Design Pattern

I design pattern sono strutture software (ovvero micro-architetture) per un piccolo numero di classi che descrivono soluzioni di successo per problemi ricorrenti e che sono buone da un punto di vista qualitativo(permettono di scrivere codice di qualità). Queste micro-architetture specificano le diverse classi ed oggetti coinvolti e loro interazioni. Si mira al riuso di un insieme di classi, ovvero la soluzione ad un problema ricorrente. Durante la progettazione, le conseguenze sulle classi di varie scelte potrebbero non essere note, e le classi potrebbero diventare difficili da riusare o non esibire proprietà. Con l'uso dei design pattern invece, è possibile implementare facilmente del codice riutilizzabile perchè ci danno un modo di ragionare, permettendo di utilizzare bene l'OOP in quanto mettono in evidenza principi della programmazione ad oggetti utilizzabili in maniera concreta per determinate soluzioni.

–> Bisogna utilizzare un design pattern solo se esso risolve il problema che si sta affrontando : Se il sistema non presenta nessuno dei problemi che risolve il design pattern che si sta adottando allora, utilizzando il design pattern si sta risolvendo un problema che non si ha e questa non è la scelta migliore.

I design pattern documentano soluzioni già applicate e che si sono rilevate di successo per certi problemi, che si sono evolute nel tempo. Aiutano i principianti ad agire da esperti e supportano gli esperti nella programmazione su grande scala. Evitano di re-inventare concetti e soluzioni , riducendo così il costo. Forniscono un vocabolario comune e permettono una comprensione dei principi del design. Garantiscono affidabilità, modificabilità, sicurezza, testabilità e riuso.

Un design pattern si compone delle seguenti parti fondamentali :

- Nome : permette di identificare il design pattern e di lavorare con un alto livello di astrazione, indica lo scopo
- Intento : descrive brevemente le funzionalità e lo scopo
- Problema : Motivazione + Applicabilità , descrive il problema in cui il pattern è applicato e le condizioni necessarie ad applicarlo
- Soluzione : descrive gli elementi (classi) che costituiscono il design pattern, le loro responsabilità e le loro relazioni
- Conseguenze : indicano risultati, compromessi, vantaggi e svantaggi nell'uso del design pattern

I design pattern sono organizzati sul catalogo GoF(Gang of Four) in base al loro scopo :

-> Creazionali : riguardano la creazione di istanze (Singleton, Factory Method, Abstract Factory, Builder, Prototype)

-> Strutturali: riguardano la scelta della struttura (Adapter, Facade, Composite, Decorator, Bridge, Flyweight, Proxy

-> Comportamentali : riguardano la scelta dell'incapsulamento di algoritmi (Iterator, Template Method, Mediator, Observer, State, Strategy, Chain of Responsability, Command, Interpreter, Memento, Visitor

Design pattern Creazionali

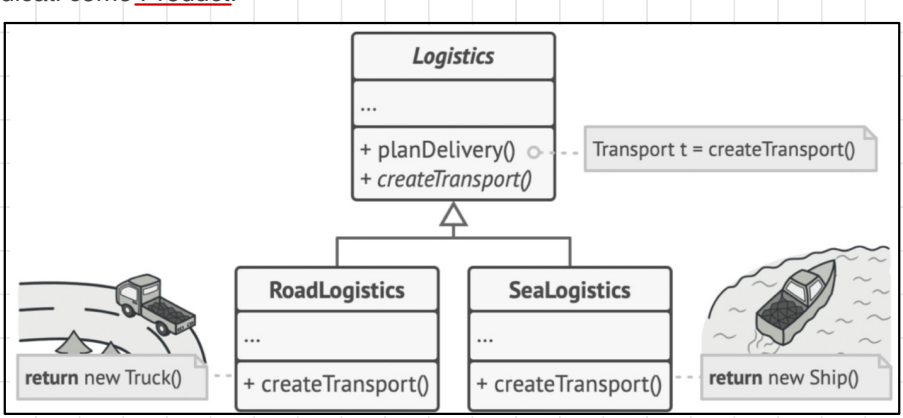
I design pattern creazionali sono un insieme di pattern del software che si concentrano sulla creazione degli oggetti in modo efficiente e flessibile. Essi forniscono soluzioni comuni per problemi di progettazione che si verificano nella creazione di oggetti e di classi. In generale, i design pattern creazionali offrono un modo strutturato e riusabile per affrontare i problemi comuni nella creazione degli oggetti e delle classi, migliorando la qualità del codice e la manutenibilità dell'applicazione.

Design pattern Factory Method

- Intento : Factory Method è un modello di progettazione di creazione che fornisce un'interfaccia per la creazione di oggetti in una superclasse, ma consente alle sottoclassi di modificare il tipo di oggetti che verranno creati.

- Problema : Immagina di creare un'applicazione per la gestione della logistica. La prima versione della tua app può gestire solo il trasporto su camion, quindi la maggior parte del tuo codice risiede all'interno della classe "Truck" (Camion). Dopo un po' la tua app diventa piuttosto popolare. Ogni giorno ricevi dozzine di richieste da compagnie di trasporto marittimo per incorporare la logistica marittima nell'app. Ottima notizia, vero? Ma per quanto riguarda il codice? Al momento, la maggior parte del codice è accoppiata alla classe Truck. L'aggiunta della classe Ship(Nave) all''app richiederebbe modifiche all'intera base di codice. Inoltre, se in seguito decidi di aggiungere un altro tipo di trasporto all'app, probabilmente dovrai apportare nuovamente tutte queste modifiche. Di conseguenza, ti ritroverai con un codice piuttosto sgradevole, pieno di condizionali che cambiano il comportamento dell'app a seconda della classe degli oggetti di trasporto.

- Soluzione : Il modello Factory Method suggerisce di sostituire le chiamate di costruzione di oggetti diretti (utilizzando l' operatore new) con chiamate a un metodo speciale "factory()". Non preoccuparti: gli oggetti vengono ancora creati tramite l' operatore new, ma viene chiamato dall'interno del factory method. Gli oggetti restituiti da un metodo factory sono spesso indicati come Product.



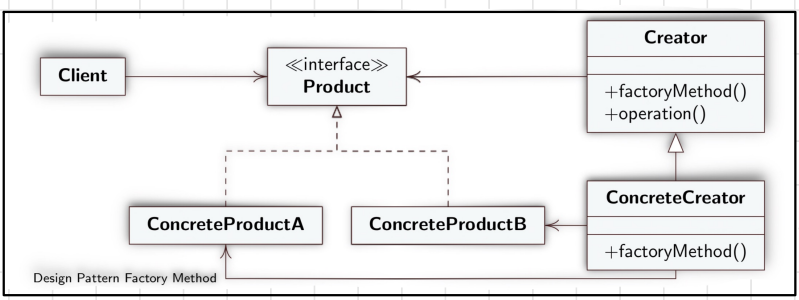
A prima vista, questo cambiamento può sembrare inutile: abbiamo semplicemente spostato la chiamata del costruttore da una parte all'altra del programma. Tuttavia, considera questo: ora puoi sovrascrivere il metodo factory in una sottoclasse e modificare la classe dei prodotti creati dal metodo.

C'è però una piccola limitazione: le sottoclassi possono restituire diversi tipi di prodotti solo se questi prodotti hanno una classe base o un'interfaccia comune. Inoltre, il tipo di ritorno del metodo factory nella classe base dovrebbe essere dichiarato come questa interfaccia.

Ad esempio, entrambe le classi Truck e Ship dovrebbero implementare l' interfaccia Transport, che dichiara un metodo chiamato deliver. Ogni classe implementa questo metodo in modo diverso: i camion consegnano il carico via terra, le navi consegnano il carico via mare. Il factory method nella classe Road Logistics restituisce oggetti camion, mentre il metodo factory nella classe SeaLogistics restituisce navi.

Il codice che utilizza il factory method (spesso chiamato codice client) non vede differenze tra i prodotti effettivi restituiti dalle varie sottoclassi. Il client tratta tutti i prodotti Transport come astratti. Il client sa che tutti gli oggetti di Transport dovrebbero avere il metodo deliver(), ma per il client non è importante conoscere i dettagli implementativi.

Struttura :



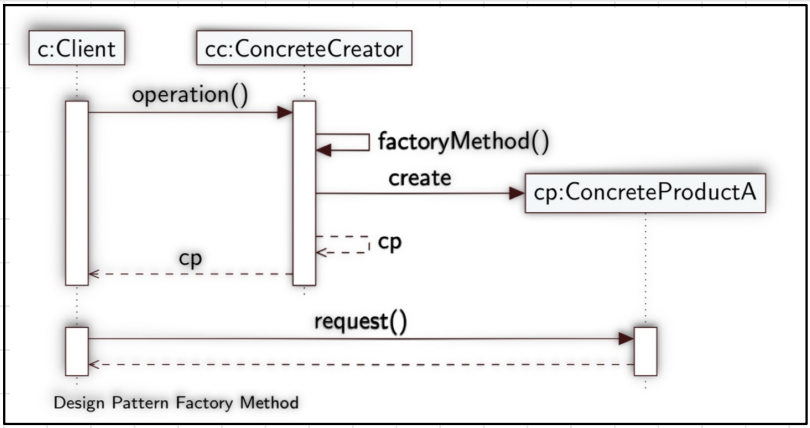
1) **Product** dichiara l'interfaccia, che è comune a tutti gli oggetti che possono essere prodotti dal creatore e dalle sue sottoclassi.

2) **Concrete Product** sono diverse implementazioni dell'interfaccia del prodotto.

3) La classe **Creator** dichiara il metodo factory che restituisce nuovi oggetti prodotto. È importante che il tipo restituito di questo metodo corrisponda all'interfaccia del prodotto. È possibile dichiarare abstract il metodo factory in modo da forzare tutte le sottoclassi a implementare le proprie versioni del metodo. In alternativa, il metodo base factory può restituire un tipo di product predefinito.
Nonostante il nome, la creazione del prodotto non è la responsabilità principale del creator. Di solito, la classe creator ha già una logica aziendale di base relativa ai prodotti. Il metodo factory aiuta a disaccoppiare questa logica dalle classi di prodotto concrete. Ecco un'analogia: una grande azienda di sviluppo software può avere un dipartimento di formazione per programmatori. Tuttavia, la funzione principale dell'azienda nel suo insieme è ancora scrivere codice, non produrre programmatori.

4) **Concrete Creators** esegue l'override del metodo factory di base in modo che restituisca un diverso tipo di prodotto. Si noti che il metodo factory non deve creare sempre nuove istanze. Può anche restituire oggetti esistenti da una cache, un pool di oggetti o un'altra origine.

UML di Sequenza



Applicabilità:

- Usa il Factory method quando non conosci in anticipo i tipi esatti e le dipendenze degli oggetti con cui il tuo codice dovrebbe funzionare.
- Utilizzare il metodo Factory quando si desidera fornire agli utenti della libreria o del framework un modo per estenderne i componenti interni.
- Utilizzare il Factory method quando si desidera risparmiare risorse di sistema riutilizzando oggetti esistenti invece di ricostruirli ogni volta.

Implementazione:

1) Fai in modo che tutti i prodotti seguano la stessa interfaccia. Questa interfaccia dovrebbe dichiarare metodi che hanno senso in ogni prodotto.

2) Aggiungi un metodo factory vuoto all'interno della classe creatore. Il tipo restituito del metodo deve corrispondere all'interfaccia del prodotto comune.

3) Nel codice del creatore trova tutti i riferimenti ai costruttori di prodotti. Uno per uno, sostituiscili con chiamate al metodo factory, mentre estrai il codice di creazione del prodotto nel metodo factory.
Potrebbe essere necessario aggiungere un parametro temporaneo al metodo factory per controllare il tipo di prodotto restituito. A questo punto, il codice del metodo factory può sembrare piuttosto brutto. Potrebbe avere una dichiarazione switch di grandi dimensioni che sceglie quale classe di prodotto istanziare. Ma non preoccuparti, lo ripareremo abbastanza presto.

4) Ora, crea un set di sottoclassi dei creatori per ogni tipo di prodotto elencato nel metodo factory. Sovrascrivi il metodo factory nelle sottoclassi ed estrai i bit appropriati del codice di costruzione dal metodo base.

5) Se sono presenti troppi tipi di prodotto e non ha senso creare sottoclassi per tutti, è possibile riutilizzare il parametro di controllo dalla classe base nelle sottoclassi.

6) Se, dopo tutte le estrazioni, il metodo base factory è diventato vuoto, puoi renderlo astratto. Se è rimasto qualcosa, puoi renderlo un comportamento predefinito del metodo

Vantaggi :

- Eviti l'accoppiamento stretto tra il creatore e i prodotti concreti.
- Principio di responsabilità unica . Puoi spostare il codice di creazione del prodotto in un punto del programma, semplificando il supporto del codice.
- Principio open/close . È possibile introdurre nuovi tipi di prodotti nel programma senza violare il codice client esistente.

Svantaggi :

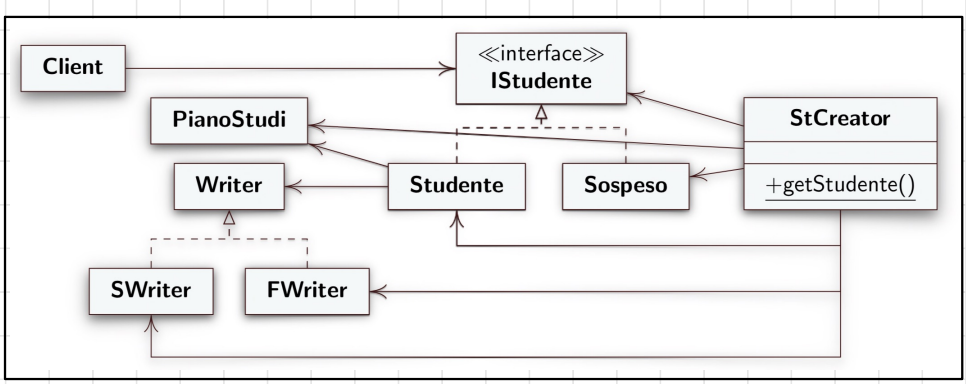
- Il codice potrebbe diventare più complicato poiché è necessario introdurre molte nuove sottoclassi per implementare il modello. Lo scenario migliore è quando si introduce il modello in una gerarchia esistente di classi di creatori.

Dependency Injection

Il design pattern Factory Method può essere usato per inserire facilmente dipendenze necessarie alle istanze ConcreteProduct. Quando si utilizza il Factory Method insieme alla DI, si definisce un'interfaccia comune per l'oggetto da creare e si delega la scelta della classe concreta da utilizzare a un componente esterno, il cosiddetto "factory". In questo modo, il componente che utilizza l'oggetto non ha la responsabilità di conoscere le dipendenze dell'oggetto stesso e il processo di creazione viene separato dalla logica del programma.

Ad esempio, supponiamo di avere una classe Car che ha bisogno di un oggetto Engine per funzionare. Invece di creare l'oggetto Engine all'interno della classe Car, si può definire un'interfaccia comune EngineInterface e utilizzare un Factory Method per creare un'istanza di Engine quando necessario. Il Factory Method può essere implementato da una classe esterna chiamata CarFactory che conosce tutte le dipendenze necessarie per creare un'istanza di Car, inclusa l'istanza di Engine. In questo modo, la classe Car può essere costruita utilizzando l'istanza di Engine fornita dal Factory, senza preoccuparsi di come viene creato l'oggetto.

L'utilizzo della Dependency Injection insieme al Factory Method consente quindi di separare la logica di creazione degli oggetti dalla logica del programma, semplificando la manutenzione e il testing del codice.



Ci sono molti altri tipi di design pattern creazionali, ma in generale, tutti hanno l'obiettivo di separare la creazione degli oggetti dal loro utilizzo, migliorando così la flessibilità e la manutenibilità del codice.

I design pattern creazionali includono, ad esempio, il Singleton, l'Abstract Factory, il Builder, il Factory Method e il Prototype.

Il Singleton è un pattern che garantisce che una sola istanza di una classe esista durante l'intera vita dell'applicazione. L'Abstract Factory permette di creare famiglie di oggetti correlati senza specificare le loro classi concrete. Il Builder consente di creare oggetti complessi passo dopo passo, separando la loro costruzione dalla loro rappresentazione finale. Il Factory Method definisce un'interfaccia per creare oggetti, ma lascia alle sottoclassi la scelta delle classi concrete da istanziare. Infine, il Prototype permette di creare nuovi oggetti duplicando un'istanza esistente.