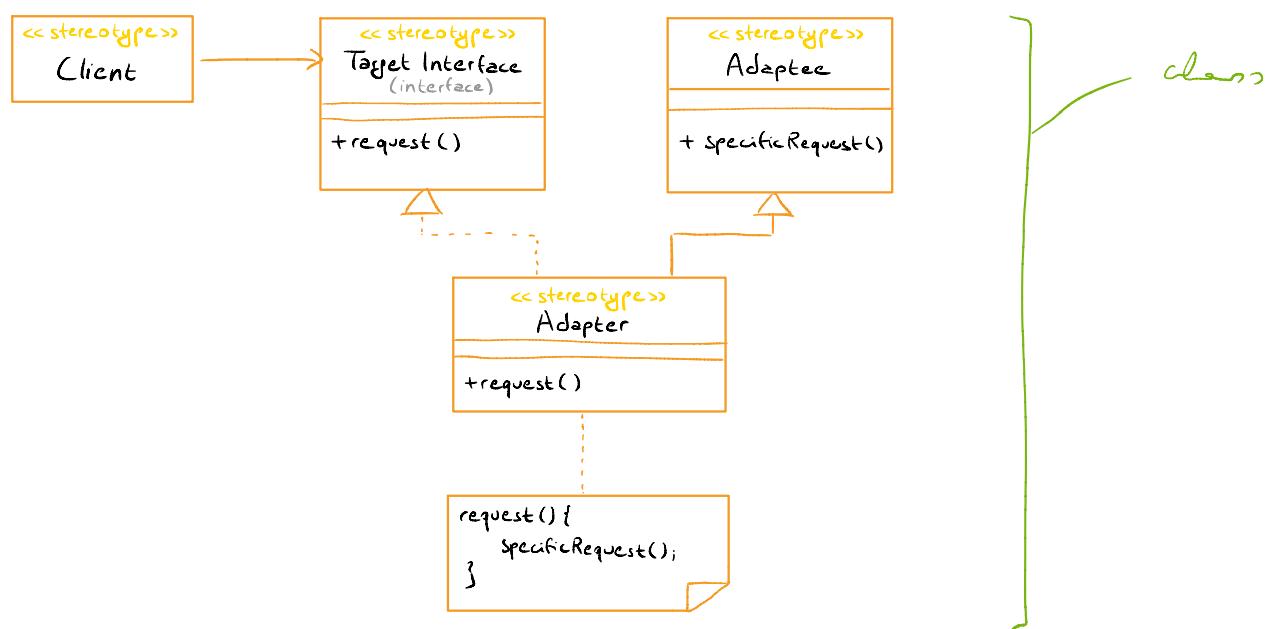
- Singleton come classe statica
- Singleton creato da un metodo statico l'oggetto viene
istanziato una volta sola, le successive chiamate
restituiscono lo stesso oggetto (inizializzazione pigrina)
- Singleton multi-thread stessa cosa di sopra ma multi-thread

preferibile

ADAPTER

Problema: Come gestire interface incompatibili → fornire un'interfaccia stabile a componenti simili ma con interface diverse?

Soluzione: Converti l'interface originale di un componente in un'altra interface, attraverso un oggetto adattatore intermedio.



COMPOSITE

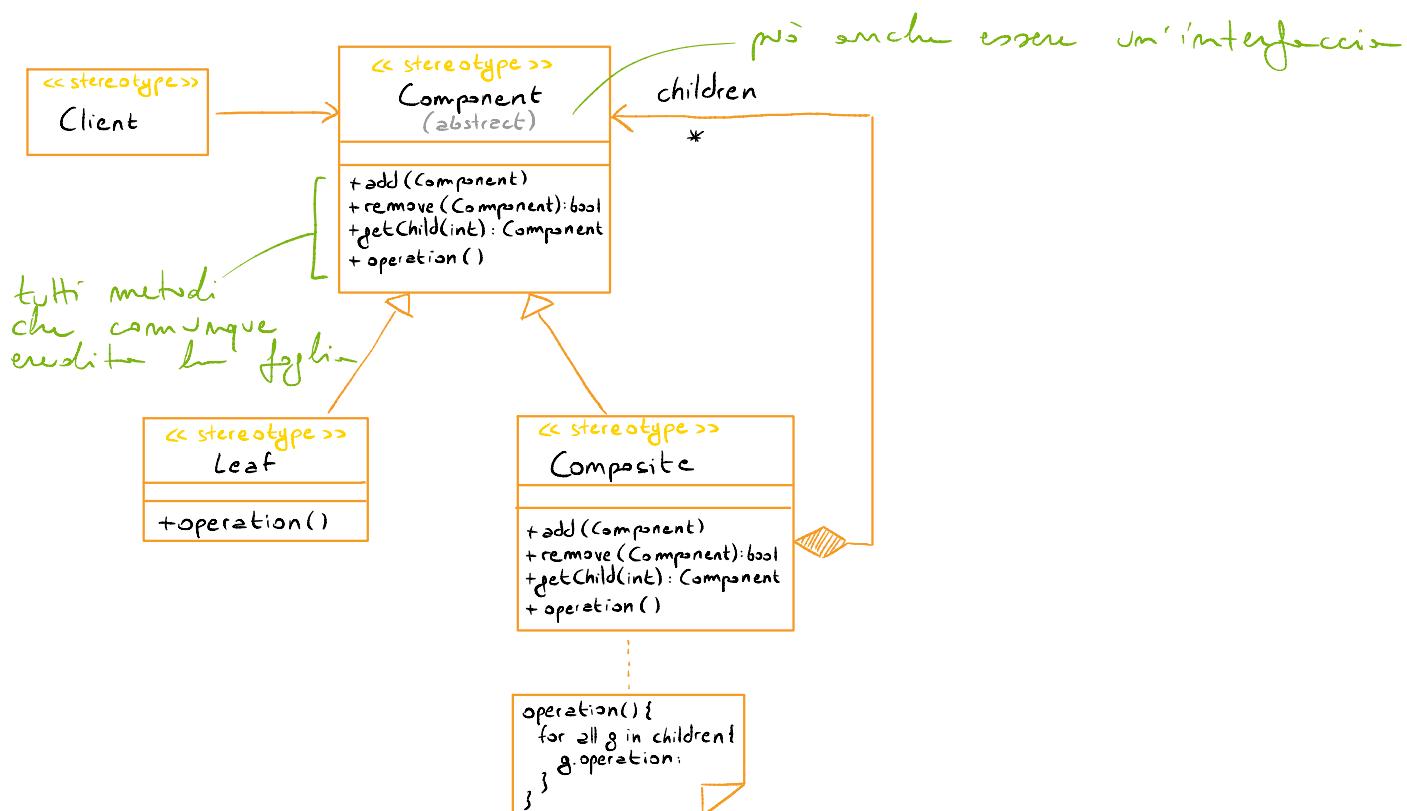
Problema: Come trattare un gruppo o una struttura composta di oggetti (polimorficamente) dello stesso tipo nello stesso modo di un oggetto non composto (atomico)?

Soluzione: Definisci le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia

Consente la costruzione di gerarchie di oggetti composti

Il pattern definisce una classe estratta componente che deve essere estesa in due sottoclassi:

- Una che rappresenta i singoli componenti (foglia)
- L'altra rappresenta i componenti composti e si implementa come un contenitore di componenti



Il pattern composite permette di costruire strutture ricorsive in modo che ad un client la struttura sia vista come singola entità

DECORATOR

noto anche come Wrapper
(si contrappone agli extends, è più flessibile)

Problema:

Come permettere gli assegnare una o più responsabilità aggiornabili ad un oggetto in maniera dinamica ed evitare il problema delle relazioni statiche? Come provvedere una alternativa più flessibile al meccanismo di sovraccarico ed evitare il problema di avere una gerarchia di classi complesse?

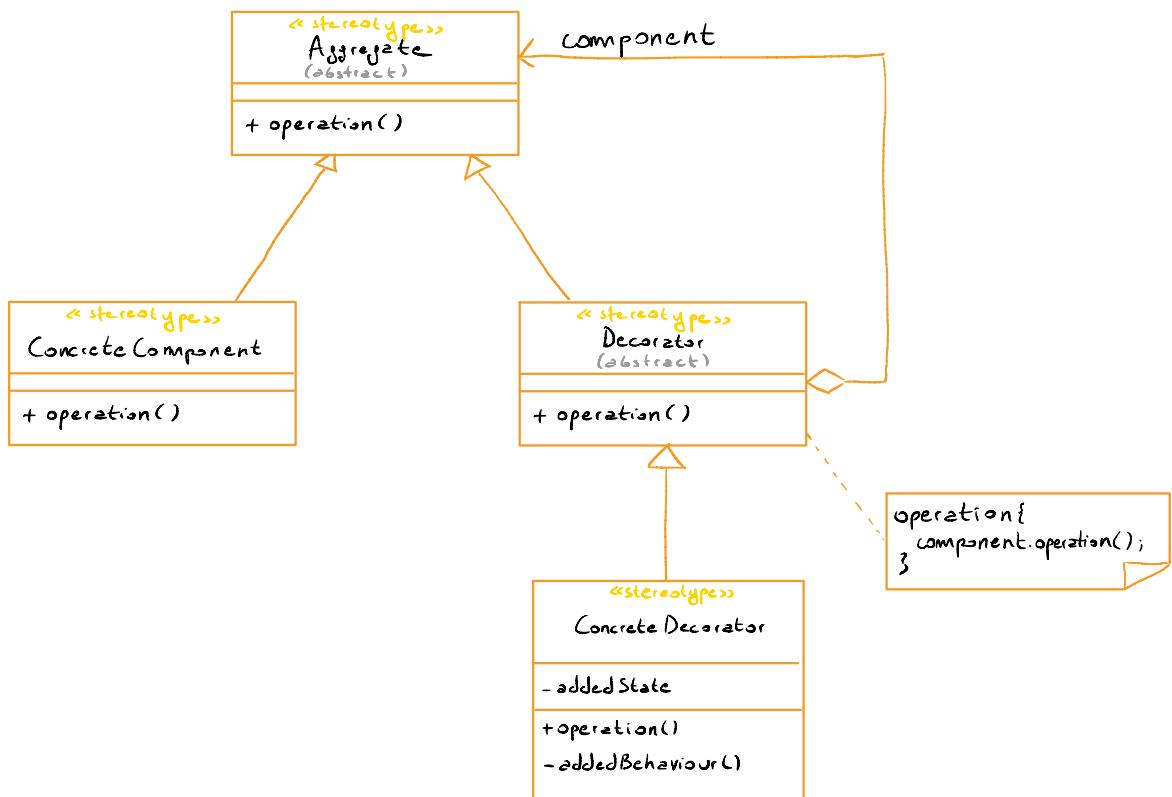
Soluzione:

Inglobare l'oggetto all'interno di un altro che aggiunge altre nuove funzionalità

Permette di affidare responsabilità ad oggetti individualmente e dinamicamente

Le responsabilità possono essere ritirate

Permette di evitare l'esplosione delle sottoclassi



OBSERVER

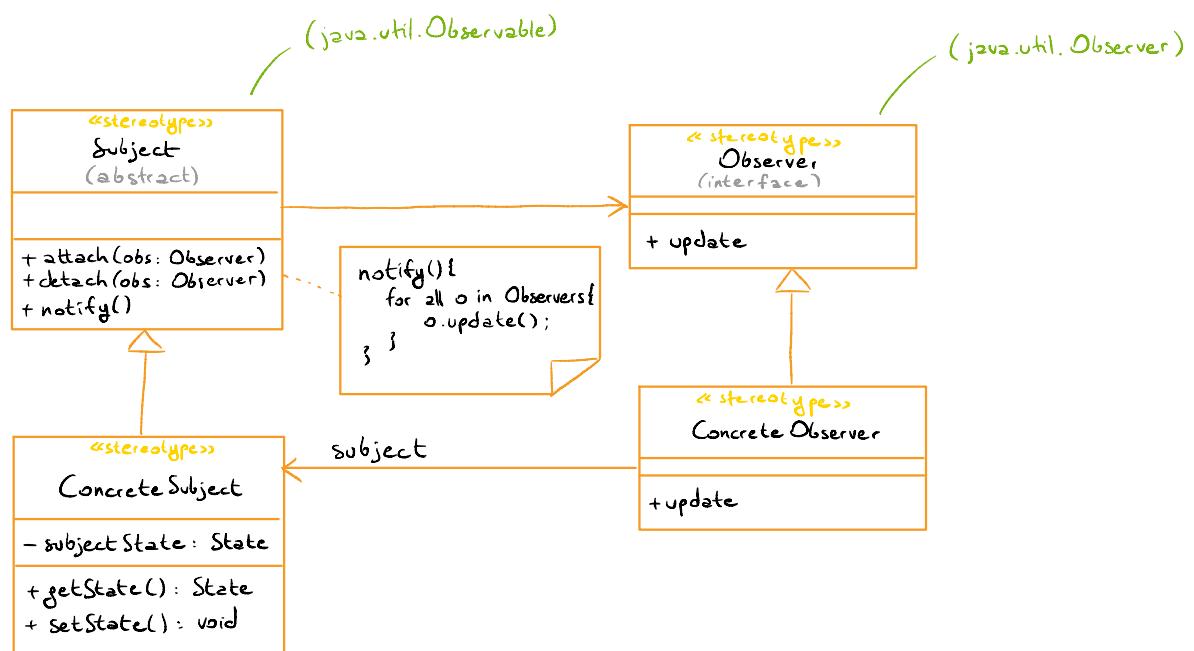
Problema: Diversi tipi di oggetti subscriber sono interessati ai cambiamenti di stato o agli eventi di un oggetto publisher. Ciascun subscriber vuole reagire in un modo proprio quando il publisher genera un evento. Inoltre il publisher vuole mantenere un accoppiamento basso verso i subscriber.

Soluzione: Definisci un'interfaccia "subscriber" o "listener". Gli oggetti subscriber implementano questa interfaccia. Il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi e li avvia quando si verificano eventi.

Definisce una dipendenza uno a molti

Il numero di oggetti effetti dal cambiamento di stato non è noto a priori

Spesso associato al pattern architettonico Model-View-Controller

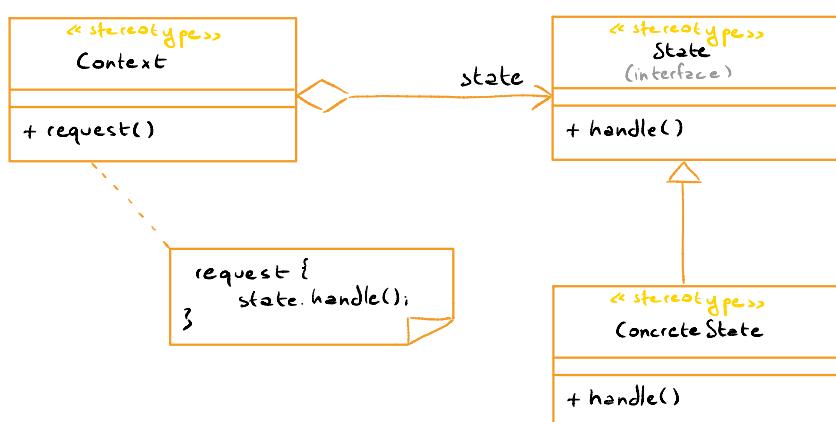


STATE

Problema: Il comportamento di un oggetto dipende da un suo stato e i suoi metodi contengono logiche condizionali per casi che riflette le azioni condizionali che dipendono dal suo stato.

Soluzione: Crea delle classi per ciascun stato che implementano un'interfaccia comune. Delega le operazioni che dipendono dallo stato dell'oggetto contesto all'oggetto stato corrispondente. Assicura che l'oggetto contesto referenti sempre un oggetto stato che riflette il suo stato corrente.

Può sembrare che un oggetto modifichi la sua classe



STRATEGY

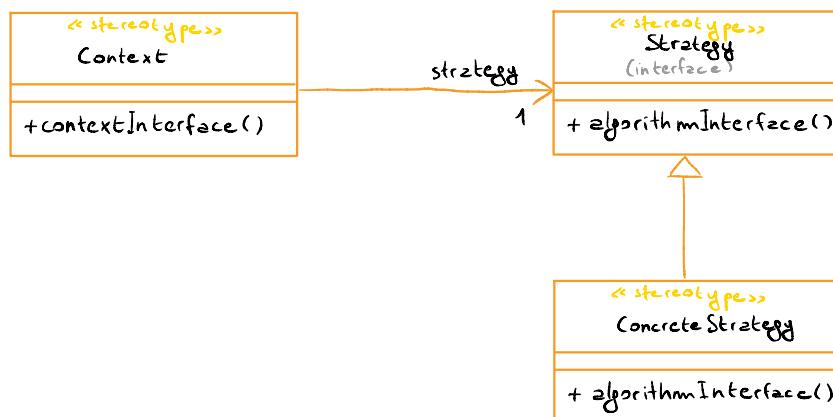
notes anche come Policy

Problema:

Come progettare per gestire un insieme di algoritmi o politiche variabili ma correlati?
Come progettare per consentire di modificare questi algoritmi o politiche?

Soluzione:

Definisci ciascun algoritmo/politica/strategy in una classe separata, con un'interfaccia comune



VISITOR

Problema: Come separare l'operazione applicata su un contenitore complesso dalle strutture dati cui è applicata? Come poter aggiungere nuove operazioni e comportamenti senza modificare la struttura stessa? Come attraversare il contenitore complesso i cui elementi sono diversi e applicare azioni dipendenti dal tipo di elementi?

Soluzione: Creare un oggetto (ConcreteVisitor) che è in grado di percorrere la collezione e di applicare un metodo proprio a ogni oggetto (Element) visitato nella collezione (avendo un riferimento a questi ultimi come parametro). Ogni oggetto della collezione aderisce a un'interfaccia (Visitable) che consente al ConcreteVisitor di essere accettato da parte di ogni Element. Il Visitor analizza il tipo di oggetto ricevuto, fa l'invocazione alla particolare operazione che deve eseguire.

- Flessibilità delle operazioni
- Organizzazione logica
- Visita di diversi tipi di classi
- Mantenimento di uno stato efforziabile ad ogni visita
- Le diverse modalità delle strutture possono essere definite come sottoclassi nel Visitor

