DESIGN PATTERN

Design pattern
I design pattern sono soluzioni di successo e ben ottimizzate per problemi ricorrenti. Si mira a riusare la soluzione a un problema ricorrente che è già stata testata, con una documentazione disponibile e le conseguenze, nella maggior parte dei casi note. Esistono vari cataloghi di design pattern in base alla situazione di programmazione a cui ci troviamo davanti: per sistemi concorrenti, distribuiti, real time ecc.

Le soluzioni dei design pattern sono adattabili al contesto in cui si trovano e specializzabili.
I design pattern oltre ad avere i vantaggi citati hanno altri vantaggi

- Aiutano i principianti a programmare come un esperto
- Aiutano gli esperti nella scrittura del codice dato che non devono costruire soluzioni nuove per problemi già risolti
- Analizzano le loro proprietà, quali sicurezza, affidabilità e riuso

La descrizione di un design pattern include 5 parti fondamentali
- Nome: Permette di identificare una soluzione con un numero ristretto di parole, in maniera tale da essere brevi e coincisi nella descrizione di un software
- Intento: Descrive brevemente le funzionalità
- Problema: descrive il problema a cui il pattern è applicato e le condizioni necessarie per applicarlo
- Soluzione: Descrive gli elementi che costituiscono il design pattern e i rispettivi ruoli
- Conseguenze: Indicano risultati, compromessi e vantaggi e svantaggi della soluzione

I design pattern sono organizzati in base allo scopo
- Creazionali: riguardano la creazione di istanze
- Strutturali: riguardano la scelta della struttura
- Comportamentali: riguardano la scelta dell'incapsulamento di algoritmi

I design pattern creazionali servono ad astrarre il processo di creazione delle classi. Rendono un sistema indipendente da come i suoi oggetti sono creati
Incapsulano la conoscenza delle classi concrete che un sistema usa, spesso in questo modo fa dipendere le classi client esclusivamente dalle interfacce
Nascondono come le istanze delle classi sono create e composte

Nota: Non sempre l'utilizzo di un design pattern semplifica la stesura del codice, ma possono esserci casi in cui applicare un design pattern va a complicare la progettazione di un codice. Il vantaggi in questo caso sono riconducibili solamente al fatto che il codice è scritto e testato in maniera professionale. E' importante quindi non abusare dei design pattern quando non c'è una reale necessità

SINGLETON

Alcune classi dovrebbero avere una sola istanza e per evitare di usare variabili globali che tengono l'unica istanza al loro interno si usa il DP singleton. Esso permette di creare un'unica istanza e controllare gli accessi ad essa. Il Singleton viene implementato tramite un metodo **static** che prende il nome di **getInstance()** Il metodo getIstance sarà un metodo statico, in maniera tale da essere indipendente da istanze della classe. Oltre al metodo getIstance vado

ad inserire all'interno dell'oggetto una variabile statica che servirà a contenere l'istanza dell'oggetto che andremo a creare. La variabile sarà anch'essa statica in maniera tale da poter essere inizializzata anche in assenza di istanze della classe.

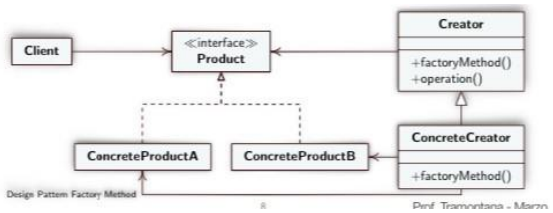
Il metodo getInstance avrà al suo interno una condizione, che verifica se la variabile contenitore è null. Se è null la va a inizializzare con un'istanza, altrimenti restituisce semplicemente la variabile.
Una variante del singleton è il multiton. Il multiton serve per creare un numero finito di istanze. Viene implementato nello stesso identico modo ma con un contatore e una lista per tener traccia delle istanze create.

FACTORY METHOD

Un framework usa classi astratte per definire e mantenere relazioni tra oggetti, esso deve creare oggetti ma conosce solo classi astratte, dunque, il factory method si occupa di definire un'interfaccia per creare un oggetto ma lascia che le sottoclassi decidano quale classe (compatibile) istanziare, in altre parole, esso rimanda l'istanziazione alle sottoclassi. Questo DP nasconde al client la logica e il tipo di oggetto istanziato, lasciandogli solo l'utilizzo. Tutto orbita intorno al metodo FactoryMethod() che si occupa di decidere quale sottoclasse istanziare e ne ritorna un riferimento. Nel design pattern sono presenti:

- **Product**: l'interfaccia comune per tutti gli oggetti;
- **ConcreteProduct**: un'implementazione di Product:
- **Creator**: dichiara il FactoryMehod che ritorna un oggetto di tipo Product;
- **ConcreteCreator**: implementa il FactoryMethod() o ne fa override, essa sceglie quale ConcereteProduct istanziare e ne ritorna l'istanza.

Ovviamente questo DP permette di rendere trasparente l'uso di diversi ConcreteProduct. Factory Method ha senso solo se ci sono almeno 2 ConcreteProduct.



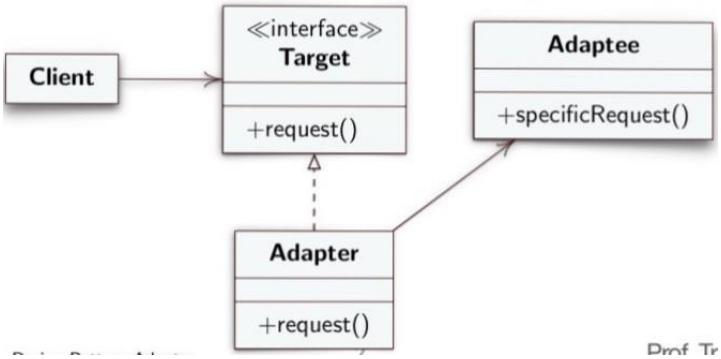
ADAPTER

Alcune volte una classe di una libreria non può essere usata perché incompatibile con l'interfaccia che si aspetta l'applicazione che stiamo scrivendo. Quando non è possibile cambiare l'interfaccia o l'applicazione è possibile utilizzare il DP Adapter che converte l'interfaccia di una classe in un'altra interfaccia che i client si aspettano, dunque, elimina il problema delle interfacce incompatibili. La soluzione si articola in tre classi:

- **Target**: l'interfaccia che i client si aspettano;
- **Adaptee**: l'oggetto di libreria che vorremmo adattare perchè è stato cambiato;
- **Adapter**: converte la chiamata che fa una classe client all'interfaccia della classe di libreria. Adapter tiene il riferimento all'oggetto di libreria (Adaptee) e implementa le

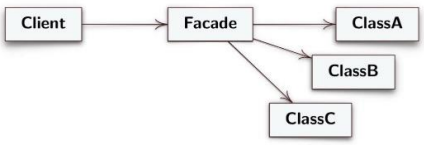
chiamate verso di esso. In una variante “**classe adapter**”, Adapter è **sottoclasse** di Adaptee;

Questo DP usa la **lazy initialization** poichè alloca l’oggetto di Adaptee subito prima di effettuare la chiamata per evitare di sprecare spazio. Un’altra variante di questo design pattern è l’**adapter a due vie** che propone al client sia la nuova che la vecchia interfaccia. Questa variante è automaticamente implementata dal class Adapter poichè essendo adapter ereditato da adaptee, porta con sé tutti i suoi metodi.
Inoltre, Adapter ci permette di aggiungere **precondizioni** e **postcondizioni** per effettuare controlli di sicurezza. D’altra parte però, esso aggiunge un **livello di indirettezza** perchè ad ogni invocazione ne corrisponde un’altra, questo potrebbe causare **rallentamenti** e scarsa comprensibilità del codice.



FACADE

Spesso si hanno tante classi che svolgono funzioni correlate e l’insieme delle interfacce può essere complesso e inoltre a volte si vorrebbero ridurre le comunicazioni dirette tra client e sottosistema. A tal proposito viene usato il design pattern Facade che fornisce un’**interfaccia unificata** al posto di un insieme di interfacce in un sottosistema. Facade fornisce un’unica interfaccia di alto livello e invoca i metodi delle sottoclassi che nasconde. Questo DP riduce le dipendenze di compilazione e promuove l’**accoppiamento lasco** tra client e sottosistema.

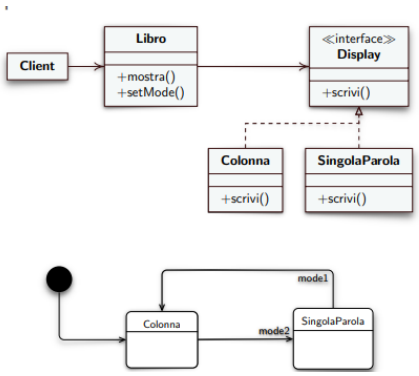


STATE

Il comportamento di un oggetto dipende dal suo stato e a volte il comportamento deve cambiare a run-time in base allo stato, l'intento del DP state è quello di permettere ad un oggetto di alterare il suo comportamento quando il suo stato interno cambia, ovvero, quello di far sembrare che l'oggetto abbia cambiato classe. Per far ciò, vengono usate le seguenti classi:

- **Context**: definisce l'interfaccia che interessa al client e mantiene un'istanza di una classe ConcreteState che definisce lo stato corrente;
- **State**: definisce un'interfaccia che incapsula il comportamento associato ad un particolare stato del context;
- **ConcreteState**: sono le sottoclassi che implementano ciascuna il comportamento associato ad uno stato del context:

Grazie a questo design pattern, il comportamento associato ad uno stato è localizzato in una sola classe (ConcreteState). Per tale motivo si possono aggiungere nuovi stati facilmente, creando nuove sottoclassi. Inoltre, la logica che gestisce il cambiamento di stato è separata dai vari comportamenti ed è in una sola classe (Context), tale separazione aiuta ad evitare stati inconsistenti.

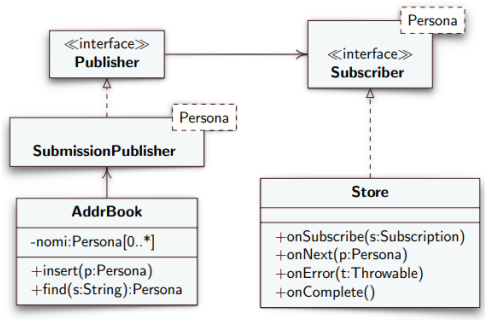


OBSERVER (PUBLISH-SUBSCRIBE)

Un sistema che è stato partizionato in un insieme di classi che cooperano deve mantenere la **consistenza** fra oggetti che hanno relazioni, questo viene garantito grazie al DP Observer che definisce una **dipendenza uno a molti** tra oggetti, così che quando un oggetto cambia stato tutti i suoi oggetti dipendenti sono notificati e aggiornati automaticamente. Questo DP è applicabile quando sono presenti delle dipendenze e un cambiamento su un oggetto richiede il cambiamento di altri, non si conosce quanti oggetti sarà necessario cambiare. I principali attori del DP sono **Subject** e **Observer**: un subject può avere tanti observer che dipendono da esso e gli observer sono notificati quando lo stato del subject cambia. In particolare:

- **Subject**: conosce i suoi osservatori e implementa le operazioni per aggiungere e togliere observer o per notificarli;
- **Observer**: definisce un'interfaccia comune a tutti gli oggetti che necessitano la notifica;
- **ConcreteSubject**: tiene lo stato che interessa agli oggetti ConcreteObserver, eredita da subject;
- **ConcreteObserver**: tiene un riferimento all'oggetto ConcreteSubject e tiene lo stato che deve rimanere consistente con quello di Subject, esso riceve notifiche e può interrogare il subject per ottenere di nuovo il dato di una notifica passata.

Ovviamente, il subject conosce solo la classe Observer e non ha bisogno di conoscere i ConcreteObserver, infatti la notifica è mandata a tutti, il ConcreteSubject no sa quanti sono gli Observer, è importante notare che l'osservatore può decidere se gestire o ignorare la notifica.



Observer in java
Il design pattern observer è talmente tanto utilizzato che è stato implementato nella libreria java.util

Vengono implementati 2 tipi Observable e Observer

- Observable: Svolge il ruolo di subject. Tiene traccia di tutti gli oggetti che dipendono da lui e li notifica tramite notifyObservers() non appena è avvenuto un cambiamento.

All'interno, la classe Observable tiene un flag, indicante se ci sono stati cambiamenti o meno. Il valore del flag può essere cambiato col metodo setnotify()

- Observer: E' un'interfaccia che implementa solo il metodo update

Il problema della libreria è che, non appena viene richiamato il metodo update, l'oggetto observable viene bloccato, visto che il metodo update utilizzerà lo stesso thread.
Questa libreria è stata sostituita con la libreria ReactiveStreams

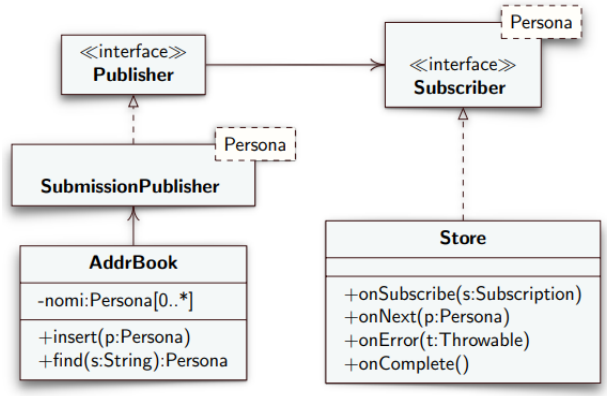
ReactiveStreams
A differenza degli observers, in reactivestreams, quando un concreteSubject richiama il metodo notify, viene creato un altro thread su cui eseguire la procedura update, in questo modo l'operato dei concreteSubject non viene bloccato.

ReactiveStreams definisce 5 tipi

- **Publisher:** Interfaccia che definisce il metodo **subscribe**
- **SubmissionPublisher**: Implementa l'interfaccia publisher, inoltre definisce il metodo **submit** per inviare le notifiche a tutti gli oggetti che andranno a iscriversi tramite il metodo **subscribe**
- **Subscriber:** Interfaccia che implementa 4 metodi

- **OnSubscribe**: Richiamato all'iscrizione
- **onComplete**: Richiamato alla fine della richiesta di aggiornamenti
- **onError**: Richiamato quando sono avvenuti errori nella comunicazione
- **OnNext**: Richiamato quando arrivano notifiche dai publisher

- Subscription: Definisce il collegamento tra publisher e subscriber. Implementa i metodi cancel e request per terminare la sottoscrizione o richiedere una nuova notifica



MVC

E' un pattern architetturale per le applicazioni interattive che individua tre componenti:

- **Model**: per funzionalità principali e dati;
  - Registra view e controller e li avvisa dei cambiamenti;
- **View**: per mostrare i dati;
  - Ognuno è associato ad un controller e mostra i dati letti da model;
- **Controller**: per prendere gli input dell'utente
  - Traduce gli eventi in richieste di servizio per Model e View.

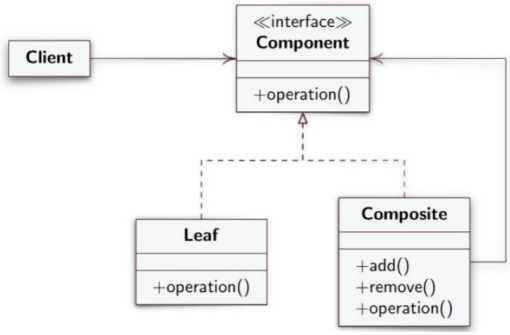
COMPOSITE

In alcuni scenari è necessario raggruppare elementi semplici tra di loro per formare oggetti più grandi, quello che si vuole è rendere la trattazione tra oggetti semplici e oggetti composti in modo omogeneo in modo da semplificare l'interfaccia del client. Quello che fa composite è svincolare il client dal capire se sta usando dati semplici o dati aggregati (composti), ovvero, gli fornisce un metodo per trattare gli oggetti semplici e quelli composti in modo uguale. In altre parole, composite compone oggetti in strutture ad albero per rappresentare gerarchie di parti di sistema, inoltre, permette di trattare anche gli oggetti composti in modo ricorsivo (un oggetto composto potrebbe contenere al suo interno altri oggetti composti).
Il DP si compone di:

- **Component**: è un'interfaccia (o una classe abastract) che rappresenta elementi semplici e non, essa dichiara le operazioni degli oggetti della composizione e implementa le operazioni comuni.
- **Leaf**: questa classe rappresenta elementi semplici, implementa component e definisce i comportamenti; avremo tante Leaf quanti gli oggetti che dobbiamo implementare;
- **Composite**: è una classe che rappresenta l'oggetto composto, essa è sottoclasse di Component e definisce il comportamento per l'aggregato di elementi leaf, inoltre,

tiene un riferimento a Component perchè deve poter individuare tutti gli elementi, sia Leaf che Component (ricorsione).

Per aggregare oggetti, è necessario fornire un metodo al client: questa operazione è data da metodo **Add()** presente in Composite e aggiunge alla lista di component presente dentro ogni composite, ogni elemento della lista è una parte dell'aggregato, allo stesso modo di Add esiste **Remove**() che ha comportamento uguale ma opposto. Notiamo che questi due metodi di manipolazione sono presenti solo nella sottoclasse, dunque per poterli utilizzare dovremo vincolare il client ad un tipo specifico, questo ci fa perdere di **trasparenza** ma ci fornisce **sicurezza** in quanto sapremo che quelle determinate operazioni quando chiamate saranno implementate opportunamente, D'altra parte, è presente una **variante** del DP che implementa Component come **classe astratta** che contiene add() e remove() implementati e operation() abstract. A questo punto il client non ha bisogno di legarsi ad un tipo, avremo dunque trasparenza ma non sicurezza, in quanto, le operazioni di add e remove non hanno senso su un tipo leaf e andranno dunque opportunamente gestite se chiamate su questo tipo.
Per migliorare l'efficienza, le liste di component sono presenti solo nel tipo Composite, inoltre, potremmo aggiungere una **cache** che ottimizza le operazioni chiamate sui child di un Composite per evitare di scorrere tutta la lista ogni volta.



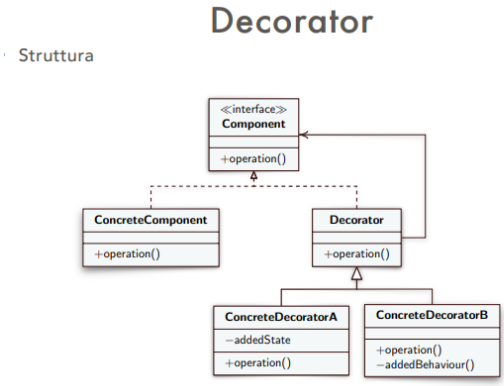
DECORATOR

Il DP decorator permette di aggiungere responsabilità a singoli oggetti e non all'intera classe, tramite questo design pattern, un client può controllare quando e come **decorare** un componente. L'approccio del Decorator è quello di racchiudere (wrappare) un componente dentro un altro oggetto che aggiunge responsabilità (wrapper), rimanendo comunque trasparente. Inoltre, il Decorator permette di evitare di definire un grandissimo numero di sottoclassi invece ottenibile combinando più decorator. In altre parole l'intento del DP è quello di aggiungere ulteriori responsabilità ad un oggetto a run-time. I decorator forniscono un'alternativa flessibile all'implementazione di sottoclassi per estendere funzionalità. Il decorator si articola in quattro classi:

- **Component**: definisce l'interfaccia di un oggetto su cui poter aggiungere responsabilità;
- **ConcreteComponent**: definisce un oggetto su cui poter aggiungere responsabilità;
- **Decorator**: mantiene un riferimento a un oggetto Component e definisce un'interfaccia conforme a quella di Component. Decorator inoltra le richieste al suo oggetto Component e può fare altre operazioni prima e dopo l'inoltro della richiesta;

- **ConcreteDecorator**: implementa la responsabilità aggiunta al Component, la responsabilità può essere solo privata.

Con decorator la stessa responsabilità può essere aggiunta più volte, semplicemente aggiungendo più istanze dello stesso ConcreteDecorator, inoltre con questo DP possiamo definire responsabilità in corso d'opera.



MEDIATOR

La distribuzione di responsabilità fra varie oggetti implica molte connessioni bidirezionali tra le classi, nel caso peggiore, un oggetto conosce tutti gli altri del sistema. Molte connessioni rendono un oggetto dipendente da altri e si comporta come se fosse monolitico, questo lo rende molto difficile da modificare e da riutilizzare. Per evitare che un oggetto conosca tutti gli altri per comunicare, il DP mediator implementa un **mediatore separato**, ovvero, una classe che funge da intermediario ed evita che gli oggetti dipendano tra di loro. In altre parole, il DP mediator definisce un oggetto che incapsula come un gruppo di oggetti interagisce, proponendo un accoppiamento lasco, inoltre, grazie a questa indirettezza, gli oggetti possono facilmente cambiare il loro stato mantenendo la comunicazione intatta.
Il mediator va utilizzato quando ci sono tante dipendenze tra le classi. A causa delle molteplici dipendenze esse diventano non strutturate e difficili da comprendere. Riutilizzare l'oggetto diventa praticamente impossibile visto che dipende da troppi altri oggetti.

Mediator fa uso di quattro classi:

- **Mediator**: definisce un'interfaccia per gli oggetti connessi (Colleague);
- **ConcreteMediator**: implementa il comportamento cooperativo e coordina gli oggetti Colleague;
- **Colleague**: interfaccia che definisce la struttura di un colleague per la comunicazione con il mediator;
- **ConcreteColleague**: mandano richieste e ricevono richieste, essi sostanzialmente parlano con mediator quando vorrebbero parlare con un altro ConcreteColleague.

Grazie a questo DP, la maggior parte della complessità dovuta dalla gestione delle dipendenze è spostata sulla classe mediator, questo rende gli oggetti più facili da

implementare e da mantenere. In genere il codice di mediator **non è riusabile** perché la gestione delle dipendenze è implementata solo per quella specifica applicazione.

Command

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:
Esemplo, 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 voliamo 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;
Triggera 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 ConcreteCommond 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 Concrete Command, 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.
Oppure quando vuoi implementare operazioni reversibili.

CHAIN OF RESPONSIBILITY

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 passare 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 dI check, ma non tutti.

Soluzione
Questo DP trasforma alcuni comportamenti in oggetti assestanti handiers
Mandler 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, senno la passa al successore
Client crea la richiesta e la invia al Concrete Hondier
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 handier 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 Concreterandler
dovrebbe preoccuparsi solo di avere Il metodo execute, Così si può comporre la catena a run-time usando vari handiers 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 handiers in un ordine preciso
Oppure quando gli hondiers e il loro ordine potrebbero cambiare a run-time.

BRIDGE

Intento:
Permette di splittare una grande classe o un set di classi in due gerarchie separato (astrazione e implementazione). Disaccoppiare l'astrazione dall'implementazione cosi che le due possono variare indipendentemente.

Problema:

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

Soluzione:

Questo DP switch 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 inoltra le richieste del client al suo oggetto Implementor;
RefinenedAobtroction estende Abstraction;
Implementor definisce l'interfaccia per le classi dell'implementazione, Questa interfaccia non deve corrispondere ad Abstraction, di solto implementor fornisce operazioni primitive invece Abstrortion operarioni di più alta livello basate su queste primitive;
Concreteimplementor implementa l'interfaccia Implementor e fornisce le operazioni concrete

Applicabilita:

Uso 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 Abstroction e Client

Template method

Intento:

Permette di definire lo scheletro di un algoritmo, nella superclasse, e lasciare alle sottoclassi la possibilità di fare ovveride 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 cuesti metodi dentro un tempiare methor
Gli step potranno essere abstroct 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 algorirmo, ma non dell'intero algoritmo o della sua struttura;
Oppure quando ha tante classi che contengono la stesso algoritmo ma con alcune differenti quindi bisognerà cambiare tutte le classi. Con questo DP no!

STRATEGY

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.

Abstract factory (Kit)

Intento:

Fornire una interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete

Appilcabilità:

Voglio indipendenza dal tipo concreto di prodotti che creo e uso;
Possibilità di configurare il sistema con una tra varie famiglie di prodotti;
Famiglie di prodotti correlati sono state progettate per essere usati insieme e si vuole imporre questo vincolo di coerenza;
Fornire librerie di classi intercambiabili rivelando solo le interfacce(API).

Conseguenze:

AbstractFactory rimandala creazione della versione di prodotti che devo usare ad una istanza di una sua sotto classe ConcreteFactory.
Rispetto al factory method, usa un oggetto e vi incapsulala crezione di molteplici prodotti.
Consente di cambiare facilmente la famiglia di prodotti;
Difficile aggiungere in seguito nuovi prodotti(l'interfaccia è ciò su cui si basano i client)

Prototype (clone)

Intento:

Specificare il tipo di oggetti da usare usando istanze prototipali;
Nuove Istanze sono clonate dal client a partire da tali istanze di riferimento già esistenti.
Il clients non crea mai istanze con "new": non conosce il tipo esatto del prodotto che usa.
Riceve un riferimento al clone di un prototipo
invece di AbstractClassobj = new ConcreteSubClass();
avrò AbstractClassobj = prototype->clone();

Applicabilità

Quando un sistema dovrebbe essere indipendente dal prodotto concreto che usa e quando la classe concreta da usare è nota solo a run-time: impossibile scrivere a priori sottoclassi (in stile FM) per incapsularne la creazione
•Ricevo in un parametro un oggetto (prototipo) già istanziato, da clonare
•Oppure, una classe ha solo in pochi stati possibili, quindi ho poche versioni di istanze tutte uguali
•Oppure Se la creazione di nuove istanze di classe è costosa

Conseguenze

Posso sostituire I prototipi facilmente(cambio l'oggetto o cambio la sua struttura interna) a run-time senza modifiche nel client;
Tengono celata la struttura(eventualmente) complessa al loro interno;
Ho meno bisogno di classi derivate, rispetto al factory method;
Devo implementare il metodoClone();
Devo poterli inizializzare, se necessario