

I pattern sono nati per aiutare a risolvere determinate tipologie di problemi che vengono normalmente riscontrate in fase di progettazione.

Esistono diversi tipi di pattern:

di creazione: risolvono problemi inerenti alla creazione di oggetti (Abstract Factory, Singleton, ...)

strutturali: risolvono problemi inerenti la composizione di classi/oggetti (Adapter, Composite, Decorator, Facade, Flyweight, ...)

comportamentali: risolvono i problemi di interazione e distribuzione di responsabilità tra classi o oggetti (Observer, Visitor, ...)

Singleton:

Assicura che una classe abbia una sola istanza: le successive richieste di creazione devono essere intercettate ed eliminate. In questo modo esiste un punto di accesso globale.

Flyweight:

E' un oggetto condiviso da utilizzare contemporaneamente ed efficientemente da più clienti (che però non devono accorgersi che si tratta di un oggetto condiviso).

Stato intrinseco: non dipende dal contesto di utilizzo e può essere condiviso tra i clienti. Viene memorizzato nel flyweight.

Stato estrinseco: dipende dal contesto quindi non può essere condiviso. Viene memorizzato dai clienti e passato al flyweight quando viene passata la sua istanza. I clienti accedono al flyweight non direttamente, ma tramite una factory che istanzia il flyweight.

Strategy:

Il client referencia un'interfaccia che fornisce l'implementazione di un algoritmo.

Adapter:

Converte l'interfaccia originale di una classe nell'interfaccia specifica che si aspetta di trovare il cliente.

Questo pattern viene utilizzato quando si desidera riutilizzare una classe esistente, o la sua interfaccia non è conforme a quella desiderata.

Esistono quindi un target, l'interfaccia che il cliente vuole vedere, e un adapter che eredita da target e che adatta l'interfaccia.

Decorator:

Permette di aggiungere dinamicamente delle responsabilità ad un oggetto (alternativa alla specializzazione che fa la stessa cosa ma staticamente).

Esiste un Component, visto dal cliente che è implementato dal componente concreto di base e dal decoratore (che mantiene sempre un collegamento con il componente che decora).

Il componente concreto definisce il tipo di oggetto a cui devono essere aggiunte le responsabilità dinamicamente.

Il decoratore è una classe astratta che definisce un'interfaccia conforme all'interfaccia di Component. Il decoratore è a sua volta ereditato da decoratori concreti che aggiungono le responsabilità al componente referenziato.

Composite:

Permette di comporre oggetti in una struttura gerarchica ad albero per trattare oggetti singoli e composti in modo uniforme.

C'è un component, una classe astratta acceduta dal cliente che dichiara l'interfaccia e realizza il comportamento di default dell'oggetto. In Component sono inserite tutte le operazioni che devono essere utilizzate dai clienti (nella maggior parte dei casi realizzando un comportamento di default che verrà specificato nelle sottoclassi).

Leaf eredita Component e descrive gli oggetti senza figli definendone il comportamento.

Composite eredita Component e descrive gli oggetti contenitori che hanno figli definendo il comportamento di tali oggetti. Il contenitore deve essere un attributo di Composite.

Di solito c'è un riferimento esplicito al genitore che permette l'attraversamento dell'albero; gli elementi che hanno come parent lo stesso componente devono essere gli unici figli di quel parent.

Nel caso di condivisione di elementi la gestione dei riferimenti ai genitori diventa difficoltosa, e le richieste verso l'alto sono spesso ambigue. In questi casi risulta necessario utilizzare un flyweight.

Se scegliamo la trasparenza: il cliente potrebbe cercare di fare cose senza senso, quindi le add/remove dichiarate nel Component devono essere gestite da opportune eccezioni.

Se scegliamo la sicurezza: il component non ha add/remove, dobbiamo avere metodi opportuni per capire se stiamo agendo su un composite (contenitore con foglie) perché qualunque invocazione sulle foglie deve generare errori in fase di compilazione.

Visitor:

Permette di definire una nuova operazione da effettuare sugli elementi di una struttura senza però modificare le classi degli elementi coinvolti.

Si separa quindi tutto il codice relativo a un singolo tipo di operazione su una singola classe, e per aggiungere un elemento, basta aggiungere una classe.

Il Visitor è una classe astratta che dichiara i metodi visit per ogni classe di elementi concreti.

Il Visitor Concreto eredita da Visitor e definisce tutti i metodi Visit, definisce l'operazione da effettuare sulla struttura.

Element è una classe astratta o interfaccia con un particolare metodo Accept che accetta un Visitor.

Concrete Element definisce l'Accept.

Observer:

Può capitare che ci siano oggetti interessati a sapere quando una classe subisce dei cambiamenti.

Gli approcci sono due: o la classe che cambia avverte tutte le classi interessate (metodo immobile perché nel caso in cui venga creata una nuova classe interessata, bisogna modificare anche il codice della classe che cambia affinché avverta anche la nuova classe) oppure viene utilizzato il metodo publisher-subscriber in cui le classi interessate si registrano presso la classe che cambia, che manda loro un messaggio quando subisce dei cambiamenti.

Model View Controller:

Permette di dividere un'applicazione in input (controller), output (view) e stato (model).

Il Modello gestisce i dati, risponde alle interrogazioni sui dati e alle istruzioni di

modifica dello stato generando un evento quando questo cambia. Registra inoltre in maniera anonima gli oggetti interessati alla notifica dell'evento (publisher-subscriber).

La View gestisce l'area di visualizzazione: mappa i dati e visualizza gli oggetti su un determinato output.

Il Controller gestisce gli input dell'utente e li invia al modello e/o alla view.

Differenze tra interfaccia e classe astratta

Interfaccia:

Non può essere istanziata

Non contiene implementazioni (le funzionalità derivano dalle classi che la implementano)

Non ha uno stato

Non ha attributi o metodi statici

Deve descrivere funzionalità semplici, implementabili da oggetti eterogenei

Deve essere stabile

Può ereditare o meno da altre interfacce

Non gestisce la creazione delle istanze delle classi che la implementano (effettuata dai costruttori delle relative classi o da una factory che può utilizzare uno switch, un dizionario + metodo Clone, un metodo per generare automaticamente un'istanza dato il nome della classe)

Classe astratta:

Non può essere istanziata

Può essere completamente implementata (le classi derivate realizzano le funzionalità non implementate e/o forniscono una realizzazione alternativa a quelle già implementate)

Può avere uno stato

Può avere attribui e/o metodi statici

Può descrivere funzionalità complesse comuni ad un insieme di oggetti omogenei

Può fornire un'implementazione di default

Può essere modificata (più facilmente di un'interfaccia)

Può ereditare o meno da altre interfacce o da altre classi (astratte e/o concrete)

Può gestire la creazione delle istanze delle sue sottoclassi.