**DESIGN PATTERN**

Caratteristiche design pattern: rappresentano soluzioni a problematiche ricorrenti, favoriscono il riuso, permette di evitare errori comuni, permettono di definire un linguaggio comune, permettono scrittura di codice valido e portano a una buona progettazione.

I design pattern hanno 2 criteri di classificazione:

1. Scopo, con le tre classi pattern creazionali, pattern strutturali, pattern comportamentali
2. Raggio d’azione (scope), un design pattern può riguardare classi o oggetti

**ABSTRACT FACTORY**

Classificazione: Creazionale basato su oggetti.

Scopo: Fornire una interfaccia per la creazione di famiglie di oggetti tra loro correlati.

Motivazione: Realizzazione di uno strumento per lo sviluppo di user interface (UI) in grado di supportare diversi tipi di look & feel. Per garantire la portabilità di una applicazione tra look & feel diversi, gli oggetti non devono essere cablati nel codice.

Applicabilità: Applicabile a sistemi che devono essere indipendenti dalle modalità di creazione dei prodotti con cui opera, e ai sistemi configurati per usare famiglie di prodotti diversi.

Conseguenze: La famiglia di prodotti può essere cambiata rapidamente perché la factory completa compare in un unico punto del codice, inoltre aggiungere nuove famiglie di prodotti richiede ricompilazione perché l'insieme di prodotti gestiti è legato all'interfaccia della factory

**FACTORY METHOD**

Classificazione: Creazionale basato su classi.

Scopo: Definire un’ interfaccia per la creazione di un oggetto, che consenta di decidere a tempo di esecuzione quale specifico oggetto istanziare.

Motivazione: E' un pattern ampiamente usato nei framework, dove le classi astratte definiscono le relazioni tra gli elementi del dominio, e sono responsabili per la creazione degli oggetti concreti.

Applicabilità: Quando una classe non è in grado di sapere in anticipo le classi di oggetti che deve creare, quando una classe vuole che le sue sottoclassi scelgano gli oggetti da creare, quando vogliamo che le classi delegano la responsabilità di creazione.

**ADAPTER**

Classificazione: Strutturale basato su classi/oggetti.

Scopo: Convertire l'interfaccia di una classe esistente incompatibile con un client, in una compatibile.

Motivazione: Consideriamo un editor che consente di disegnare e comporre elementi grafici. L'astrazione chiave è un singolo oggetto grafico. Supponiamo di voler integrare un nuovo componente, ma che questo non abbia una interfaccia compatibile con l'editor.

Applicabilità: Si usa quando si vuole riusare una classe esistente, ma con interfaccia incompatibile con quella desiderata.

Conseguenze: E' necessario prendere in considerazione l'effort necessario all'adattamento.

**COMPOSITE**

Classificazione: Strutturale basato su oggetti.

Scopo: Comporre oggetti in strutture che consentano di trattare i singoli elementi e la composizione in modo uniforme.

Motivazione: Le applicazioni grafiche consentono di trattare in modo uniforme sia le forme geometriche di base (linee, cerchi, ... ) sia gli oggetti complessi che si creano a partire da questi elementi semplici.

Applicabilità: Si usa quando si vogliono rappresentare gerarchie di oggetti in modo che

oggetti semplici e oggetti compositi siano trattati in modo uniforme.

Conseguenze: I client sono semplificati perché gli oggetti semplici e quelli compositi sono trattati allo stesso modo. Può rendere il sistema troppo generico, non è possibile fare in modo che un oggetto composito contenga solo un certo tipo di oggetti.

**DECORATOR**

Classificazione: Strutturale basato su oggetti.

Scopo: Aggiungere dinamicamente funzionalità (responsabilità) ad un oggetto. Il subclassing è una alternativa statica e il cui scope è a livello di classe e non di singolo oggetto.

Motivazione: Responsabilità quali il testo scorrevole o un particolare bordo devono poter essere aggiunti a livello di singolo oggetto. Un approccio flessibile è quello di racchiudere un oggetto elementare in un altro, che aggiunge una responsabilità particolare, l'oggetto contenitore è chiamato Decorator.

Applicabilità: Si applica quando è necessario aggiungere responsabilità agli oggetti in modo Trasparente e dinamico, si applica inoltre quando il subclassing non è adatto(subclassing non adatto perché è a livello di classe ma a noi serve a livello di oggetto singolo).

Conseguenze: Maggiore flessibilità rispetto all'approccio statico, inoltre evita di definire strutture gerarchiche complesse.

**OBSERVER**

Classificazione: Comportamentale basato su oggetti.

Scopo: Definire una dipendenza uno a molti tra oggetti, mantenendo basso il grado di coupling. In altre parole, la variazione dello stato di un oggetto deve essere osservata da altri oggetti, in modo che possano aggiornarsi automaticamente.

Motivazione: Lo scenario classico è quello di applicazioni con GUI, realizzate secondo il paradigma Model-View-Control. Quando il Model cambia, gli oggetti che implementano la View devono aggiornarsi.

Approccio sbagliato: utilizzare, nell'oggetto osservato, attributi pubblici oppure metodi pubblici che leggono il valore di un attributo protetto. Non è una buona soluzione perché non è scalabile (se aumentano troppo gli osservatori l'oggetto osservato è sovraccaricato dalle richieste), gli osservatori dovrebbero continuamente interrogare l'oggetto osservato, le variazioni rapide potrebbero comunque non essere rilevate da qualche osservatore.

Approccio corretto: Il pattern Observer prevede che gli osservatori si registrino presso l'oggetto osservato. In questo modo è l'oggetto osservato che notifica ogni cambiamento di stato agli osservatori. Quando l'osservatore rileva la notifica può interrogare l'oggetto osservato, oppure può svolgere altre operazioni indipendenti dal valore specifico dello stato.

Applicabilità: Gestire le modifiche di oggetti conseguenti alla variazione dello stato di un oggetto.

Conseguenze: La notifica è una comunicazione di tipo broadcast.

**TEMPLATE** **METHOD**

Classificazione: Comportamentale basato su classi.

Scopo: Definire la struttura di un algoritmo all'interno di un metodo, delegando alcuni passi alle sottoclassi. Le sottoclassi ridefiniscono solo alcuni passi dell'algoritmo ma non la sua struttura.

Motivazione: consideriamo un framework per costruire applicazioni in grado di gestire documenti diversi. Il Template Method definisce un algoritmo in base ad operazioni astratte che saranno definite nelle sottoclassi specifiche.

Applicabilità: È utilizzato per implementare la parte invariante di un algoritmo, lasciando alle sottoclassi la definizione degli step variabili, è utile quando ci sono comportamenti comuni che possono essere inseriti nel template.

Conseguenze: I metodi template permettono il riuso del codice; Creano una struttura di controllo invertito dove è la classe padre che chiama le operazioni ridefinite nei figli e non viceversa; Per controllare l'estendibilità delle sottoclassi, i metodi richiamati dal template sono chiamati metodi gancio (hook); I metodi hook possono essere implementati, offrendo un comportamento standard, che la sottoclasse può volendo ridefinire.

**STRATEGY**

Classificazione: comportamentale basato su oggetti.

Scopo: Definire ed incapsulare una famiglia di algoritmi in modo da renderli intercambiabili indipendentemente dal client che li usa.

Motivazione: Consideriamo la famiglia degli algoritmi di ordinamento. Ne esistono diversi. Costruiamo una applicazione che li supporti tutti, che possa essere facilmente estendibile, e che permetta una scelta rapida del tipo di algoritmo.

Applicabilità: Molte classi correlate differiscono solo per il comportamento (il pattern fornisce un modo per avere una interfaccia comune), applicabile anche quando sono necessarie più varianti di uno stesso algoritmo, a seconda dei tipi di dato in ingresso o delle condizioni operative.

Conseguenze: Il pattern separa l'implementazione degli algoritmi dal contesto dell'applicazione (usare il subclassing della classe Client per aggiungere un algoritmo non sarebbe stata una buona scelta); Le diverse strategie eliminano i blocchi condizionali che sarebbero necessari inserendo tutti i diversi comportamenti in una unica classe; Lo svantaggio principale è che i client devono conoscere le diverse strategie