Design Patterns

Ogni pattern descrive un problema che ricorre più volte nel nostro ambiente, descrive poi il nucleo della soluzione del problema in modo tale poter utilizzare una soluzione un milione di volte senza mai ripetersi due volte.

Nella maggior parte dei design patterns si cerca di separare ciò che cambia da ciò che rimane invariato, in modo tale che quando qualcosa cambia non si debba modificare il resto del codice.

Un design pattern è sostanzialmente composto da punto (dove cambia l'entità del design pattern) e da una soluzione (che non cambia).

Un design pattern è sostanzialmente composto da:

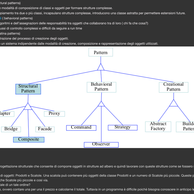
- Un nome
- Un caso scatenante (dove cambia)
- Una soluzione (che non cambia) che permette la situazione in cui il pattern può essere utilizzato.
- Conseguenze (che descrive gli elementi che cambiano/il compromesso/i pro e cons dell'adozione del pattern)
- Un esempio di implementazione (che determina i fattori chiave di criticità che si oltrepassano applicando il pattern. Sono utili per capire i trade-off e le alternative che descrive essere considerate per implementare il pattern).

Il quarto volume di Design Patterns comprende ventuno pattern suddivisi in:

- Azione e caratteristiche importanti per il vocabolario
- Pattern che si focalizzano sulla comunicazione tra oggetti
- Tecniche di realizzazione per l'accesso ai dati
- Modo per descrivere algoritmi e pattern di programmazione a più livelli
- Metaprogrammi, strategie di flusso di esecuzione e pattern di programmazione concorrente
- Applicazione aspetti e pattern di programmazione aspettuale
- Pattern per la manutenzione del codice legacy
- Pattern per la creazione di oggetti/componenti/servizi/dati configurabili
- Collaborazioni per a creare sistemi ad agente/ad agenti
- Implementazione curve di pure implementazione 5 pattern.
- Pattern per la concorrenza
- Pattern per creare sistemi che utilizzano
- Utilizzi vari esempi di utilizzo reale dei pattern in sistemi esistenti.

Design pattern è composto da 4 elementi essenziali:

- Un nome che descrive il problema
- Si occupano delle modalità di progettazione di classi e oggetti per fornire soluzione comune
- Pitoresco / incomprensibile la due è un pattern. Questo è sicuramente pittoresco, ma chiaro anche per il pubblico
- Un problema che descrive quando applicare il pattern
- Si occupano di algoritmi e dell'assegnazione della responsabilità tra oggetti che collaborano fra di loro (chi fa che cosa?)
- Consequence: dice quali sono le conseguenze e difficoltà che seguono a un tale
- Pattern composti / pattern polivalenti
- Pattern che si possono combinare per ottenere pattern più complessi.
- Analisi e discussione di varie soluzioni dato modelli di soluzione, conseguenze e rappresentazione degli oggetti coinvolti.



Pattern strutturali

Pattern composite

Composite è un modello di progettazione strutturale che consente di compone oggetti in strutture ad albero e quindi lavorare con queste strutture come se fossero singoli oggetti.

Immaginiamo di avere due tipi di oggetti: Prodotti e Scatole. Una scatola può contenere un certo numero di oggetti della classe Prodotti e un numero di Scatole più piccole. Quelle piccole Scatole possono contenere ulteriori scatole di dimensioni ancora più piccole e così via.

Una scatola esterna potrebbe essere strutturata in questo modo:

Nel mondo reale sarebbe facile, ovvero contare una pila una pila o conoscere a calcolare il totale. Tuttavia in un programma è difficile perché bisogna conoscere in anticipo le classi, quanti livelli di scatole ecc.

Il problema con l'approccio diretto è che il client deve conoscere tutte le classi che compongono l'ordine, quindi è difficile estendere il codice.

Successivamente suggerisce che tenti con tutti gli oggetti che implementano la stessa interfaccia o una classe astratta che tutti gli elementi secondari sono "scatole" e che Composite, che rappresenta un insieme di scatole, implementi anche la stessa interfaccia. Questo rende l'elemento a scatole indifferente a livelli di scatole annidate.

Un client funziona con tutti gli elementi attraverso l'interfaccia comune e non si preoccupa di come è strutturato l'albero di oggetti sotto di lui.

Facade è un modello di progettazione che fornisce un'interfaccia semplificata a una libreria, un framework o qualsiasi altro insieme complesso di classi. L'architettura di tale modello è strutturata nella maniera seguente: ci sono n client che comunicano con una facade che a sua volta comunica con n classi.

Esempio:

Immaginiamo di avere a che fare con un sistema complesso. Di solito bisogna inizializzare tutti gli oggetti, tenere traccia delle dipendenze, eseguire i metodi nell'ordine corretto rendendo così estremamente legato al dettaglio di implementazione della logica aziendale. Questo rende il codice difficile da testare e riutilizzare indipendente in quel momento.

Un'alternativa è quella di creare classe di facciata aggiungendo per l'insieme di oggetti che si vuole legare. Successivamente non contiene che probabilmente rendendo ancora più difficile chiarire cosa fa ciascuna classe del client che ne sfrutta.

Il vantaggio consiste nel fatto che il client non deve più preoccuparsi di come funziona il sistema complesso. Il client comunica solo con la classe di facciata, che fornisce tutte le funzioni richieste, mentre tutte le classi complesse del sistema vengono gestite in maniera opportuna. Tuttavia una classe di facciata può diventare un'interfaccia "globale" per l'intera applicazione.

Proxy è un modello di progettazione strutturale che fornisce un'interfaccia semplificata a una libreria, un framework o qualsiasi altro insieme complesso di classi. L'architettura di tale modello è strutturata nella maniera seguente: ci sono n client che comunicano con una classe proxy che a sua volta comunica con n classi.

Esempio:

Immaginiamo di avere a che fare con un sistema complesso. Di solito bisogna inizializzare tutti gli oggetti, tenere traccia delle dipendenze, eseguire i metodi nell'ordine corretto rendendo così estremamente legato al dettaglio di implementazione della logica aziendale. Questo rende il codice difficile da testare e riutilizzare indipendente in quel momento.

Un'alternativa è quella di creare classe proxy aggiungendo per l'insieme di oggetti che si vuole legare. Successivamente non contiene che probabile rendendo ancora più difficile chiarire cosa fa ciascuna classe del client che ne sfrutta.

Il vantaggio consiste nel fatto che il client non deve più preoccuparsi di come funziona il sistema complesso. Il client comunica solo con la classe proxy, che fornisce tutte le funzioni richieste, mentre tutte le classi complesse del sistema vengono gestite in maniera opportuna. Tuttavia una classe proxy può diventare un'interfaccia "globale" per l'intera applicazione.

Il pattern offre un concetto versatile e in varie particolare dello functionality del sistema. Su tale struttura si innesta l'idea che chiama e crea tre funzionare che è quindi modi.

Il proxy è utile quando una classe di facciata aggiungerà per l'insieme di oggetti che si vuole legare. Successivamente non contiene che probabile rendendo ancora più difficile chiarire cosa fa ciascuna classe del client che ne sfrutta.

Il vantaggio consiste nel fatto che il client non deve più preoccuparsi di come funziona il sistema complesso. Il client comunica solo con la classe proxy, che fornisce tutte le funzioni richieste, mentre tutte le classi complesse del sistema vengono gestite in maniera opportuna. Tuttavia una classe proxy può diventare un'interfaccia "globale" per l'intera applicazione.

Il pattern offre un concetto versatile e in varie particolare dello functionality del sistema.