



UNIVERSITÀ DI CATANIA
Dipartimento di Matematica e Informatica



APPUNTI LEZIONI

Ingegneria Del Software

Sergio Mancini

Anno Accademico 2021-2022

Design Pattern

SINGLETON - CREAZIONALI

Intento:

Assicura che una classe abbia una sola istanza, e fornisce un punto di accesso globale ad essa

Problema:

- Questo DP risolve due problemi allo stesso momento, quindi viola il *Principio di singola responsabilità*

Assicurarsi che una classe abbia una sola istanza.

- Dovremmo controllare quante istanze abbia una classe - per tenere il controllo di accesso alle risorse condivise (DB..)

Fornire un punto di accesso globale all'istanza.

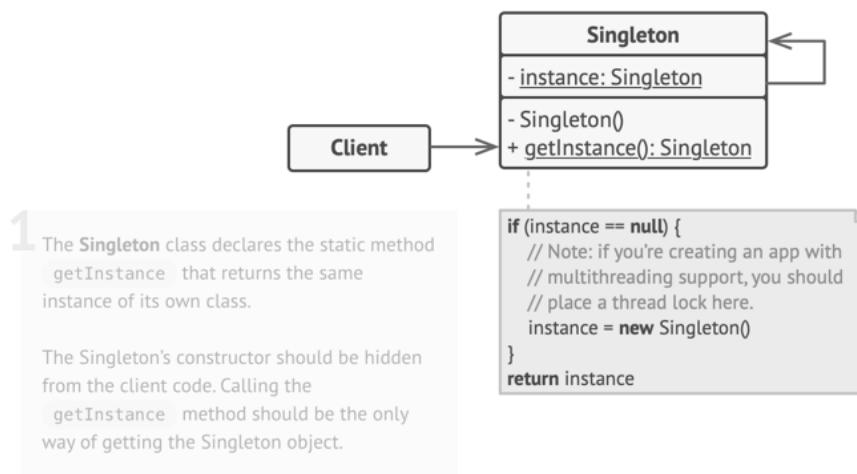
- Le variabili globali sono facili da usare ma non sicure perché il codice può essere sovrascritto cambiando il loro valore. Il Singleton ti permette di accedere agli oggetti da qualunque parte del programma

Soluzione:

- Costruttore privato, per prevenire la creazione di nuovi oggetti con `new`
- Metodo `getInstance()` che ritorna l'istanza stessa

Applicabilità:

- Usa questo DP quando si vuole che una classe abbia una sola istanza disponibile a tutti i client (singolo DB)
- Quando vuoi avere un controllo stretto delle variabili globali



FACTORY METHOD - CREAZIONALI

Intento:

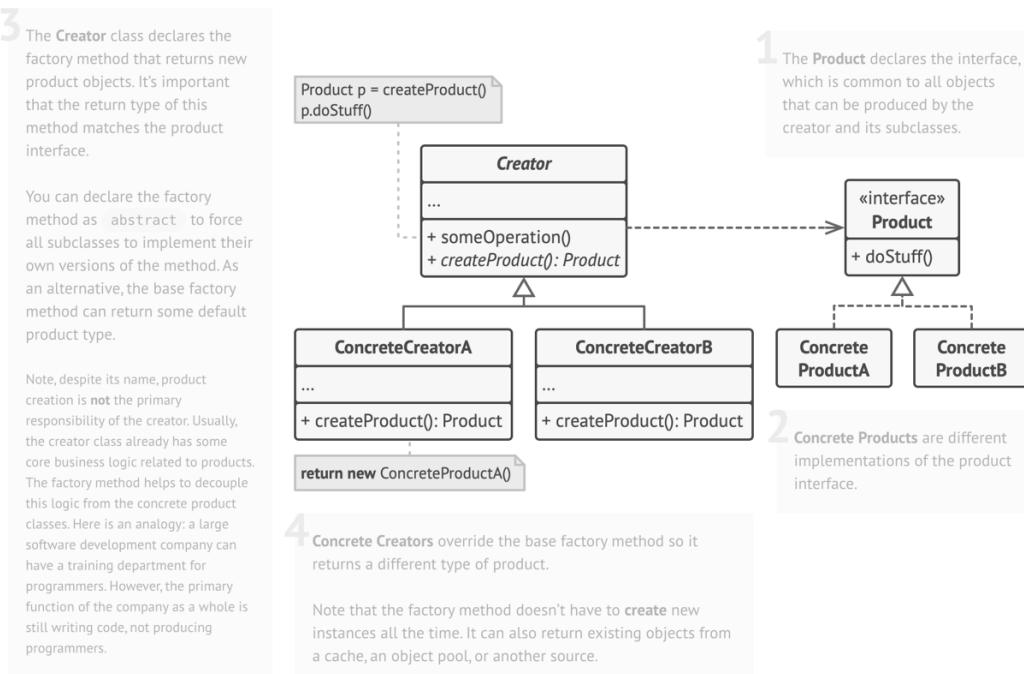
Fornisce un'interfaccia per creare oggetti nella superclasse, ma lascia alle sottoclassi la possibilità di cambiare il tipo dell'oggetto creato

Problema:

- Se si sta creando un'applicazione di trasporti, la prima versione potrebbe essere quella di trasporti via terra, quindi con una classe *truck*, ma se un giorno si volesse aggiungere anche la possibilità di altri trasporti, bisogna cambiare tutto il codice

Soluzione:

- Questo DP suggerisce di rimpiazzare le chiamate dirette per la costruzione di oggetti, con chiamate al *factory method*
- Gli oggetti saranno comunque creati con *new* ma dentro il *factory method*
- L'oggetto ritornato farà riferimento a un *Products*



Applicabilità:

- Usa questo DP quando non conosci prima il tipo esatto e le dipendenze dell'oggetto
- Quando vuoi fornire a utenti un modo per estendere i loro componenti interni

ABSTRACT FACTORY – CREAZIONALI

Intento:

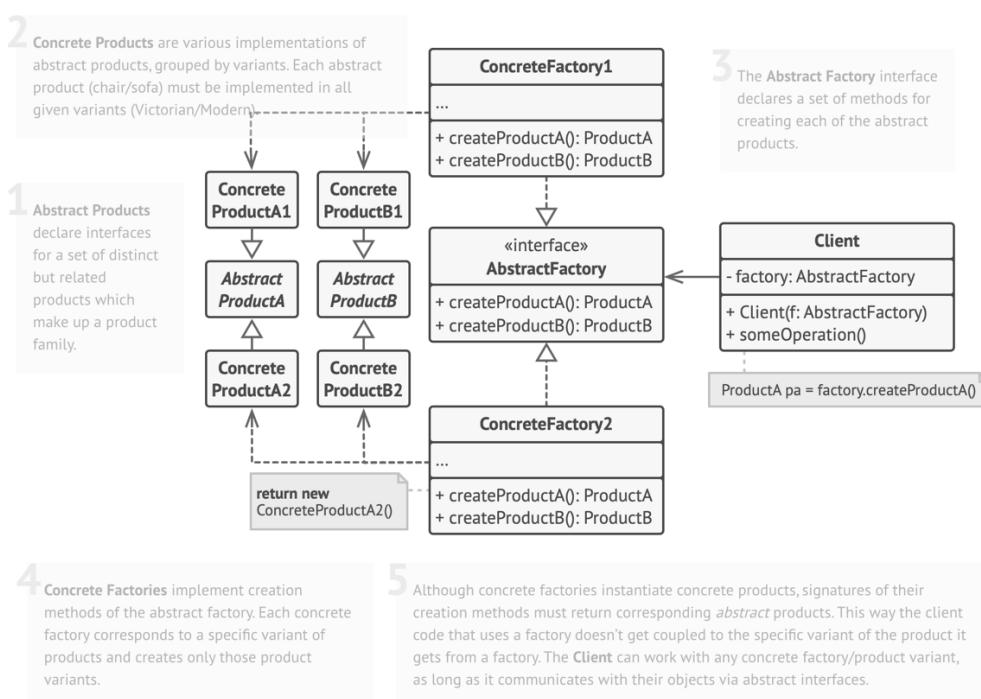
Permette di produrre una famiglia di oggetti relazionati senza specificare la loro classe concreta

Problema:

- Si vuole creare uno shop, una famiglia relazionata di oggetti (sedia, divano, tavolo), e ci sono varianti di questi oggetti (moderni, antichi)
- Serve un modo per creare un oggetto individuale della fornitura così possa corrispondere con gli altri oggetti della stessa famiglia
- Non si vuole cambiare il codice ogni volta che si aggiunge una nuova variante

Soluzione:

- Interfaccia *abstract product* per ogni tipo di prodotto. Ogni classe *concrete product* implementa questa interfaccia
- Interfaccia *abstract factory* con dei metodi per creare gli *abstract products*
- Un set di *concrete factory* per ogni variante di prodotti



Applicabilità:

- Usa questo DP quando il codice ha bisogno di lavorare con tante famiglie di prodotti relazionati, ma non vuoi farli dipendere dalle classi concrete

PROTOTYPE – CREAZIONALI

Intento:

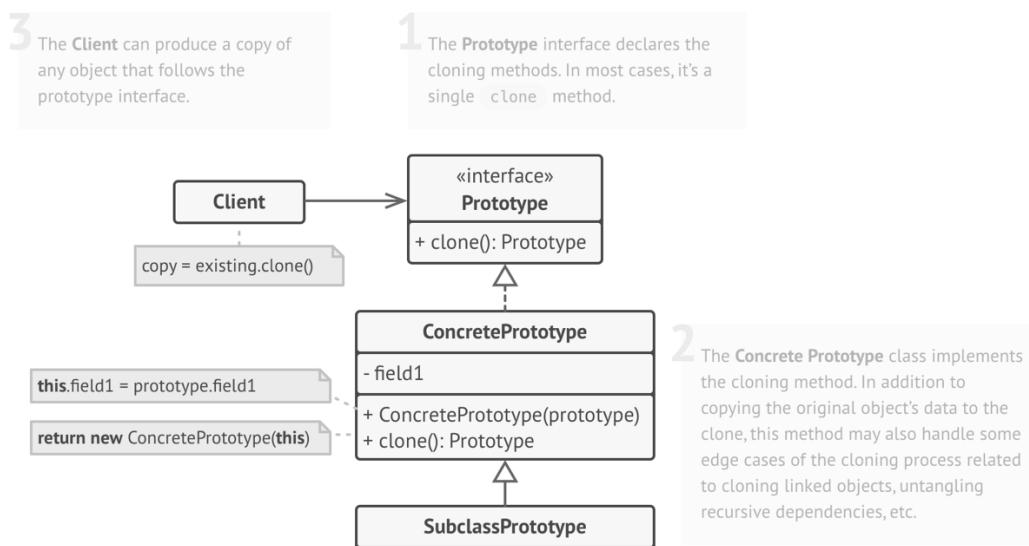
Permette di copiare oggetti esistenti senza far dipendere il codice con le altre classi. Clona

Problema:

- Avendo un oggetto, se si vuole fare la copia di questo, si potrebbe creare un altro oggetto dalla stessa classe, poi bisogna prendere tutti i campi dell'oggetto originale e copiarli nel nuovo
- Ma non tutti gli oggetti possono essere copiati, per esempio possono avere campi privati
Se si copiasse così, l'oggetto copiato dipenderebbe da quella classe

Soluzione:

- Questo DP delega il processo di clonazione all'attuale oggetto che deve essere clonato
- Dichiara un'interfaccia che permette di copiare l'oggetto senza far dipendere il codice con la classe di questo oggetto. Usa il metodo *clone*
- Questo metodo crea un oggetto della classe corrente e porta tutti i campi del vecchio oggetto, nel nuovo, anche quelli privati



Applicabilità:

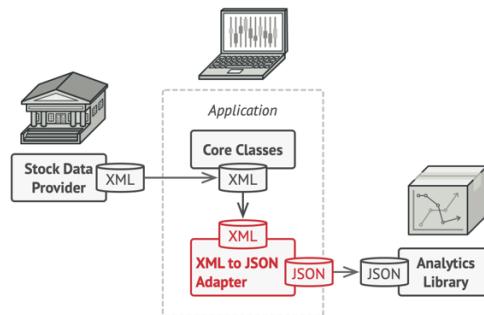
- Usa questo DP quando il codice non deve dipendere con la classe concreta dell'oggetto che si deve copiare
- Oppure, quando si vuole ridurre il numero di sottoclassi che differiscono solo nel modo in cui vengono inizializzati i rispettivi oggetti

ADAPTER – STRUTTURALI

Intento:

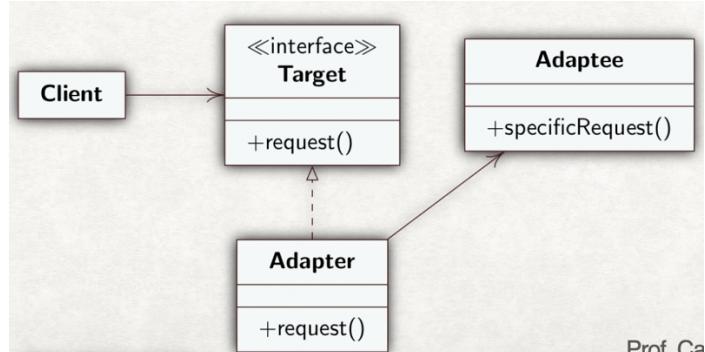
Questo DP permette a oggetti con interfacce incompatibili, di poter collaborare. Rende interfacce incompatibili, compatibili

Problema:



Soluzione:

- Si crea un *adapter*, quest'oggetto speciale converte l'interfaccia di un oggetto così che un altro possa capirla
- L'interfaccia *Target* è quella che si aspetta il client
- *Client* usa oggetti conformi all'interfaccia
- *Adaptee* è l'oggetto che ha bisogno dell'adattamento
- L'*Adapter* converte la chiamata che fa un *Client* all'interfaccia di *Adaptee*. Il chiamante usa *Target* come fosse l'oggetto di libreria
- L'*Adapter* implementa *Target*, tiene il riferimento di *Adaptee* e sa quando invocarlo



Applicabilità:

- Usa questo DP quando vuoi usare una classe esistente, ma la sua interfaccia non è compatibile col codice

DECORATOR – STRUTTURALI

Intento:

Permette di attaccare nuovi comportamenti a oggetti, piazzando questi oggetti dentro *oggetti wrapper speciali* che contengono questi comportamenti

Problema:

- Con questo pattern, se voglio aggiungere la proprietà Bordo al componente Testo, dovrò inserire Testo dentro un altro oggetto che a sua volta aggiunge Bordo
- Se si usa l'*ereditarietà*, ereditando Testo da Bordo, non si avrà flessibilità, perché non si possono avere bordi attorno a testi
- L'*ereditarietà* è statica, non si può alterare il comportamento di un oggetto a run-time. Si può solo rimpiazzare questo oggetto, con un altro che è stato creato da una sottoclasse
- L'*ereditarietà* non permette di ereditare comportamenti di diverse classi allo stesso momento

Soluzione:

- Usare l'*aggregazione* al posto dell'*ereditarietà*
- *Component* è l'interfaccia comune per gli oggetti che si possono “wrappare”; contiene l'operazione che li lega semanticamente
- *ConcreteComponent* è l'oggetto base cui poter aggiungere responsabilità
- *Decorator* è un Component con all'interno una referenza a un altro Component, inoltre le richieste di servizio al suo Component interno ed esegue l'operazione prima/dopo l'inoltro della richiesta
- *ConcreteDecorator* implementa una responsabilità aggiuntiva per il Component

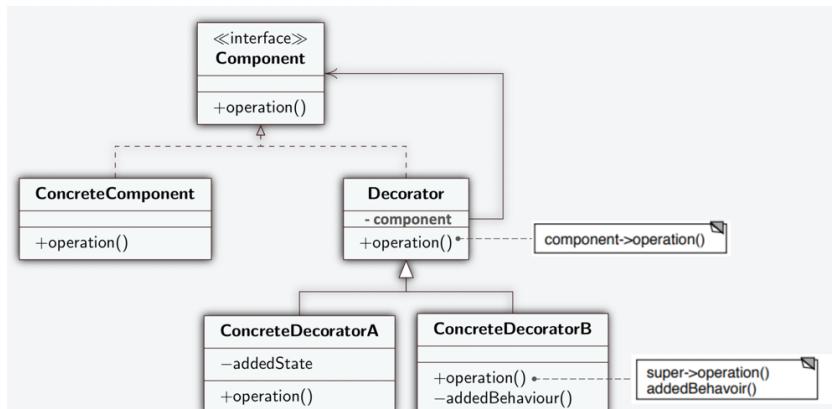
```
interface Component {
    public void operation();
}

class ConcreteComponent implements Component {
    @Override public void operation() {
        // ...
    }
}

class Decorator implements Component {
    private final Component innerC;
    public Decorator(Component c) {
        innerC = c;
    }
    @Override public void operation() {
        innerC.operation();
    }
}

class ConcreteDecoratorA extends Decorator {
    public ConcreteDecoratorA(Component c) {
        super(c);
    }
    @Override public void operation() {
        super.operation();
        // ...funzione aggiuntiva
    }
}

Component c = new ConcreteDecoratorA(new ConcreteComponent());
```



Applicabilità:

- Usa questo DP quando si vuole avere la possibilità di aggiungere nuovi comportamenti a oggetti a run-time senza rompere il codice che usa questi oggetti
- Oppure quando è difficile o impossibile aggiungere nuovi comportamenti a un oggetto tramite *ereditarietà*

FACADE – STRUTTURALI

Intento:

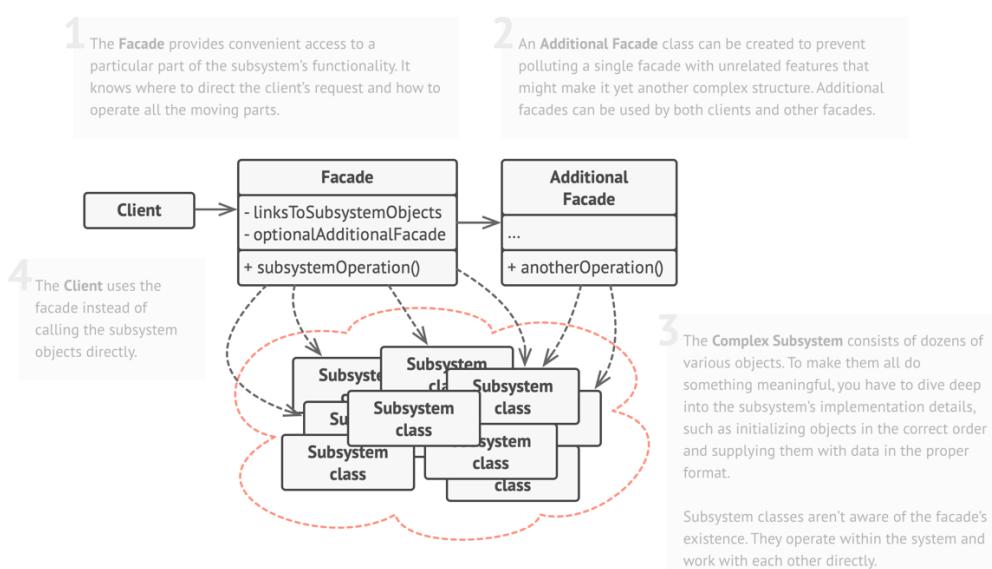
Fornire un'interfaccia semplificata, per un insieme complesso di classi

Problema:

- Se si vuole far lavorare il codice con un insieme di oggetti che appartengono a una sofisticata libreria o framework, bisognerebbe inizializzare tutti questi oggetti, tenere traccia delle dipendenze ed eseguire metodi
- Come risultato si avrà una logica complicata e se si volessero aggiungere nuove classi, diventa difficile da mantenere

Soluzione:

- Un *facade* è una classe che fornisce un'interfaccia semplice per un sottosistema complesso
- Fornisce funzionalità limitate rispetto a lavorare direttamente con il sottosistema, include solo le funzionalità principali che servono al client



Applicabilità:

- Usa questo DP quando hai bisogno di un numero limitato e finito di interfacce per un sottosistema complesso
- Oppure vuoi una struttura a livelli per il sottosistema

COMPOSITE – STRUTTURALI

Intento:

Permette di inserire oggetti dentro una struttura ad *albero* (strutture ad oggetti *composite*, cioè che contengono relazioni *tutto-parte*, quindi gerarchie) e poi lavorare con queste strutture come se fossero oggetti individuali. Permette al client di trattare oggetti singoli e composizione di oggetti singolarmente

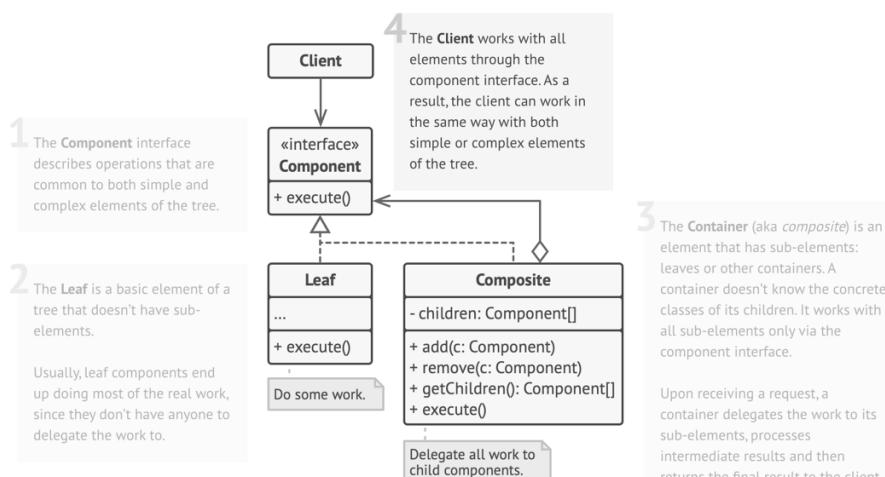
Problema:

- Esempio: Due tipi di oggetti, *Products* e *Boxes*. Una *Box* può contenere molti *Products* sicuramente di un numero minore della grandezza della scatola. Queste *Boxes* possono contenere *Products* o anche altre piccole *Boxes*.
- Creando un'applicazione di ordini che usa queste due classi; Gli ordini possono contenere prodotti e scatole fino a quando non si riempie. Come si determina il prezzo?

L'approccio diretto sarebbe di scartare tutti i prodotti e sommare i prezzi, fattibile nella realtà ma non in un programma perché bisognerebbe conoscere la classe *Products* e *Boxes*, ma ci possono essere molti livelli di nidificazione quindi difficile o impossibile

Soluzione:

- Questo DP suggerisce di lavorare con queste due classi tramite un'interfaccia comune che dichiara un metodo per calcolare il prezzo
- Per un *Product* ritorna il prezzo, invece per la *Boxe* va dentro ogni prodotto e chiede qual è il prezzo e dopo ritorna il totale, se contiene una scatola ci entra e continua fino alla fine di tutti i prodotti/scatole
- Non bisogna preoccuparsi delle classi concrete degli oggetti che formano l'albero, non importa sapere se è una scatola o un prodotto, ma si possono trattare tutti nello stesso modo tramite l'interfaccia. Quando si richiama un metodo, l'oggetto stesso passa la richiesta ai nodi sottostanti
- *Component* è l'interfaccia che rappresenta uniformemente i vari elementi; Dichiara le operazioni comuni applicabili a tutti gli oggetti *semplici*. Può avere un metodo che permette ad un elemento di accedere al padre nella struttura
- *Leaf* è la classe che rappresenta gli elementi *semplici*; Sottoclasse di *Component* e implementa il comportamento degli oggetti *semplici* facendo override dei comportamenti di default
- *Composite* è sottoclasse di *Component* e rappresenta elementi del contenitore
Mantiene un riferimento a dei *child* (leaf o composite); Implementa operazioni per gestire i *child* (add,remove...); Implementa operazioni dichiarate in *Component* applicandole ai *child* con ricorsione



Applicabilità:

- Usa questo DP quando devi implementare una struttura di oggetti tipo albero
- Oppure quando il client deve trattare oggetti semplici e complessi, uniformemente (interfaccia)

BRIDGE – STRUTTURALI

Intento:

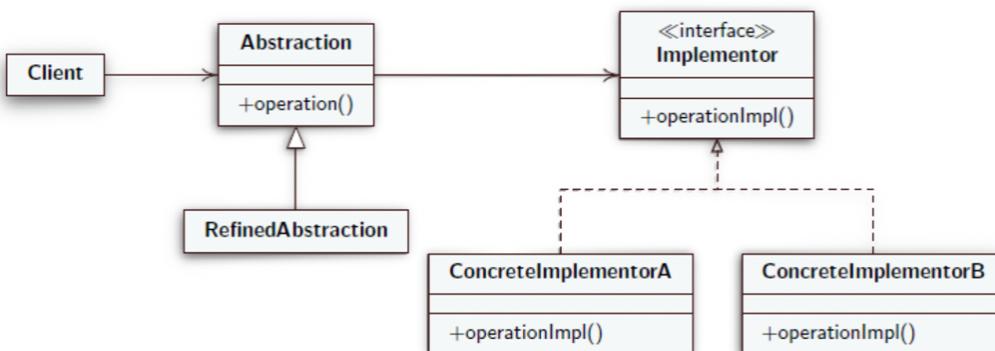
Permette di dividere una grande classe o un set di classi in due gerarchie separate (astrazione e implementazione); Disaccoppiare l'astrazione dall'implementazione così che le due possono variare indipendentemente

Problema:

- Esempio: Classe *Shape* con due sottoclassi *Circle* e *Square*. Si vuole estendere questa gerarchia per incorporare i colori, creando le sottoclassi forme *Red* e *Blue*, quindi si avranno 4 sottoclassi (*RedSquare*, *RedCircle*, *BlueSquare*, *BlueCircle*)
- Aggiungere forme e colori alla gerarchia, cresce esponenzialmente, es: se aggiungo un triangolo saranno altre due classi

Soluzione:

- Questo DP switcha dall'ereditarietà alla composizione di oggetti, in questo caso si avrà una classe *Shape* e una *Color*
- *Abstraction* definisce l'interfaccia per i client e mantiene un riferimento ad un oggetto *Implementor*; Inoltre le richieste del client al suo oggetto *Implementor*
- *RefinedAbstraction* estende *Abstraction*
- *Implementor* definisce l'interfaccia per le classi dell'implementazione. Questa interfaccia non deve corrispondere ad *Abstraction*, di solito *Implementor* fornisce operazioni primitive invece *Abstraction* operazioni di più alto livello basate su queste primitive
- *ConcreteImplementor* implementa l'interfaccia *Implementor* e fornisce le operazioni concrete



Applicabilità:

- Usa questo DP quando bisogna dividere e organizzare una classe monolitica che ha varianti di alcune funzionalità
- Oppure per switchare implementazione a run-time
- Un'implementazione non sarà connessa permanentemente a un'interfaccia; Si può cambiare implementazione senza dover ricompilare *Abstraction* e *Client*

TEMPLATE METHOD – COMPORTAMENTALI

Intento:

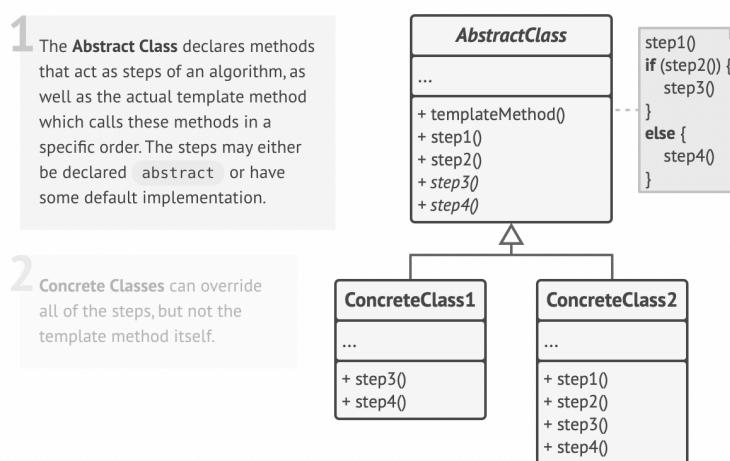
Permette di definire lo scheletro di un algoritmo, nella superclasse, e lasciare alle sottoclassi la possibilità di fare override di alcuni o tutti i passi di questo algoritmo, senza cambiare la sua struttura

Problema:

- Se si sta realizzando un'applicazione per estrarre vari tipi di documenti (doc, pdf, csv...)
- La prima versione include i doc, la seconda i pdf...
- Quindi ci saranno 3 classi con il codice molto simile, cambia solo il modo di estrarre e analizzare il documento nel formato differente; Ci sarà codice duplicato

Soluzione:

- Questo DP permette di “spezzettare” l’algoritmo in vari step, trasformare questi step in metodi e inserire delle chiamate a questi metodi dentro un *template method*
- Gli step potranno essere *abstract* o avere qualche implementazione
- Le sottoclassi faranno *override* di questi metodi



Applicabilità:

- Usa questo DP quando vuoi permettere al client di estendere solo alcuni step di un algoritmo, ma non dell’intero algoritmo o della sua struttura
- Oppure quando hai tante classi che contengono lo stesso algoritmo ma con alcune differenze, quindi bisognerà cambiare tutte le classi; Con questo DP no

STRATEGY – COMPORTAMENTALI

Intento:

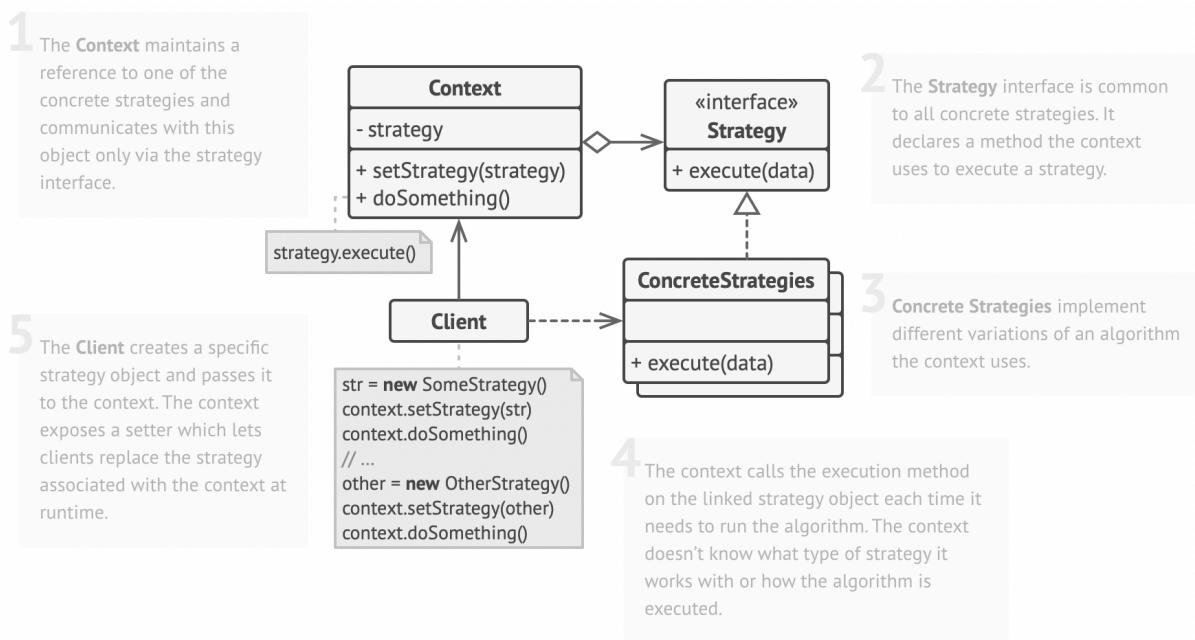
Permette di definire una famiglia di algoritmi e mettere ciascuna di essa dentro una classe separata, e rendere i loro oggetti intercambiabili

Problema:

- Se si vuole creare un'applicazione di un navigatore con la funzionalità di creare il percorso, la prima versione costruirà la strada per le strade (macchine), la seconda versione costruirà la strada a piedi, poi in bici...
- Ogni volta che si aggiunge un nuovo algoritmo di “routing” il codice si duplicherà
- Ogni cambiamento a un algoritmo interesserà tutta la classe

Soluzione:

- Questo DP permette di creare una classe che fa qualcosa in molti modi diversi ed estrarre questi algoritmi dentro classi separate, chiamate *strategies*
- La classe *Context* avrà un campo per salvare un riferimento a una strategia, delegherà il lavoro a un oggetto *strategy* collegato, invece di eseguire il lavoro nel suo
- Il *Client* passa la strategia desiderata al *Context*
- Così il *Context* sarà indipendente dai *concrete strategy*, è possibile aggiungere nuovi algoritmi o modificarne uno esistente senza cambiare il codice del *Context* o di qualche strategia



Applicabilità:

- Usa questo DP quando vuoi usare diverse varianti di un algoritmo dentro un oggetto e poter switchare da un algoritmo a un altro a run-time
- Oppure quando ci sono tante classi simili che differiscono solo dal modo in cui eseguono qualche comportamento

STATE – COMPORTAMENTALI

Intento:

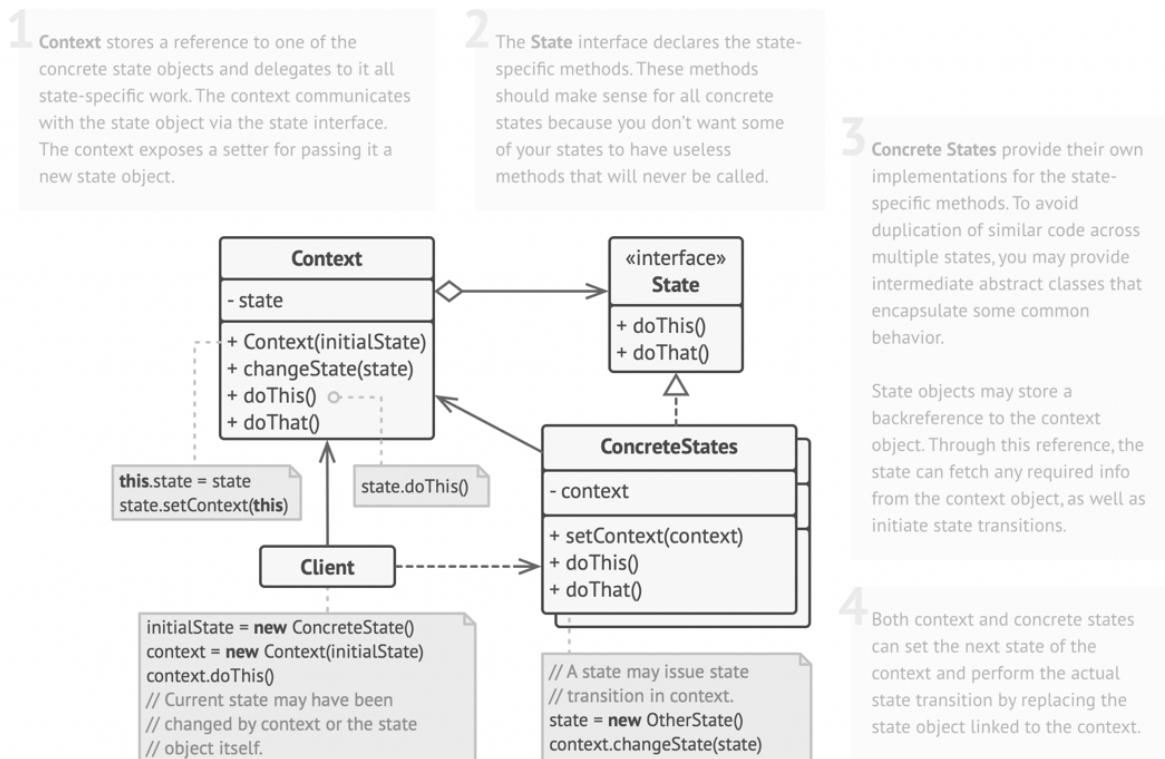
Permette di cambiare il comportamento di un oggetto quando il suo stato interno cambia, sembrerà come se l'oggetto abbia cambiato classe

Problema:

- Collegato al concetto di *automa a stati finiti*
- In qualunque momento, ci sono un numero finito di stati in cui un programma si può trovare; Dentro uno stato il programma si comporta differentemente e il programma può switchare da uno stato all'altro istantaneamente
- Un programma con tanti *switch o if*

Soluzione:

- Questo DP permette di creare classi diverse per tutti i possibili stati di un oggetto ed estrae tutti i "comportamenti in base allo stato" dentro queste classi
- *Context* salva una referenza a un *oggetto state* che rappresenta il suo stato corrente, e delega tutti i "lavori in base allo stato" a quest'oggetto
- Per cambiare il *Context* dentro un altro stato, rimpiazza l'*oggetto state* con un altro che rappresenta il nuovo stato



Applicabilità:

- Usa questo DP quando hai un oggetto che si comporta in modo diverso in base al suo stato, il numero di stati è enorme e il codice di quello specifico stato cambia spesso
- Oppure quando ci sono classi che hanno un sacco di *switch o if* per alterare il comportamento
- Oppure quando c'è tanto codice duplicato attraverso stati simili e passaggi di condizione basati su un ASF

OBSERVER – COMPORTAMENTALI

Intento:

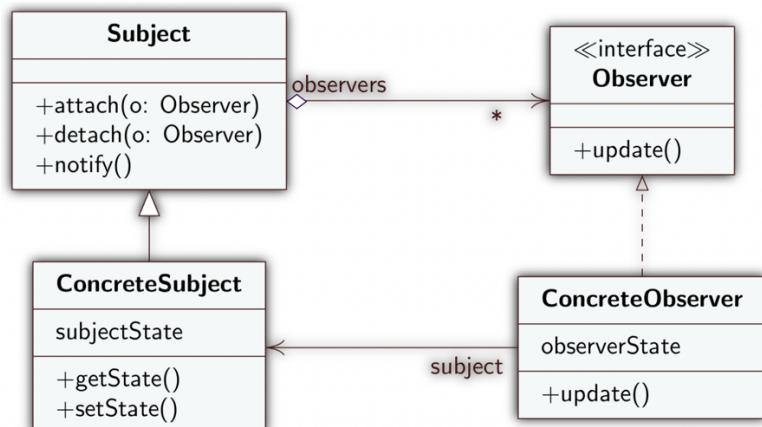
Permette di definire una dipendenza *uno a molti* fra oggetti, così che quando un oggetto cambia stato tutti i suoi oggetti dipendenti sono notificati e aggiornati automaticamente

Problema:

- Se si hanno due oggetti *Customer* e *Store*; Il *Customer* è interessato a un prodotto che ancora deve uscire nei negozi, lui andrà ogni giorno al negozio per vedere se è disponibile, ma finché il prodotto non è ancora uscito sarà inutile
- Oppure lo *Store* può mandare tante email a tutti i *Customer* ogni volta che un nuovo prodotto è disponibile; Questo aiuta molti *Customer* a fare strada inutile, ma molti altri possono non essere interessati

Soluzione:

- *Subject* conosce i suoi osservatori, un qualsiasi numero di *Observer* può osservare un *Subject*. Implementa funzionalità per aggiungere, togliere e notificare gli *Observer*
- *Observer* definisce un'interfaccia (*update()*) comune a tutti gli oggetti che necessitano della notifica
- *ConcreteSubject* tiene lo *stato* che interessa agli oggetti *ConcreteObserver*. Notifica i suoi osservatori quando il suo stato cambia ed eredita da *Subject*
- *ConcreteObserver* tiene un riferimento a un *ConcreteSubject*, tiene lo stato che deve rimanere consistente con quello del *Subject*. Implementa *Observer* per ricevere notifiche dei cambiamenti del *Subject*. Dopo la notifica lo può interrogare per ottenere il nuovo stato



Applicabilità:

- Usa questo DP quando il cambiamento di stato di un oggetto richiede il cambiamento di altri oggetti, e il set di oggetti non si conosce a priori o cambia dinamicamente
- Oppure quando alcuni oggetti devono osservarne altri, ma solo per un tempo limitato o in specifici casi

MEDIATOR – COMPORTAMENTALI

Intento:

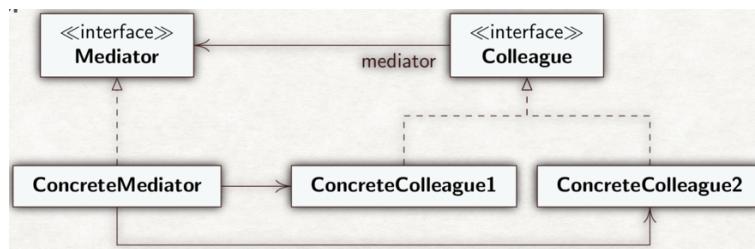
Permette di ridurre le dipendenze caotiche tra gli oggetti. Limita le comunicazioni dirette tra gli oggetti e li forza a collaborare solo attraverso l'oggetto *Mediator*

Problema:

- Esempio alcuni elementi di un form interagiscono con altri (bottone per validare email e pass, prima di salvare i dati)
- Avendo questa logica implementata direttamente dentro il codice degli elementi del form, le classi saranno difficili da riusare in altri form dell'app
- Difficile cambiare il comportamento dell'intero sistema perché è distribuito fra gli oggetti e bisognerebbe sottoclassarli tutti o quasi

Soluzione:

- Far collaborare i *Colleague* indirettamente, chiamando un *oggetto mediator* che reindirizza le chiamate al *Colleague* appropriato
- I *Colleague* dipenderanno solo su una singola classe *Mediator*
- Si incapsula il comportamento collettivo in un *Mediator* separato. Fa da intermedio ed evita che i *Colleague* dipendano fra loro
- *ConcreteMediator* implementa il comportamento cooperativo e coordina i *Colleague*
- Ogni *Colleague* conosce solo il *Mediator*, e comunica solo con lui quando avrebbe comunicato con un altro *Colleague*
- I *ConcreteColleague* mandano richieste e le ricevono a/da un *Mediator*, lui implementa il comportamento cooperativo inoltrando le richieste a i *ConcreteColleague* corretti
- L'astrazione di *Colleague* serve a imporre il requisito di un *Mediator* ai *ConcreteColleague*



Applicabilità:

- Usa questo DP quando è difficile cambiare qualcosa delle classi perché sono fortemente accoppiate con altre classi; Un oggetto conosce tutti gli altri
- Oppure quando non è possibile riusare un componente in un programma diverso perché è troppo dipendente da altri componenti

COMMAND – COMPORTAMENTALI

Intento:

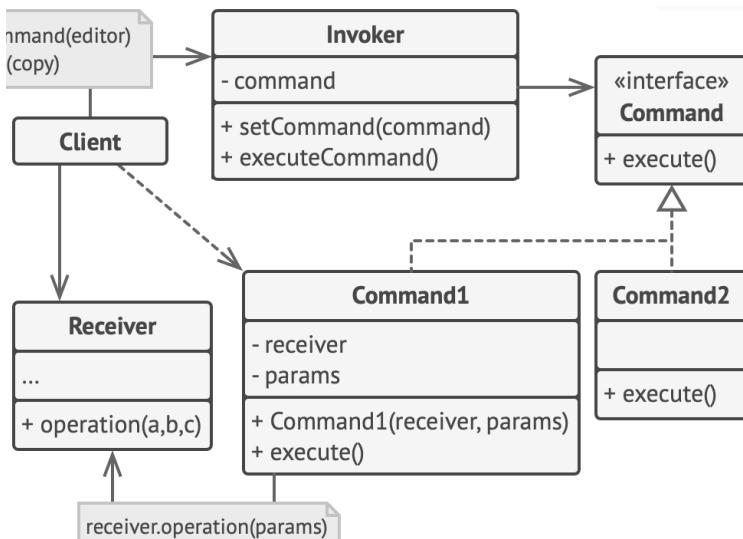
Trasforma una richiesta in un oggetto assestante che contiene tutte le riguardanti la richiesta. Permette di passare una richiesta come argomento di un metodo, ritardare o mettere in coda una richiesta; Incapsulare una richiesta, in un oggetto

Problema:

- Esempio: un nuovo text-editor, bisogna creare una toolbar con tanti bottoni per le varie operazioni. Come primo passo si può pensare di creare una classe *Button*, come un generico bottone. Poi vogliamo aggiungere altri tipi di bottoni (OkButton, CancelButton, OpenButton...), fanno cose diverse ma avranno il codice molto simile, perché si attiveranno tutti quando c'è l'evento del click. Questo approccio risulta difettoso perché ci saranno tante classi. Alcune operazioni saranno eseguite in parti diverse del programma, quindi se vogliamo aggiungere nuove funzionalità (scorciatoie, menu..), il codice del bottone si duplicherà.

Soluzione:

- Questo DP suggerisce di non mandare richieste direttamente, ma di estrarre i dettagli della richiesta, il nome del metodo e la lista degli argomenti in classi *Command* separate con un singolo metodo che "triggera" la richiesta
- L'*Invoker* inizializza la richiesta. Avrà un campo per salvare la referenza a un oggetto *command*; Triggerà il command invece di mandare la richiesta direttamente al ricevente; Di solito non crea il command
- L'interfaccia *Command* solitamente dichiara un metodo per eseguire il command
- I *ConcreteCommand* implementano richieste di vario tipo; Non esegue lui stesso il lavoro, ma passa la chiamata a un oggetto; I parametri richiesti per eseguire un metodo su un oggetto ricevuto, possono essere i campi dentro i *ConcreteCommand*
- Il *Receiver* contiene delle operazioni; Molti commands tengono i dettagli di come passare una richiesta al *Receiver*, mentre lui stesso esegue il lavoro
- Il *Client* crea e configura i *ConcreteCommand*, passa tutti i parametri della richiesta, l'istanza del *Receiver* dentro il costruttore del *Command*; Dopo questo il *Command* può essere associato a uno o più *Invoker*



Applicabilità:

- Usa questo DP quando vuoi parametrizzare oggetti con operazioni
- Oppure quando vuoi mettere in coda operazioni, schedularle o eseguirle
- Oppure quando vuoi implementare operazioni reversibili

CHAIN OF RESPONSABILITY – COMPORTAMENTALI

Intento:

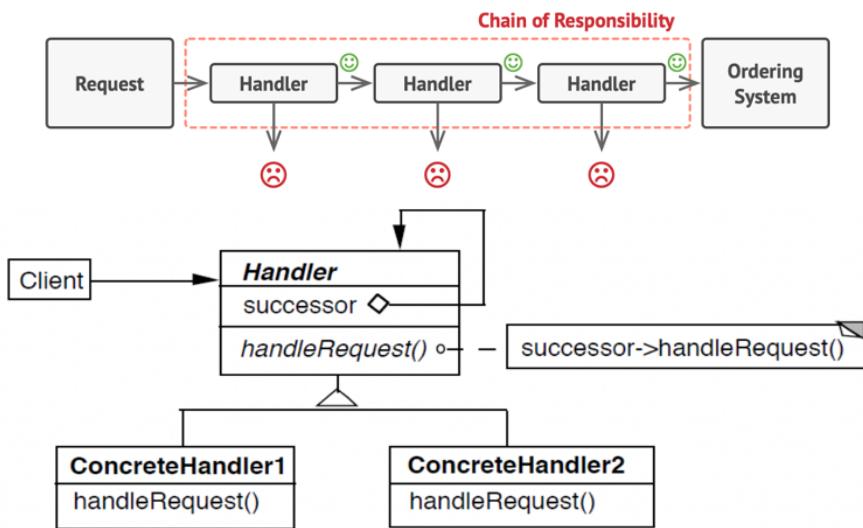
Evitare di accoppiare il mandante di una richiesta con il ricevente, dando la possibilità a più di un oggetto di gestire la richiesta; Permette di passare una richiesta in una catena di gestori. Una volta ricevuta la richiesta, ogni gestore decide se processare la richiesta o se passarla al prossimo gestore della catena

Problema:

- Esempio: Sistema di ordinazione online; Si vuole limitare l'accesso al sistema solo gli utenti autenticati possono ordinare, chi ha i permessi di amministratore ha accesso completo agli ordini L'applicazione può provare ad autenticare un utente quando riceve una richiesta che contiene le credenziali, se non sono corrette non va avanti Poi si vorranno aggiungere nuovi controlli sequenziali, cambiare un check potrebbe colpire anche gli altri; Provando a riusare il check per proteggere gli altri componenti del sistema si duplica il codice da questi componenti che richiedono il check, ma non tutti

Soluzione:

- Questo DP trasforma alcuni comportamenti in oggetti assestanti *handlers*
- *Handler* definisce l'interfaccia per gestire la richiesta, può fare riferimento a un successore
- *ConcreteHandler* gestisce le richieste; può accedere al suo successore; gestisce la richiesta se può farlo, sennò la passa al successore
- *Client* crea la richiesta e la invia al *ConcreteHandler*
- Dopo aver ricevuto una richiesta, un *handler* decide se lui riesce a processarla. Se ci riesce, non passa la richiesta a nessun futuro; sarà l'unico *handler* a processare la richiesta o non del tutto
- Questa catena può essere vista con un albero di oggetti
- Importante che tutti gli *handlers* implementino la stessa interfaccia, ogni *ConcreteHandler* dovrebbe preoccuparsi solo di avere il metodo *execute*. Così si può comporre la catena a run-time usando vari *handlers* senza far dipendere il codice con le loro classi concrete



Conseguenze:

- Usa questo DP quando il programma si aspetta di processare diversi tipi di richieste in tanti modi, ma l'esatto tipo della richiesta e la loro sequenza non si conoscono a priori
- Oppure quando è fondamentale eseguire degli *handlers* in un ordine preciso
- Oppure quando gli *handlers* e il loro ordine potrebbero cambiare a run-time