22. Design Pattern

Paola Barra
2022/2023

I design pattern

Con i programmi informatici determinati processi tendono sempre a ripetersi, per questo è nata l'idea di creare dei modelli.

Questi schemi progettuali (chiamati design pattern) possono **semplificare il lavoro di programmazione**.

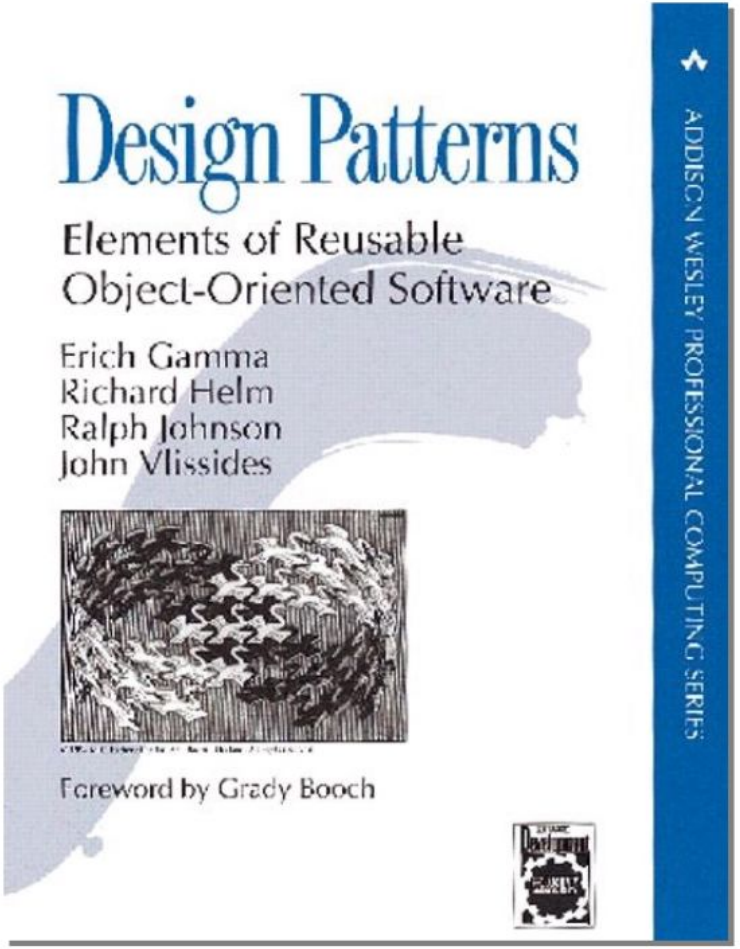
Definizione di design pattern

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

  - -- Christopher Alexander A Pattern Language, 1977

Cosa sono i design pattern?

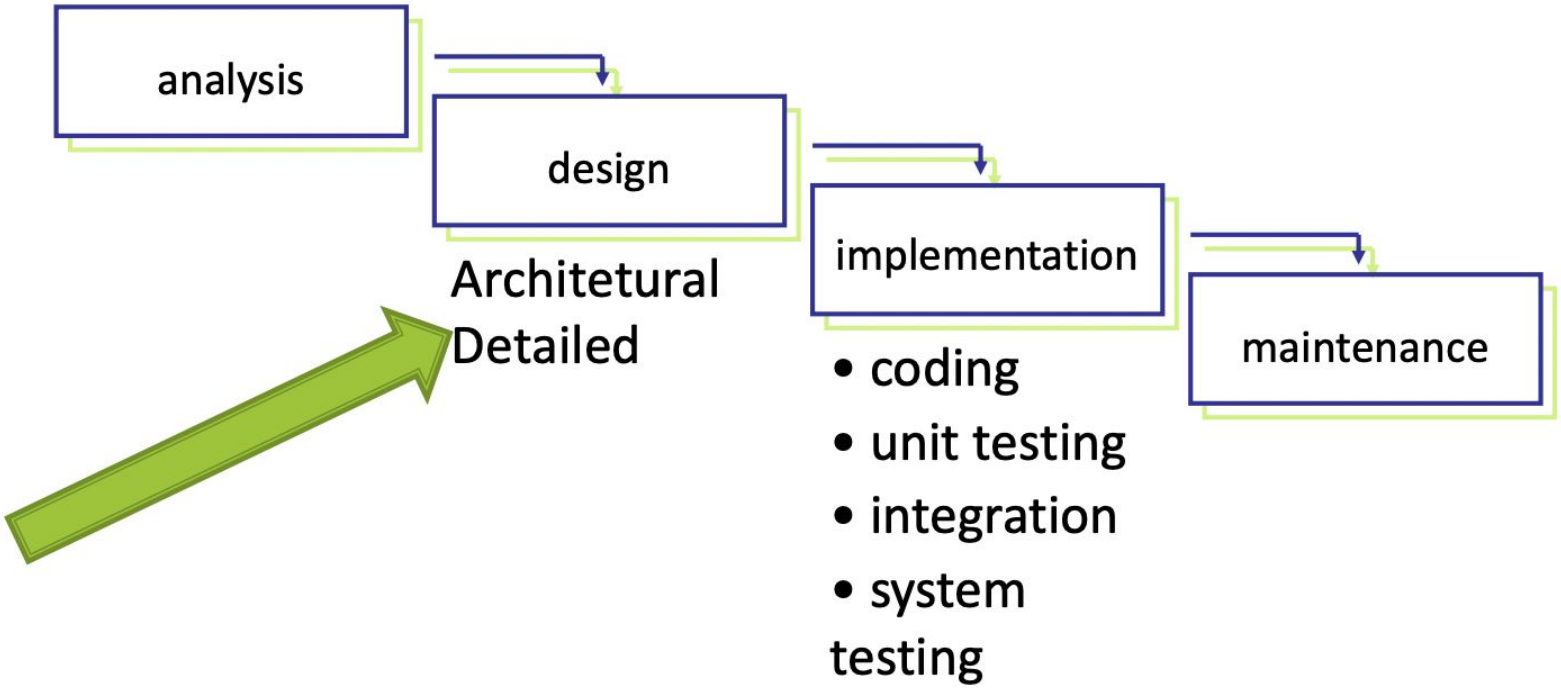
Il termine “design pattern” è da attribuirsi all’architetto statunitense Christopher Alexander, che mise insieme in una raccolta tutti i modelli **riutilizzabili**. Il suo obiettivo era **coinvolgere** gli utenti futuri di edifici **nella fase di progettazione.** Quest’idea è stata poi ripresa da diversi informatici. La cosiddetta Gang of Four (GoF), composta da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides ha rappresentato un punto di svolta nei **software design pattern** con il libro “Design Patterns – Elements of Reusable Object-Oriented Software” (it. “Design Patterns: Elementi per il riuso di software ad oggetti) pubblicato nel 1994.

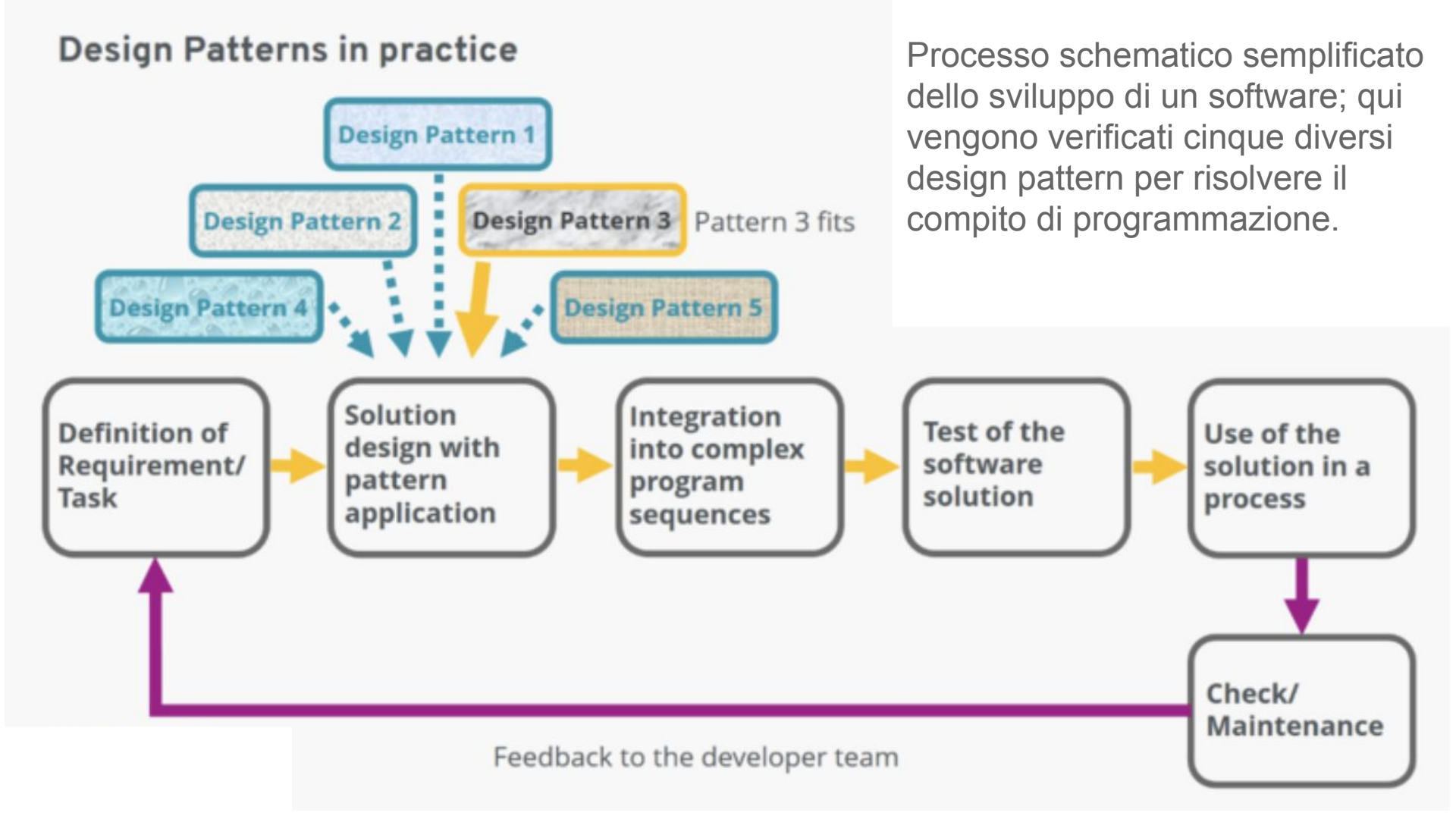


Definizione data da Christopher Alexander

- C. Alexander ha definito i design patterns studiando tecniche per migliorare il processo di progettazione di edifici e aree urbane
- Ogni pattern è una regola in tre parti, che esprime una relazione tra
  - Un contesto
  - Un problema
  - Una soluzione
- DEF: "una soluzione a un problema in un contesto"
- I pattern possono essere applicati a diverse aree, compreso lo sviluppo software

In che fase si applicano





Processo schematico semplificato dello sviluppo di un software; qui vengono verificati cinque diversi design pattern per risolvere il compito di programmazione.

Che tipo di design pattern esistono?

Sono 23 design pattern suddivisi in base al loro scopo.

I tipi di schemi progettuali rappresentano i principali **campi di applicazione** dei software design pattern in essi assemblati.

Modelli strutturali

Modelli comportamentali

Modelli creazionali

Nel corso degli anni si sono aggiunti altri **tipi di schemi progettuali,** che non rientrano in nessuna delle tre categorie citate. Di questi fanno parte i modelli di **mappatura relazionali a oggetti**, per memorizzare oggetti e relativi legami in una banca dati relazionale.

Pro e contro relativi all'utilizzo dei design patter

VANTAGGI:

La possibilità di attingere a **soluzioni valide** va di pari passo con un **risparmio di tempo e costi**. I team di sviluppatori non devono costantemente ricominciare da zero per trovare una soluzione per un nuovo programma in caso di problemi già parzialmente risolti.

Di solito, i singoli schemi vengono denominati in base a un **vocabolario comune di termini tecnici,** semplificando in questo modo sia la discussione tra sviluppatori, sia la comunicazione con l'utente destinatario della soluzione finale. Si **semplifica** anche la **documentazione** di un software, considerato che possono essere utilizzate componenti già documentate in precedenza. Tutti questi vantaggi aiutano nella fase di manutenzione e ulteriore sviluppo di un programma.

Pro e contro relativi all’utilizzo dei design patter

SVANTAGGI:

L’utilizzo degli schemi progettuali richiede un **ampio bagaglio di conoscenze** pregresse. Inoltre,la disponibilità di design pattern può anche far credere che gli schemi progettuali presenti possano risolvere quasi tutti i problemi. Detto in parole povere: **questo può limitare la creatività** e la curiosità di trovare soluzioni nuove (e migliori).

Modelli creazionali

I **creational patterns** permettono di creare oggetti che rappresentano in modo semplificato istanze precise, indipendentemente dal modo in cui i singoli oggetti sono creati e rappresentati in un software.
Propongono soluzioni per creare oggetti.

- **Builder Pattern: i**l **costruttore** nella categoria dei pattern creazionali separa lo sviluppo di oggetti (complessi) dalle loro rappresentazioni.
- **Factory Pattern:** come pattern, il **factory method** crea un oggetto attraverso il richiamo ad un metodo invece che a un costruttore.
- **Singleton Pattern:** come pattern, il **singleton** fa in modo che per ogni classe esista solo un'unica istanza. Per di più un singleton è disponibile a livello globale.

Modelli strutturali

I cosiddetti **structural patterns** sono degli schemi preimpostati per i legami tra le classi. Con essi si mira a un'astrazione che possa comunicare anche con altre soluzioni; la parola chiave è programmazione di interfacce.
Propongono soluzioni per la composizione strutturale di classi e oggetti.

- **Composite Pattern:** un pattern strutturale composto, chiamato in inglese **composite**, rivolto principalmente alle strutture dinamiche, per es. per l'organizzazione o la compressione di dati.
- **Decorator Pattern:** il cosiddetto **decorator** integra le classi esistenti di ulteriori funzionalità o responsabilità.
- **Facade Pattern:** il modello **facciata** rappresenta un'interfaccia verso altri sistemi, sottosistemi o subsistemi.

Modelli comportamentali

Con i **behavioral pattern** si modella il comportamento del software. Questi pattern semplificano processi complessi di comando e controllo. Si può scegliere tra algoritmi e responsabilità di oggetti.
Propongono soluzioni per gestire il modo in cui vengono suddivise le responsabilità delle classi e degli oggetti.

- **Observer Pattern:** l'**osservatore** trasmette alle strutture le modifiche apportate a un oggetto, che dipendono dall'oggetto iniziale.
- **Strategy Pattern:** la **strategia** definisce una famiglia di algoritmi intercambiabili.
- **Visitor Pattern:** il **visitatore** permette di isolare le operazioni eseguibili, in modo da compiere nuove operazioni senza modificare le classi interessate.

Modelli creazionali: Builder Pattern

Modelli creazionali : Builder pattern

Il pattern Builder fa parte dei design pattern. Si tratta di modelli ben collaudati che facilitano il lavoro di **programmazione orientata agli oggetti**, in quanto permettono agli sviluppatori di non dover ripetere ogni volta i passaggi che si ripetono, dando loro la possibilità di usare una soluzione già definita. Questi elementi software risalgono al libro pubblicato nel 1994 *Design Patterns: elementi per il riuso di software a oggetti* dei quattro sviluppatori di software statunitensi conosciuti come “la banda dei quattro” (in inglese “Gang of Four”, abbreviato in GoF).

Nel dettaglio: Il builder fa parte del gruppo dei modelli di creazione conosciuto come design pattern. **Migliora sia la sicurezza** durante il processo di costruzione **che la leggibilità del codice** del programma. L’obiettivo dei modelli builder è di permettere la costruzione di un oggetto utilizzando una classe helper invece che con i soliti costruttori.

Modelli creazionali : Builder pattern

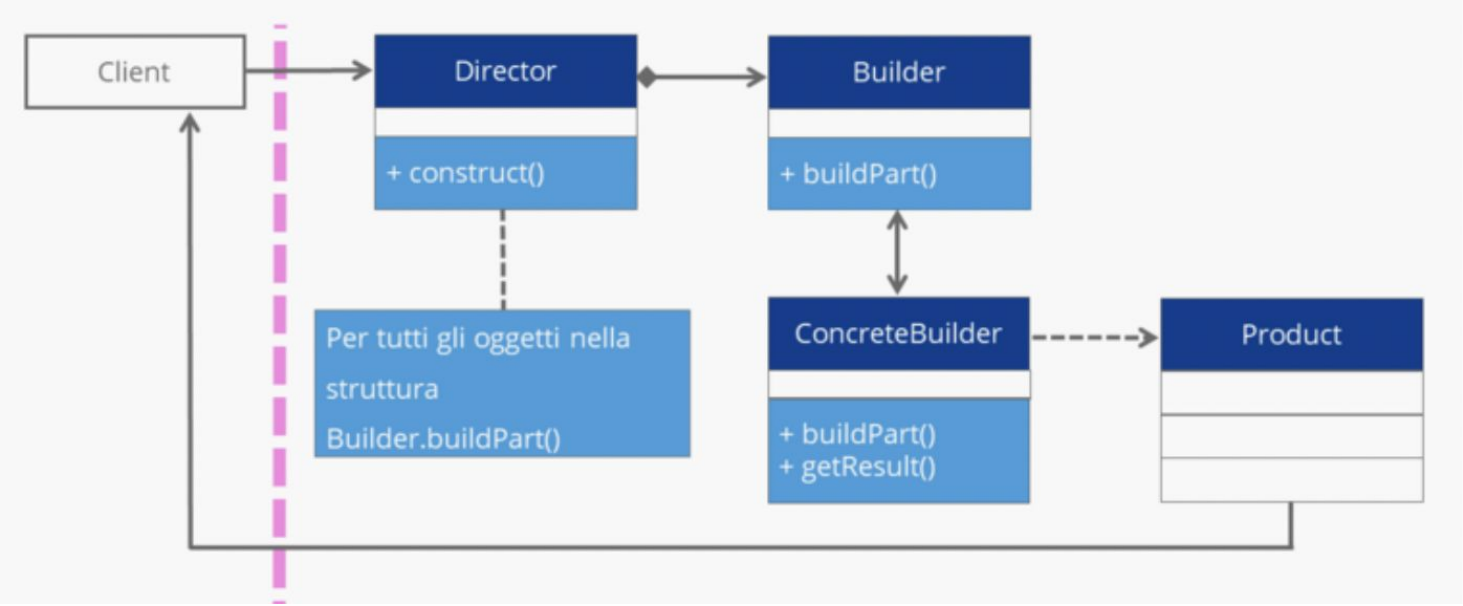
Nel builder design pattern si distinguono **quattro diversi attori**:

- **Director**: questo attore costruisce l'oggetto complesso utilizzando l'interfaccia del costruttore. Conosce i requisiti della sequenza di lavoro del costruttore. È al livello del direttore che la costruzione di un oggetto viene disaccoppiata dal client.
- **Builder**: il costruttore mette a disposizione un'interfaccia per la costruzione delle componenti di un oggetto complesso (il prodotto).
- **ConcreteBuilder**: questo attore si occupa dell'effettiva creazione delle parti dell'oggetto complesso, definendo e gestendo la rappresentazione dell'oggetto, disponendo anche di un'interfaccia per l'output dell'oggetto.
- **Product**: il risultato dell'attività del builder pattern, ossia l'oggetto complesso da costruire.

Tuttavia, il passaggio decisivo di questo pattern avviene a livello del direttore, dove la **creazione di un oggetto/prodotto viene separata dal client**

Il builder pattern nella rappresentazione UML

Per la rappresentazione grafica dei processi di programmazione viene usato il linguaggio di modellazione, più comunemente chiamato è UML. La grafica qui sotto mostra come il builder pattern sia **composto da numerosi oggetti che interagiscono tra loro**. Struttura basilare di un builder pattern in UML, utile a illustrare le molteplici relazioni al suo interno. L'utente dell'oggetto creato è completamente dissociato da questa creazione.



Pro e contro del builder pattern

VANTAGGI DEL BUILDER

La costruzione o creazione **e la rappresentazione** (output) **vengono isolate**. Le rappresentazioni interne del costruttore vengono “nascoste” dal director. Nuove rappresentazioni possono essere facilmente aggiunte grazie a nuove classi di costruzioni concrete. Il processo di costruzione viene esplicitamente gestito dal director. Nel caso in cui si debbano applicare delle modifiche lo si può fare senza dover passare per i client.

SVANTAGGI DEL BUILDER

Data la stretta connessione tra prodotto, ConcreteBuilder e le classi coinvolte nel processo di costruzione, **può risultare difficile apportare modifiche al processo**. La creazione degli oggetti richiede spesso una conoscenza delle specifiche applicazioni e del loro ambiente. L’utilizzo di pattern conosciuti e del builder può portare i programmatori a non accorgersi di soluzioni più semplici e magari anche più eleganti. Il builder pattern è perciò uno dei design pattern meno importanti tra i programmatori.

Dove viene impiegato il builder pattern ?

Per rendere più intuitiva la spiegazione del builder pattern lo possiamo comparare alle dinamiche semplificate di un ristorante, dove un cliente fa la propria ordinazione. Lo staff del ristorante, che qui corrisponde ai vari attori, agisce alla ricezione dell'ordine al fine di servire al tavolo quanto richiesto. L'intero processo fino all'arrivo al tavolo del cibo ordinato da parte del cliente avviene dietro le quinte. Il cliente non vede cosa accade in cucina in seguito alla sua ordinazione, ma ne riceve solamente il risultato ultimo servito al tavolo. Nel linguaggio di programmazione questo viene comunemente denominato "print".

L'ordinazione di un cliente al ristorante è separato dalla preparazione delle pietanze. Il director e il costruttore si occupano della realizzazione del prodotto con l'aiuto del modello di costruzione builder

Modelli creazionali: Factory Pattern

Modelli creazionali : Factory pattern

Tra le numerose strategie di design si annovera anche il cosiddetto factory method (“metodo di fabbrica”), detto anche factory pattern, che permette a una classe di **delegare la creazione di oggetti alle sottoclassi**.

Il factory method pattern descrive un approccio di programmazione con il quale **creare oggetti senza bisogno di dover specificare la loro classe**. Questo permette di cambiare comodamente e in maniera flessibile l’oggetto creato. Lo sviluppatore sceglie se specificare il factory method in un’interfaccia e quindi implementarlo come classe figlio o come classe base ed eventualmente sovrascriverlo dalle classi derivate. Questo metodo **opera a livello della classe costruttore standard** per separare la costruzione degli oggetti dagli oggetti stessi e permettere così l’utilizzo dei principi SOLID.

Si parla quindi di factory method ma anche di **factory pattern o factory design pattern**, nonostante queste versioni non vengano mai menzionate nell’opera della GoF. Oltre al già menzionato factory method pattern, sfogliando il libro trovate solamente l’abstract factory pattern, utile a definire l’interfaccia per la creazione di una famiglia di oggetti le cui classi concrete vengono definite soltanto durante l’esecuzione.

I principi SOLID sono una parte dei principi del design orientato agli oggetti, che hanno lo scopo di migliorare i processi di sviluppo dei software orientati agli oggetti.
L'acronimo "SOLID" sta per i seguenti cinque principi:

- Single responsibility principle: ogni classe deve possedere un'unica responsabilità.

- Open/closed principle: le unità del software devono essere estendibili senza modificarne il comportamento.

- Liskov substitution principle: una classe derivata deve sempre poter essere sostituita dalla sua classe base.

- Interface segregation principle: le interfacce devono corrispondere perfettamente alle richieste dei client che effettuano l'accesso.

- Dependency inversion principle: le classi con un livello di astrazione elevato non devono mai dipendere da classi con un livello di astrazione più basso.

Qual è lo scopo del factory design pattern?

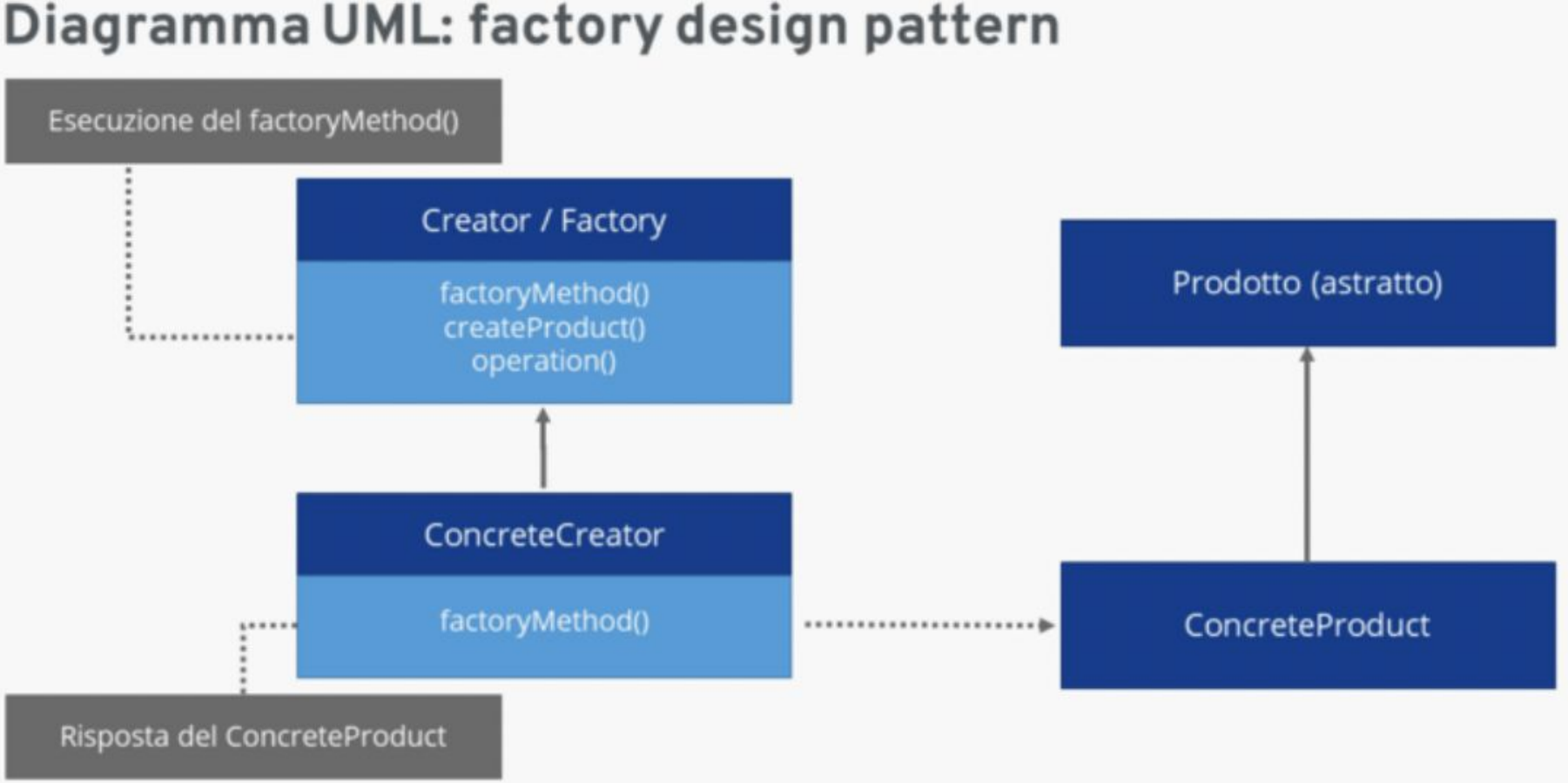
Il factory pattern serve a risolvere un problema di fondo durante l'istanziazione, ossia la creazione di un oggetto concreto di una classe, nell'ambito della programmazione orientata agli oggetti.

Creare un oggetto direttamente all'interno della classe di cui l'oggetto ha bisogno, è possibile ma molto poco flessibile. Infatti, la classe viene collegata all'oggetto preciso, rendendo impossibile modificare l'istanziazione, indipendentemente dalla classe. Un codice di questo tipo può essere evitato grazie al factory pattern che **definisce un'operazione separata per la creazione dell'oggetto** – il cosiddetto metodo factory – sostituendo la classe costruttore nella creazione dell'oggetto.

Factory pattern:diagramma UML del modello Factory

Nei software adatti all'impiego del factory design pattern, il **codice dell'oggetto da creare** (definito in queste circostanze anche come "prodotto") **viene esternalizzato** separatamente in una propria classe. Questa classe astratta chiamata anche "creator" o "factory" delega l'istanziazione dell'oggetto a una **sottoclasse** (ConcreteCreator), la quale decide infine che prodotto creare. A questo scopo il **ConcreteCreator** adopera il metodo *createProduct()* e fornisce in risposta un **ConcreteProduct**, ampliabile dal Creator utilizzando un codice di creazione, prima che il prodotto finito venga trasmesso all'interfaccia.

Il seguente diagramma di classe UML aiuta a chiarire il funzionamento del factory pattern, **riassumendo graficamente** le relazioni e i processi descritti.



Rappresentazione grafica del factory pattern in UML

Vantaggi e svantaggi del factory design pattern

VANTAGGI:

Con il factory pattern l'esecuzione di un metodo di programma è completamente separata dall'implementazione di nuove classi, il che porta con sé alcuni vantaggi. Questo ha un effetto particolarmente positivo sull'**ampliabilità del software**: le istanze factory possiedono un elevato grado di autonomia e permettono l'**aggiunta di nuove classi durante l'esecuzione** senza alterare in alcun modo l'applicazione. È sufficiente implementare le interfacce factory e successivamente istanziare il Creator (passando per il ConcreteCreator).

Un ulteriore vantaggio consiste nella **buona testabilità delle componenti factory**. Se, ad esempio, un Creator implementa tre classi, la loro **funzionalità può essere testata singolarmente** e indipendentemente dalla classe in esecuzione. Per farlo, è sufficiente assicurarsi che le classi richiamino correttamente il Creator, anche nel caso in cui venissero apportate delle modifiche al software in un secondo momento. Altrettanto vantaggiosa è la possibilità di assegnare ai metodi di fabbrica un nome significativo, possibilità non concessa alla classe costruttore.

Vantaggi e svantaggi del factory design pattern

SVANTAGGI:

Il grande punto debole del factory design pattern è che la sua applicazione porta a un **aumento delle classi coinvolte**, in quanto ogni ConcreteProduct richiede sempre anche un ConcreteCreator. L'approccio factory è vantaggioso per la possibilità di espansione del software, ma presenta degli svantaggi per quanto riguarda l'impegno necessario. Se deve essere aggiunta una famiglia di prodotti, non deve essere adeguata solamente l'interfaccia, ma anche tutte le classi di ConcreteCreator sottostanti. Risulta perciò **irrinunciabile una buona pianificazione** sulla base del tipo di prodotto desiderato.

Vantaggi	Svantaggi
Estensione modulare dell'applicazione	Elevato numero di classi richieste
Buona testabilità	L'ampliamento dell'applicazione è molto impegnativo
Possibilità di assegnare nomi significativi ai metodi	

Dove viene impiegato il factory method pattern?

Il factory pattern trova applicazione in diversi contesti, in modo particolare, nei software in cui i **prodotti concreti da creare non sono conosciuti** o ben definiti in precedenza, il suo approccio alternativo per la gestione delle sottoclassi risulta particolarmente vantaggioso. Alcuni esempi classici di utilizzo sono **i framework o le librerie di classe**, diventate l'architettura di base pressoché irrinunciabile per lo sviluppo delle moderne applicazioni.

Anche i sistemi di autenticazione godono dei vantaggi offerti dai factory design pattern. Al posto di una classe centrale con diversi parametri che variano a seconda dei **permessi utente**, con i factory pattern si può delegare il processo di autenticazione alle classi factory, capaci di prendere decisioni autonomamente sulla gestione dei vari utenti.

Inoltre, un design dotato dell'approccio factory pattern è generalmente adatto a qualsiasi software nel quale vengono **aggiunte regolarmente nuove classi secondo pianificazione**. Soprattutto se queste classi devono eseguire lo stesso processo di creazione.

Modelli creazionali: Singleton Pattern

Modelli creazionali : Singleton pattern

I cosiddetti design pattern mettono a disposizione degli sviluppatori molteplici modelli comprovati in grado di risolvere i passaggi più ostici della programmazione orientata agli oggetti. Una volta trovato il modello di design che fa al caso proprio tra i circa settanta a disposizione, bastano alcuni accorgimenti per un risultato ottimale. L'approccio di base di ciascun modello rimane però sempre lo stesso. Il singleton pattern o **modello di design singleton,** sebbene **molto performante,** è reputato ad oggi alquanto obsoleto nell'ambito della programmazione orientata agli oggetti. In questo articolo vi spieghiamo quello che permette di fare, quando e come impiegarlo.

Il singleton pattern appartiene alla categoria dei modelli creazionali ed è uno dei pattern più semplici ma anche più potenti per lo sviluppo software. Il suo compito consiste nell'**impedire che da una classe possa essere creato più di un oggetto**. Per farlo, l'oggetto desiderato viene creato all'interno di una classe per poi essere invocato come istanza statica.

> **Citazione**
>
> Il team di programmatori americano "Gang of Four" (GoF) ha dichiarato quanto segue riguardo al singleton pattern: "Assicuratevi che una classe possegga esattamente un'istanza e che la renda accessibile globalmente."

Quali sono le caratteristiche del singleton pattern?

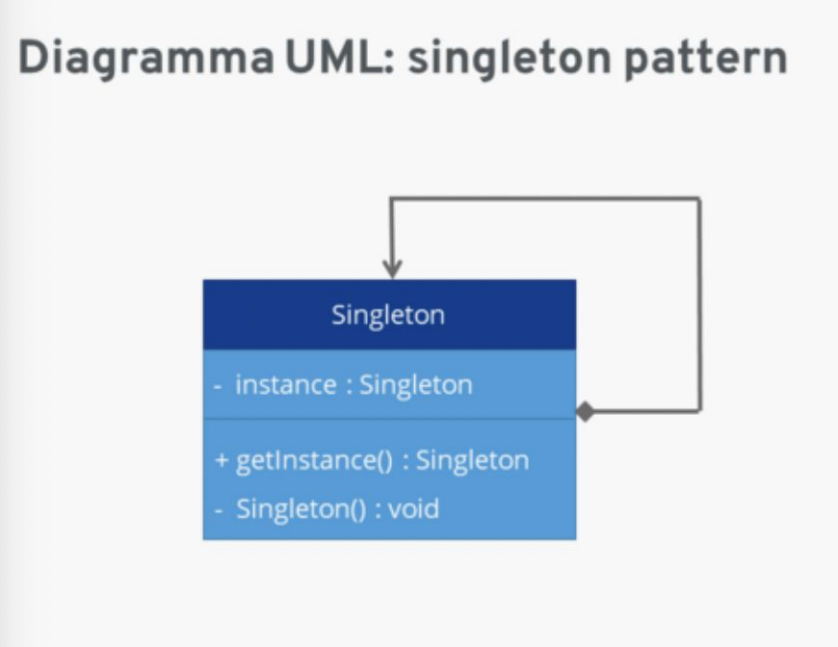
Creando un'istanza da una classe con il singleton design pattern ci si assicura che non ne vengano create delle altre. Il singleton **rende questa classe accessibileglobalmente** all'interno del software, tramite uno dei vari metodi disponibili nei linguaggi di programmazione. Per assicurarsi che l'istanza creata rimanga l'unica, bisogna impedire che l'utente possa crearne di nuove. Il costruttore deve quindi **dichiarare il modello "private"**. In questo modo solamente il codice contenuto nel singleton può istanziare lo stesso singleton. Questo garantisce che l'utente ottenga un solo e unico oggetto, sempre lo stesso. Quando questa istanza viene resa disponibile, non ne vengono create di nuove. Un singleton si presenta così:

```
public class Singleton {

    private static Singleton instance; // statico e protetto da accesso esterno

    private Singleton() {} // costruttore privato con protezione da accesso esterno

    public static Singleton getInstance() { // metodo aperto, invocazione tramite codice

    if (instance == null) { // solamente quando non esiste alcuna istanza, ne crea una nuova


        instance = new Singleton();
        }
        return instance;
    } }
```

Il singleton pattern rappresentato in UML

La **rappresentazione dell'intero singleton design pattern** con il linguaggio di modellazione unificato, abbreviato in UML, consiste di un unico oggetto, in quanto si tratta della creazione di un'unica istanza di una classe.

Dall'esterno non è possibile apportare modifiche all'unica istanza generata. È proprio questa la finalità d'utilizzo del singleton design pattern.



Vantaggi e svantaggi del modello di design singleton

VANTAGGI

Scrivere un singleton è un'**operazione veloce e poco complicata**, in quanto non consta di un numero elevato di variabili (globali). Incapsula la propria creazione così da poter **esercitare pieno controllo** su quando e come sia possibile accedervi. Un singleton pattern esistente può essere declinato per mezzo di sottoclassi per eseguire nuove funzionalità. Cosa utilizzare viene deciso in maniera dinamica.

Non di secondaria importanza è il fatto che un singleton venga creato solamente quando necessario. Questa caratteristica prende il nome di lazy loading. **Eager loading**, al contrario, definisce il processo di istanziare un singleton in maniera anticipata, ossia quando non ancora necessario.

SVANTAGGI

L'uso sconsiderato dei singleton porta a uno stato di programmazione procedurale, quindi non più orientata agli oggetti, e a un codice di programmazione che possiamo definire **"sporco"**. La disponibilità globale dei modelli di design singleton cela dei pericoli nel caso in cui serva a trattare dati sensibili. Quando vengono apportate modifiche al singleton non è dato sapere quali componenti del programma siano interessate. Questo **complica la manutenzione del software**, poiché risulta difficile risalire ai malfunzionamenti.

La disponibilità globale rende altrettanto complicata l'eliminazione dei singleton, in quanto alcune componenti del software possono riferirsi a questo. Nel caso di applicazioni con molti utenti (applicazioni multiutente), un singleton può avere l'effetto di **diminuire le prestazioni**, in quanto non permette un flusso regolare dell'intera mole di dati, creando un'impasse.

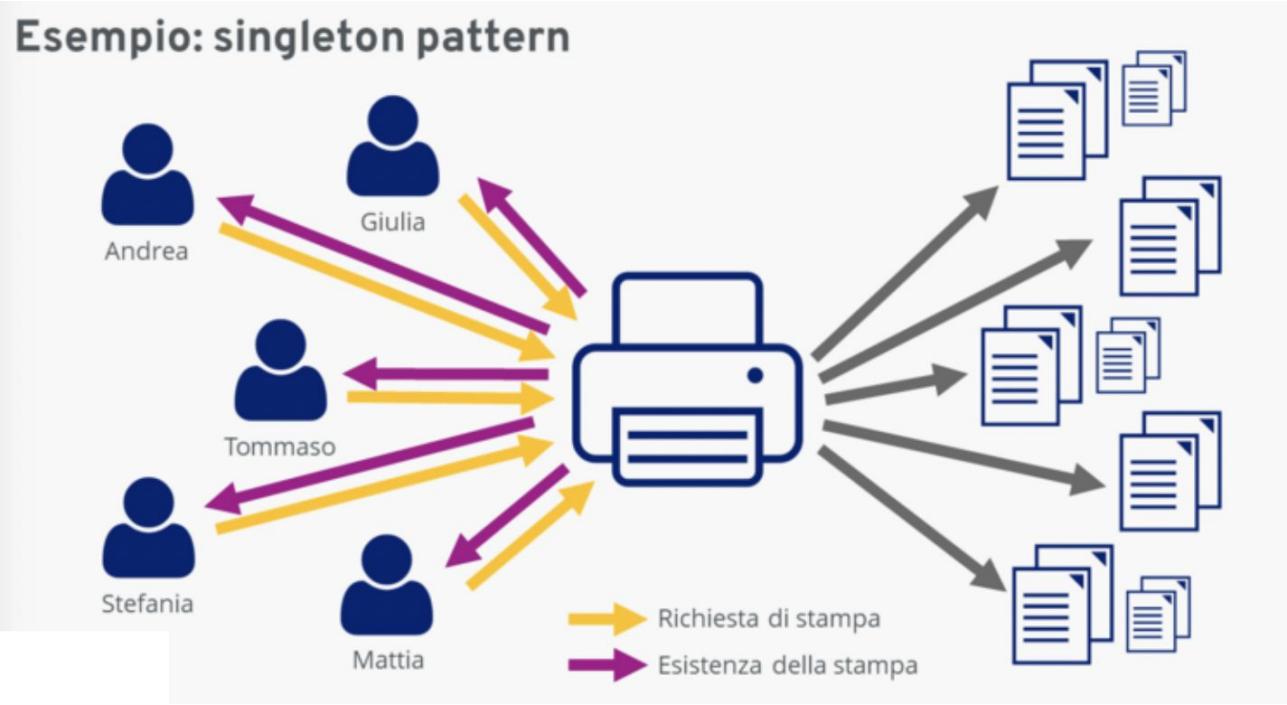
Il modello di design singleton nella realtà

Il singleton viene utilizzato spesso quando risulta necessario risolvere **processi molto ripetitivi** nella routine di un programma. Ad esempio: per scrivere dati all'interno di un file, nel caso si tratti di **file di registro o per ordini di stampa** che devono essere eseguiti nello stesso buffer. Dato che anche i **driver e i meccanismi di cache** usano solitamente gli stessi processi, anche in questo caso il singleton design pattern trova spesso impiego.

Data la difficoltà di testare un singleton pattern, vi illustriamo qui di seguito **il suo funzionamento tramite l'esempio di una piccola azienda** nella quale numerosi dipendenti utilizzano la stessa stampante.

Nel software di gestione della stampante viene adoperato un singleton che serve a elaborare le richieste di stampa senza alterarle ed eseguire l'operazione di stampa.

Quando un utente invia una richiesta alla stampante, il singleton avanza questa domanda: "esiste già un oggetto stampante? Se la risposta è no, allora deve essere creato". Per farlo si utilizza un'indicazione di carattere if/then (*returnStampante* == *Null ?*). Inoltre, per **evitare accessi e modifiche indesiderati** vengono impiegate delle variabili specifiche e la stampante impostata come "private" (privata).



Modelli strutturali: Composite Pattern

Modelli strutturali: Composite pattern

Le strutture di dati dinamiche come le strutture ad albero di un file management o di un’interfaccia di programma richiedono una **struttura gerarchica chiara** e il più possibile inequivocabile. L’implementazione di tali strutture spesso non è per niente semplice. Ad esempio, è importante assicurarsi che il tipo di oggetto non debba essere richiamato ogni volta prima dell’effettiva elaborazione dei dati, poiché un tale scenario non sarebbe né efficiente né performante. Soprattutto nel caso in cui molti oggetti primitivi incontrino oggetti compositi, si consiglia l’utilizzo del cosiddetto Composite design pattern (in italiano “modello di progettazione composito”). L’approccio di progettazione del software consente ai client di trattare gli oggetti individuali e compositi in modo uniforme, rendendo le differenze ininfluenti per il client.

Che cos'è il Composite pattern

Il Composite design pattern, abbreviato in Composite pattern, è uno dei 23 design pattern per lo sviluppo software che sono stati rilasciati nel 1994 da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (noti anche come "**G**ang of **F**our"). Come il Facade pattern e il Decorator pattern, il Composite design pattern si annovera tra i modelli strutturali la cui funzione fondamentale è quella di **riunire oggetti e classi in strutture di maggiori dimensioni**.

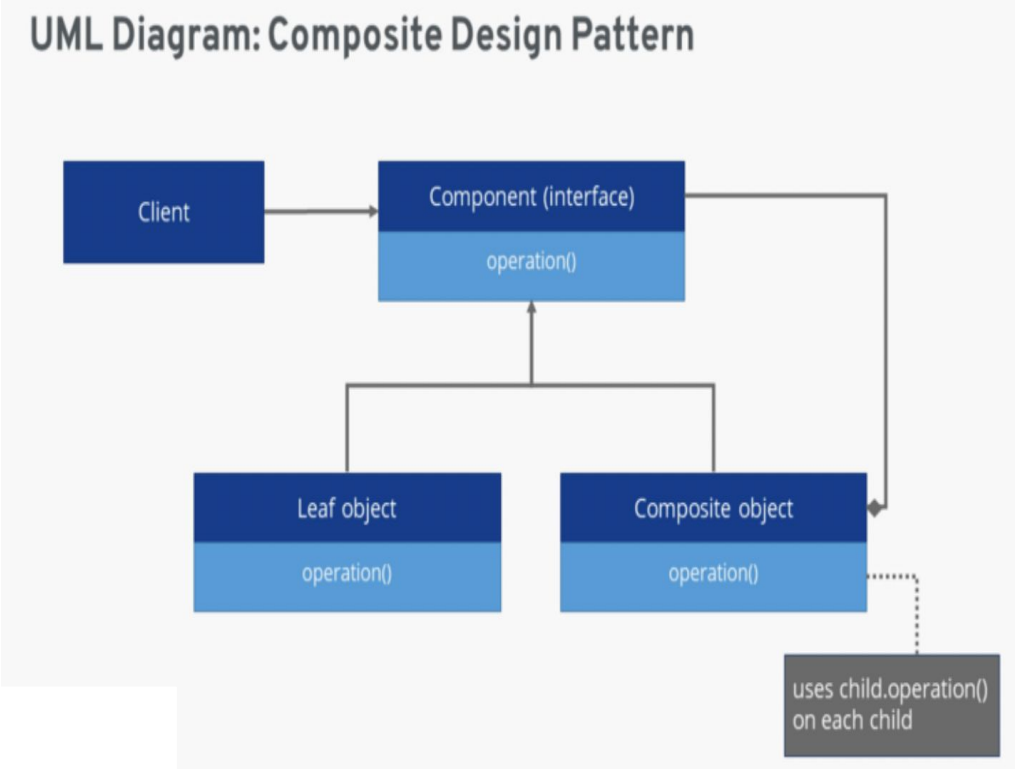
Il fine principale del Composite design pattern è, come in tutti i modelli della GoF, quello di **affrontare al meglio i problemi ricorrenti di progettazione nello sviluppo orientato agli oggetti**. Il risultato che si desidera ottenere è un software il più possibile flessibile che si contraddistingue per oggetti facili da implementare, testabili, sostituibili e riutilizzabili. A questo scopo, il Composite pattern descrive una possibile strada per trattare singoli oggetti e oggetti compositi alla stessa maniera. In questo modo si ottiene una struttura a oggetti che è **semplice da comprendere** e consente un **accesso di massima efficienza al client**. Inoltre viene minimizzata anche la suscettibilità del codice agli errori.

Composite design pattern: rappresentazione grafica (UML)

Per implementare le efficienti gerarchie parte-tutto già menzionate, il modello Composite prevede l'implementazione di **un'interfaccia dei componenti unitaria** per semplici oggetti-parte, chiamati anche **oggetti Leaf** (inglese per "foglia", ci si riferisce infatti alla metafora dell'albero dove le foglie sono una parte del tutto) e degli **oggetti Composite** (appunto compositi). I singoli oggetti Leaf integrano direttamente questa interfaccia, gli oggetti Composite trasmettono automaticamente le concrete **richieste del client all'interfaccia** ai loro componenti subordinati. Per il client non ha importanza con quale tipo di oggetto abbia a che fare (se parte o tutto), perché si tratta semplicemente di indirizzarsi all'interfaccia.

Il seguente diagramma di classe nel linguaggio di modellazione UML aiuta a chiarire i **rapporti e le gerarchie** in un software basato sul Composite pattern.

Il client si rivolge allo stesso modo agli oggetti Leaf e Composite (attraverso la stessa interfaccia). Successivamente gli oggetti Composite trasmettono le richieste ai propri elementi figlio.



Vantaggi e svantaggi del modello composite.

VANTAGGI

Il Composite pattern è una costante nello sviluppo del software. In particolare i **progetti construtture fortemente annidate** possono approfittare dell'approccio pratico per l'organizzazione di oggetti: non importa che si tratti di un oggetto primitivo o composito, con dipendenze semplici o complesse, perché la **profondità e larghezza degli annidamenti** è fondamentalmente **irrilevante** nel Composite design pattern. Il client può tranquillamente ignorare la differenza tra i tipi di oggetto e in questo modo non sono necessarie funzioni separate per l'accesso. In questo modo si ha il vantaggio che il codice del client rimane snello.

Un altro punto a favore del modello Composite è la **flessibilità** e l'**estensibilità** che il pattern conferisce a un software: l'interfaccia dei componenti universale consente di collegare nuovi oggetti Leaf e Composite senza variazioni del codice, sia dal lato client che nel caso in cui si tratti di strutture oggetti preesistenti.

Vantaggi e svantaggi del modello composite.

SVANTAGGI

Nonostante il Composite pattern e la sua interfaccia unitaria offrano numerosi vantaggi, questa soluzione non è esente da punti deboli: infatti, in particolare l'**interfaccia** può causare non poche difficoltà agli sviluppatori. Già l'**implementazione** pone grandi sfide per i responsabili, perché bisogna per esempio scegliere quali operazioni devono essere definite nell'interfaccia e quali nelle classi Composite. Inoltre eventuali **successive modifiche delle proprietà del Composite** (ad esempio le limitazioni, quali elementi figlio sono ammessi) si rivelano **complicati** e difficili da realizzare.

Vantaggi	Svantaggi
Ha tutto ciò che serve per rappresentare strutture oggetti fortemente annidate	L'implementazione dell'interfaccia dei componenti è molto complicata
Codice di programmazione snello e facilmente comprensibile	Le successive modifiche delle proprietà del Composite sono complicate e difficili da implementare
Buona estensibilità	

Scenari di applicazione per il composite pattern

L’utilizzo dei modelli Composite è particolarmente vantaggioso quando occorre effettuare operazioni su strutture di dati dinamici, le cui **gerarchie** sono di larghezza e/o profondità complesse. Si parla in questi casi anche di **struttura ad albero binaria**, che è interessante per i più svariati scenari software e viene utilizzata frequentemente. Alcuni tipici esempi sono:

Sistemi di file: i file system sono tra i componenti più importanti del software dei dispositivi. Con il Composite pattern si possono mappare in modo ottimale: i singoli **file** come oggetti Leaf e le **cartelle**, che a propria volta possono contenere file o ulteriori cartelle, come oggetti Composite.

Menu dei software: anche i menu dei programmi rappresentano un caso d’utilizzo tipico per una struttura ad albero binaria basata sul Composite design pattern. Nella **barra del menu** si trovano una o più voci di base (oggetti Composite) come “file”. Essi consentono l’accesso a varie voci del menu che sono direttamente **cliccabili** (Leaf) o contengono **ulteriori menu** (Composite).

Interfacce utente grafiche (GUI): le strutture ad albero e il modello Composite possono giocare un ruolo importante anche nella progettazione di interfacce utente grafiche. Accanto ai semplici elementi Leaf come pulsanti, campi di testo o caselle di controllo, ci sono i contenitori Composite, come frame o panel, che assicurano una struttura chiara e una maggiore comprensibilità.

Modelli strutturali: Decorator Pattern

Modelli strutturali: Decorator pattern

Se volete ampliare le classi già presenti in un software orientato agli oggetti con nuove funzionalità, avete due modi per farlo. La soluzione più facile, ma che può rapidamente complicarsi, riguarda l'implementazione di sottoclassi che estendano le classi base con ciò di cui avete bisogno. L'alternativa riguarda invece l'utilizzo di un'**istanza decoratore** secondo il cosiddetto decorator design pattern. Questo modello, facente parte dei 23 design pattern della GoF, permette l'**estensione dinamica delle classi durante l'esecuzione** del software. Senza, tra l'altro, che si venga a creare una gerarchia di ereditarietà infinita e di difficile comprensione.

Di seguito scoprite che cos'è il decorator pattern e quali vantaggi e svantaggi porta con sé. Inoltre, **illustriamo il funzionamento del modello** avvalendoci di una rappresentazione grafica e di un esempio concreto.

Il decorator design pattern o decorator pattern è un modello pubblicato nel 1994 per un'**estensione chiara delle classi** nei software orientati agli oggetti. Con questo modello è possibile aggiungere un comportamento desiderato senza influenzare il comportamento degli altri oggetti della stessa classe. Da un punto di vista strutturale, il decorator pattern ricorda molto il chain of responsibility pattern. Nonostante, diversamente da quest'ultimo dotato di un processore centrale, sono le classi a prendere in consegna le richieste.

La componente software che deve essere ampliata viene per così dire "decorata" con una o più **classi decorator**, che, secondo il principio del decorator pattern, la racchiudono. Ogni decorator è dello stesso tipo della **componente posta al suo interno** e dispone delle stesse interfacce. In questo modo può delegare senza problemi l'invocazione dei metodi, prima o dopo l'esecuzione del suo comportamento. In teoria anche l'elaborazione diretta di un'invocazione è possibile con il decorator.

A cosa serve il decorator design pattern?

Come per altri modelli GoF, come lo strategy pattern o il builder pattern, il decorator pattern ha l'obiettivo di gestire le componenti dei software orientati agli oggetti in modo che possano essere riutilizzate in maniera più semplice e flessibile. A questo scopo il modello offre la soluzione per poter aggiungere o **eliminare le dipendenze a un oggetto in maniera dinamica e durante l'esecuzione**, se necessario. Proprio per questo motivo il pattern rappresenta una buona alternativa all'utilizzo delle sottoclassi. Se queste sono infatti in grado di completare una classe in molti modi diversi, non permettono però di apportare modifiche durante l'esecuzione.

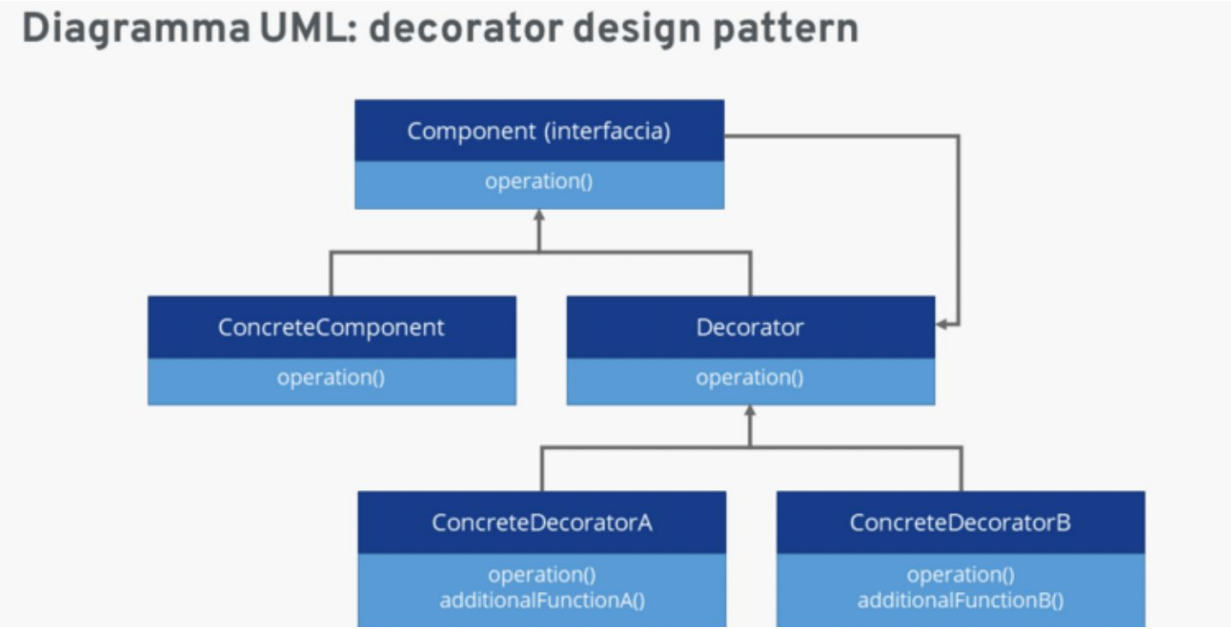
NB! Una componente software può essere **ampliata da un qualsiasi numero di classi decorator**. Tuttavia, queste estensioni rimangono del tutto invisibili per le istanze che vi accedono, tanto che queste non si accorgono che alla classe principale sono state aggiunte delle sottoclassi.

Il decorator e le classi decorator (ConcreteDecorator) **dispongono delle stesse interfacce** delle componenti software che hanno il compito di decorare (ConcreteComponent) e sono dello stesso tipo. Questo è importante per la gestione delle richieste in quanto serve a stabilire se queste debbano essere inoltrate con o senza modifiche, nel caso in cui non sia il decorator a farsi carico della loro elaborazione. Il decorator pattern definisce questa interfaccia elementare come "component", teoricamente corrispondente a una superclasse astratta.

La **relazione tra la componente base e il decorator** può essere meglio illustrata grazie a una rappresentazione grafica sotto forma di diagrammi di classe in UML. Nello schema relativo al decorator design pattern qui sotto abbiamo usato il linguaggio di modellazione grafica per la programmazione orientata agli oggetti.

Funzionamento del decorator pattern: diagramma UML

La rappresentazione esemplificativa del decorator pattern mostra due diverse classi ConcreteDecorator con funzioni aggiuntive. Il numero di queste classi può essere superiore a due.



Vantaggi e svantaggi del decorator pattern in sintesi

VANTAGGI

Considerare l'utilizzo del modello decorator durante l'ideazione di un software risulta ben giustificato per una serie di motivi. Primo di tutto, emerge il **grado di flessibilità** che ottenete adottando una struttura decorator, la quale permette di ampliare le classi con nuovi comportamenti sia durante la fase di compilazione che durante l'esecuzione. Inoltre, con l'approccio decorator **evitate completamente l'ereditarietà e la gerarchiache ne consegue**, il che migliora significativamente la leggibilità del codice di programmazione.

Con la suddivisione della funzionalità su più classi decorator **migliorano anche le prestazioni** del software, ottenendo così di poter richiamare ed eseguire qualsiasi funzione di cui si ha bisogno. Non si ha, invece, questa possibilità di **ottimizzazione delle risorse** con una classe di base complessa che mette a disposizione tutte le funzioni in modo permanente.

Vantaggi e svantaggi del decorator pattern in sintesi

SVANTAGGI

Lo sviluppo basato sul decorator pattern porta con sé anche degli svantaggi. L'inserimento del modello **aumenta automaticamente la complessità del software**. In particolare, l'interfaccia decorator è solitamente molto carica di scritte oltre a usare molte nuove terminologie, il che la rende tutt'altro che adatta ai principianti.

Un ulteriore svantaggio consiste nell'**elevato numero di oggetti decorator**, per i quali è consigliabile una sistematizzazione dedicata, per evitare di avere problemi con la strutturazione, come già accade quando si lavora con le sottoclassi. La catena delle chiamate degli oggetti decorati (le componenti software ampliate) appesantiscono inoltre l'individuazione di eventuali errori e l'intero processo di debugging in generale.

Vantaggi	Svantaggi
Elevato grado di flessibilità	Elevata complessità del software, in particolare dell'interfaccia decorator
Ampliamento della funzionalità delle classi senza ereditarietà	Poco adatto agli utenti alle prime armi
Codice di programmazione di facile lettura	Elevato numero di oggetti
Chiamata delle funzioni ottimizzata per le risorse	Processo di debugging appesantito

Decorator design pattern: gli scenari d’impiego classici

Il decorator pattern offre la base per **oggetti estensibili dinamici e trasparenti** di un software. In modo particolare questo modello trova applicazione nelle componenti delle interfacce grafiche utente, dette anche GUI. Ad esempio: per dotare un campo di testo di un bordo è sufficiente utilizzare un apposito decorator che viene attivato in maniera “invisibile” tra il campo di testo (l’oggetto) e la chiamata, per aggiungere questo nuovo elemento di interfaccia.

Un esempio molto conosciuto di utilizzo del decorator pattern sono le cosiddette **classi stream della libreria Java**, responsabili per la gestione degli input e output di dati. Qui, le classi decorator vengono usate specificamente per aggiungere nuove caratteristiche e informazioni di stato al flusso di dati o mettere a disposizione nuove interfacce.

Java non è l’unico linguaggio di programmazione con il quale viene utilizzato il decorator pattern. Anche i seguenti linguaggi impiegano il modello di design:

- C++
- C#
- Go
- JavaScript
- Python
- PHP

Modelli strutturali: Facade Pattern

Modelli strutturali: Facade pattern

Chi ricerca delle strategie adatte a **semplificare software particolarmente complessi** si imbatte inevitabilmente nel facade design pattern o facade pattern. Assieme ad altri modelli, come il decorator pattern o il composite pattern, fa parte della categoria dei cosiddetti modelli strutturali GoF (abbreviazione di Gang of Four) che dal 1994, data della loro pubblicazione, plasmano il design dei software.

In questo articolo scoprite che cos'è esattamente il facade pattern e come questo aiuta gli sviluppatori ad alleggerire i sottosistemi.

Che cos’è il facade pattern?

Il facade design pattern è uno dei 23 GoF design pattern originariamente pubblicati nel lontano 1994 come guida per lo sviluppo software dai quattro autori Erich Gamma, Ralph Johnson, Richard Helm e John Vlissides nel libro *Design Patterns: Elements of Reusable Object-Oriented Software*. Il principio generale di questi modelli consiste nel semplificare la creazione di software flessibili e riutilizzabili. Nello specifico, il facade pattern offre una soluzione per **raggruppare con facilità le varie interfacce** presenti all’interno dei sistemi più complessi.

Una **classe facade universale,** oltre a fungere da interfaccia, delega alcune importanti funzioni del software ai relativi sottosistemi, così da garantire la migliore gestione possibile delle relazioni tra le varie sottocomponenti di un programma.

Quali problemi risolve l’approccio del facade pattern?

I client che accedono a un sottosistema complesso utilizzano una **moltitudine di oggetti con interfacce anche molto diverse tra loro** o che comunque sono in una relazione di dipendenza con questi oggetti. Dal punto di vista dello sviluppatore, questo rende particolarmente difficoltoso implementare, modificare, testare e riusare i client. È qui che entra in gioco il facade design pattern.

Il modello facade prevede la definizione di un **oggetto facade centrale** (chiamato anche “façade”) che:

- implementa un’**interfaccia universale** per le singole interfacce dei sottosistemi;
- possa eseguire **funzioni aggiuntive**, qualora necessario, prima o dopo l’inoltro di una richiesta da parte del client.

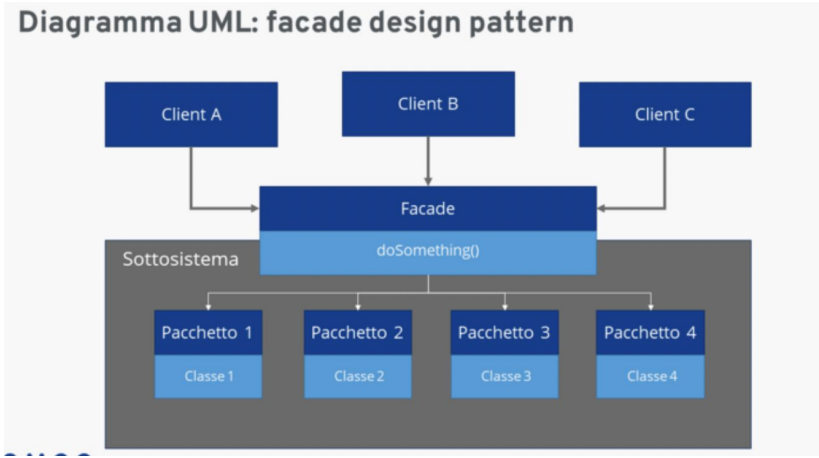
Data la sua funzione di intermediario, l’oggetto facade fa sì che l’accesso e la comunicazione con le singole componenti di un sottosistema sia semplificata e la **dipendenza diretta delle singole componenti minimizzata**. Inoltre, esso delega le richieste al client, in modo che i client non debbano conoscere né le classi né le loro relazioni e dipendenze.

Facade pattern: diagramma di classe in UML

La classe facade è l'unità strutturale decisiva del facade pattern. La sua implementazione ed elaborazione spetta allo sviluppatore che desidera semplificare la complessità del proprio software grazie all'utilizzo di questo pratico modello di design. Una volta applicato il modello, gli oggetti client coinvolti concentrano l'intera comunicazione sulla classe facade che diventa così l'**unica istanza** nel nuovo sistema che viene a crearsi, **dalla quale i client sono direttamente dipendenti**.

Il diagramma UML di seguito riportato serve a **chiarire la relazione** tra i client, l'oggetto facade e le classi del sottosistema durante l'utilizzo del facade pattern.

I quattro pacchetti con le rispettive classi, riportati nel diagramma in UML, serve solo da esempio. Le classi facade permettono di controllare un numero qualsiasi di classi nel sosttosistema.



Facade pattern: vantaggi e svantaggi

VANTAGGI:

I punti di forza del facade design pattern sono chiari: l'oggetto facade "nasconde" i sottosistemi alla base del software, diminuendo così la complessità del sistema. Il suo approccio favorisce inoltre il **principio del loose coupling** o accoppiamento lasco. Il minor grado di dipendenza tra le singole componenti permette di apportare modifiche ed effettuare manutenzioni in qualsiasi momento senza particolari difficoltà, in quanto tali modifiche vengono per lo più eseguite a livello locale. Questo accoppiamento serve inoltre a **rendere il sistema più facile da estendere**.

! N.B.

Se i client necessitano di un accesso diretto a delle specifiche classi del sottosistema, questo può essere reso possibile anche con il modello del facade pattern. In questo caso la visibilità del sottosistema deve essere programmata in modo tale che, all'evenienza, il client possa trascurare il facade.

Facade pattern: vantaggi e svantaggi

SVANTAGGI:

L'impiego del modello di design facade può tuttavia comportare anche degli svantaggi. A causa del suo ruolo centrale, l'implementazione di un facade è un'**operazione molto impegnativa e complicata**. Ancor di più, se lo si desidera inglobare in un codice precedentemente creato. La creazione di un'interfaccia facade corrisponde generalmente a un **livello aggiuntivo di indirezione** e al conseguente allungamento dei tempi per l'elaborazione dell'invocazione dei metodi e delle funzioni, per l'accesso alla memoria, ecc. Infine, il facade pattern cela il rischio che il software diventi troppo dipendente dall'interfaccia superiore principale.

Vantaggi	Svantaggi
Minimizza la complessità dei sottosistemi	Implementazione complicata (in modo particolare all'interno di un codice già esistente)
Promuove il principio dell'accoppiamento lasco	L'approccio è legato a un ulteriore livello di indirezione
Rende il software più flessibile e facilmente riutilizzabile	Maggiore dipendenza dall'interfaccia facade

Facade design pattern: gli scenari d’impiego classici

Le caratteristiche del modello di design facade lo rendono interessante per numerosi scenari di utilizzo. Primo tra tutti, nel caso in cui si desideri **disporre di un’interfaccia uniforme** per l’accesso a sottosistemi complessi o a un numero qualsiasi di oggetti. Un facade garantisce una semplificazione notevole, motivo per il quale l’impiego di una strategia di facade pattern in fase di pianificazione del progetto dovrebbe giocare un ruolo di prim’ordine.

Un altro esempio tipico sono i software in cui si vuole **minimizzare la dipendenza tra client e sottosistemi fondamentali**.

Infine, l’approccio facade risulta valido se vi trovate a pianificare un **progetto software** che deve essere ripartito **su più livelli**. In questo caso, l’utilizzo di facade come interfacce di comunicazione tra i livelli fornisce maggiore flessibilità nel caso di estensioni e modifiche a posteriori delle componenti.

Modelli comportamentali: Observer Pattern

Modelli comportamentali: Observer pattern

Per la programmazione di software bisogna considerare diversi aspetti: il prodotto finale non solo deve avere le funzioni desiderate, ma anche un **codice sorgente il più leggibile e comprensibile** possibile. Il tutto deve avvenire con il minimo sforzo, in particolare quando i programmi o parti di essi sono progettati con funzioni o elementi ricorrenti. Per questo scopo, i cosiddetti pattern o schemi GoF ("**G**ang **o**f **F**our") offrono una serie di **modelli risolutivi predefiniti** per diverse classi di problemi durante la fase di creazione di software.

Oltre ad altri pattern noti, come il Visitor pattern o il Singleton pattern, anche il cosiddetto Observer pattern fa parte di questa raccolta di pratici schemi progettuali che permettono di **semplificare notevolmente la programmazione di routine**. Vi spiegheremo cosa comporta l'Observer design pattern (includendo una rappresentazione grafica in UML) e vi illustreremo i punti forti e deboli di questo schema.

Cos’è l’Observer pattern?

L’Observer design pattern, abbreviato in Observer pattern, è uno tra gli schemi più amati per il design di software informatici. Presenta la possibilità di definire **una dipendenza da uno a molti, quindi tra due o più oggetti**, per **comunicare** le **modifiche** complessive a un oggetto preciso nel modo più semplice e rapido possibile. A tal fine, qualsiasi oggetto, che in questo caso agisce come Observer o osservatore, può registrarsi con un altro oggetto. Quest’ultimo, che in questo caso si definisce soggetto, **informa** gli osservatori registrati non appena ci sono delle modifiche o degli aggiustamenti.

Come già accennato, anche l’Observer pattern fa parte degli **schemi GoF** pubblicati nel 1994 nel “Design Patterns: Elements of Reusable Object-Oriented Software”. Si tratta di più di 20 schemi risolutivi per il design di software che svolgono ancora oggi un ruolo importante per la concezione e l’elaborazione di applicazioni informatiche.

Scopo e funzionamento dell'Observer pattern

L'Observer pattern lavora con due tipi di attori: da una parte troviamo il **subject (soggetto)**, ossia un oggetto il cui stato deve essere osservato a lungo termine. Dall'altra parte troviamo invece gli **oggetti che osservano** (Observer o osservatori), che devono essere informati di tutte le modifiche apportate al subject.

Di solito a un subject sono attribuiti più observer. In linea di massima, però, l'Observer pattern può essere applicato anche in caso di un solo oggetto che osserva.

Scopo e funzionamento dell'Observer pattern

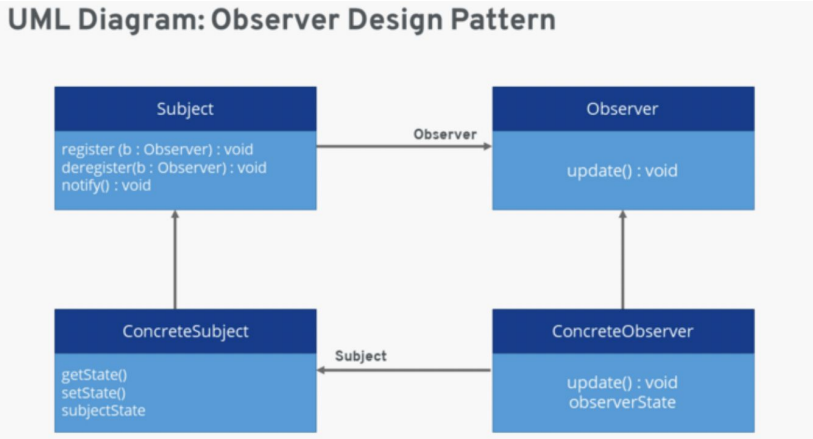
Senza l'ausilio dell'Observer pattern, gli oggetti che osservano dovrebbero richiedere al subject **aggiornamenti sullo stato** a intervalli regolari; ciascuna richiesta implicherebbe i relativi tempi di calcolo e le risorse hardware necessarie. L'idea alla base dell'Observer pattern è di centralizzare il processo di informazione in un soggetto. Questo porta alla creazione di una lista in cui inserire gli osservatori. In caso di una modifica, il soggetto informa gli osservatori registrati nella lista, senza che questi debbano diventare attivi. Se non si desidera più un aggiornamento automatico dello stato di un determinato oggetto che osserva, esso viene semplicemente rimosso dalla lista.

Per informare i singoli osservatori sono disponibili due modelli: nel modello push, il subject comunica lo stato modificato insieme alla notifica. Questo può però comportare problemi quando si trasmettono informazioni che l'observer non è in grado di utilizzare. Questo problema non si verifica nel metodo pull: qui il subject trasmette solo l'informazione di una modifica avvenuta. Gli osservatori possono richiedere infine informazioni dettagliate sullo stato modificato con una richiesta separata.

Rappresentazione grafica dell'Observer pattern (diagramma UML)

Il funzionamento e l'uso di Design Pattern come l'Observer pattern sono spesso incomprensibili a un occhio esterno. Una **rappresentazione grafica dello schema progettuale** può semplificarne la comprensione. In particolare UML (Unified Modeling Language), il linguaggio di modellazione ampiamente diffuso, rende visibili e comprensibili le dipendenze sia per gli utenti tradizionali sia per quelli più esperti. Per questo motivo abbiamo deciso di attingere a UML come linguaggio rappresentativo per la seguente rappresentazione astratta dell'Observer pattern.

Nell'esempio abbiamo rappresentato un solo osservatore; in pratica, però, possono essere molteplici osservatori a interrogare e aggiornare lo stato attuale del soggetto tramite ConcreteObserver e ConcreteSubject.



Vantaggi e svantaggi dell'Observer design pattern

VANTAGGI:

Utilizzare l'Observer pattern per lo sviluppo di software può aiutare in molte situazioni. Il maggior vantaggio offerto da questo modello è l'**elevato grado di indipendenza** tra un oggetto osservato (subject) e gli oggetti che osservano e che si orientano allo stato attuale di quest'oggetto. L'oggetto osservato, ad esempio, non deve disporre di alcuna informazione sui suoi osservatori, perché l'interazione avviene indipendentemente dall'interfaccia dell'observer. Gli **oggetti che osservano ricevono i relativi aggiornamenti in modo automatico**, eliminando del tutto le inutili richieste dal sistema dell'Observer pattern (perché il subject non è cambiato).

Vantaggi e svantaggi dell’Observer design pattern

SVANTAGGI:

Non è però sempre un vantaggio che il soggetto informi in modo automatico tutti gli osservatori registrati riguardo alle relative modifiche: le **informazioni relative alle modifiche**, infatti, vengono comunicate anche se **irrilevanti** per un observer. Questo può avere un impatto particolarmente negativo quando il numero di osservatori registrati è molto alto perché a questo punto lo schema dell’observer potrebbe impiegare molte risorse. Un altro problema dell’Observer pattern è che spesso nel codice sorgente del subject non è chiaro quali osservatori debbano ricevere le informazioni.

Observer pattern: dove si utilizza?

L’Observer design pattern è richiesto soprattutto per quelle applicazioni basate su componenti il cui **stato**

- da una parte **è osservato assiduamente dalle altre componenti,**
- dall’altra è sottoposto a **modifiche regolari**.

Tra i tipici casi d’uso troviamo le GUI (Graphical User Interfaces), che servono agli utenti per comunicare con un software attraverso un’**interfaccia**. Non appena i dati vengono modificati, devono essere aggiornati in tutte le componenti della GUI: uno scenario perfetto per la struttura soggetto-osservatore dell’Observer pattern. Anche i programmi che lavorano con record da visualizzare (tabelle classiche o diagrammi grafici) sfruttano l’ordine offerto dallo schema progettuale.

Per quel che riguarda il **linguaggio di programmazione**, in linea di massima non ci sono limitazioni precise per l’Observer design pattern. È importante soltanto che sia supportato il **paradigma orientato all’oggetto** per dare senso all’implementazione dello schema. Tra i linguaggi che ricorrono spesso a questo pattern troviamo C#, C++, Java, JavaScript, Python e PHP.

Modelli comportamentali: Strategy Pattern

Modelli comportamentali: Strategy pattern

Nella programmazione orientata agli oggetti, i design pattern (in italiano lett. “schemi progettuali”) supportano gli sviluppatori con approcci e modelli risolutivi collaudati. Una volta trovata la soluzione giusta, non resta che apportare ciascuna singola modifica. Al momento, in tutto, esistono **70 schemi progettuali** su misura per aree di applicazione specifiche. Gli strategy design pattern si concentrano sul **comportamento del software**.

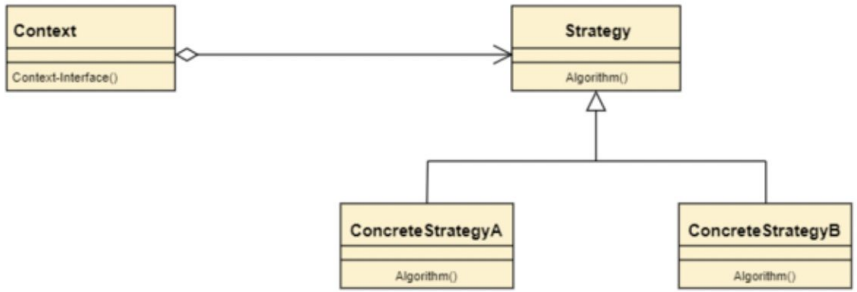
Cos’è uno strategy pattern?

Lo strategy pattern è un tipo di **design pattern comportamentale** (modello di comportamento) che offre al software diversi tipi di soluzione. Dietro le **strategie** c’è una famiglia di algoritmi separati dal programma vero e proprio e autonomi (= scambiabili). Gli schemi progettuali di strategia includono anche specifiche e assistenza per gli sviluppatori. Gli strategy pattern descrivono quindi come creare classi, organizzare un gruppo di classi e creare oggetti. Una peculiarità degli strategy design pattern è la possibilità di implementare un **comportamento del programma e degli oggetti** variabile anche mentre il software è in esecuzione.

Come appare la rappresentazione UML di uno strategy pattern?

Gli strategy pattern sono generalmente progettati con il linguaggio di modellazione grafico UML (Unified Modeling Language). Questo presenta gli schemi progettuali tramite una **notazione standardizzata** e utilizza caratteri e simboli speciali. L'UML fornisce vari tipi di diagramma per la programmazione orientata agli oggetti. Un diagramma di classe con almeno **tre componenti di base** viene solitamente scelto per rappresentare un modello di progettazione della strategia:

- Context (contesto o classe di contesto)
- Strategy (strategia o classe di strategia)
- ConcreteStrategy (strategia concreta)



Struttura di base di uno schema progettuale di strategia in UML con tre componenti di base: Context (classe principale), Strategy (interfaccia) e ConcreteStrategy (algoritmi esternalizzati e dettagli di soluzioni per la risoluzione di problemi specifici)

Strategy pattern

Nello strategy design pattern le componenti di base assumono funzioni speciali: i modelli di comportamento della **classe di contesto** vengono esternalizzati in diverse classi di strategia. Queste classi separate ospitano gli algoritmi noti come ConcreteStrategy. Se necessario, il contesto può accedere alle varianti di calcolo esternalizzate (ConcreteStrategyA, ConcreteStrategyB, ecc.) tramite un **riferimento** (interno). In tal modo non interagisce direttamente con gli algoritmi, ma con un'interfaccia.

L'**interfaccia della strategia incapsula** le varianti di calcolo e può essere implementata da tutti gli algoritmi contemporaneamente. Per l'interazione con il contesto, l'interfaccia generica fornisce un unico metodo per l'attivazione degli algoritmi ConcreteStrategy. Le interazioni con il contesto includono non solo il richiamo di una strategia, ma anche lo **scambio di dati**. L'interfaccia strategica è coinvolta inoltre nei cambiamenti di strategia che possono avvenire anche mentre un programma è in esecuzione.

L'incapsulamento impedisce l'accesso diretto agli algoritmi e alle strutture di dati interne. Un'istanza esterna (client, contesto) può utilizzare solo calcoli e funzioni tramite interfacce definite rendendo così accessibili solo i metodi e i dati di un oggetto rilevanti per l'istanza esterna.

Strategy pattern spiegato con un esempio

Il nostro esempio (basato sul Progetto di studio sullo Strategy Pattern di Philipp Hauer) consiste nel creare un'app di navigazione utilizzando uno schema progettuale di strategia. L'**app deve calcolare un percorso** basato sui normali mezzi di trasporto. L'utente può scegliere fra tre opzioni:

- Pedone (ConcreteStrategyA)
- Auto (ConcreteStrategyB)
- Trasporto locale (ConcreteStrategyC)

Trasferendo queste specifiche su un grafico UML, **la struttura e la funzionalità** dello strategy pattern richiesto diventano chiare:

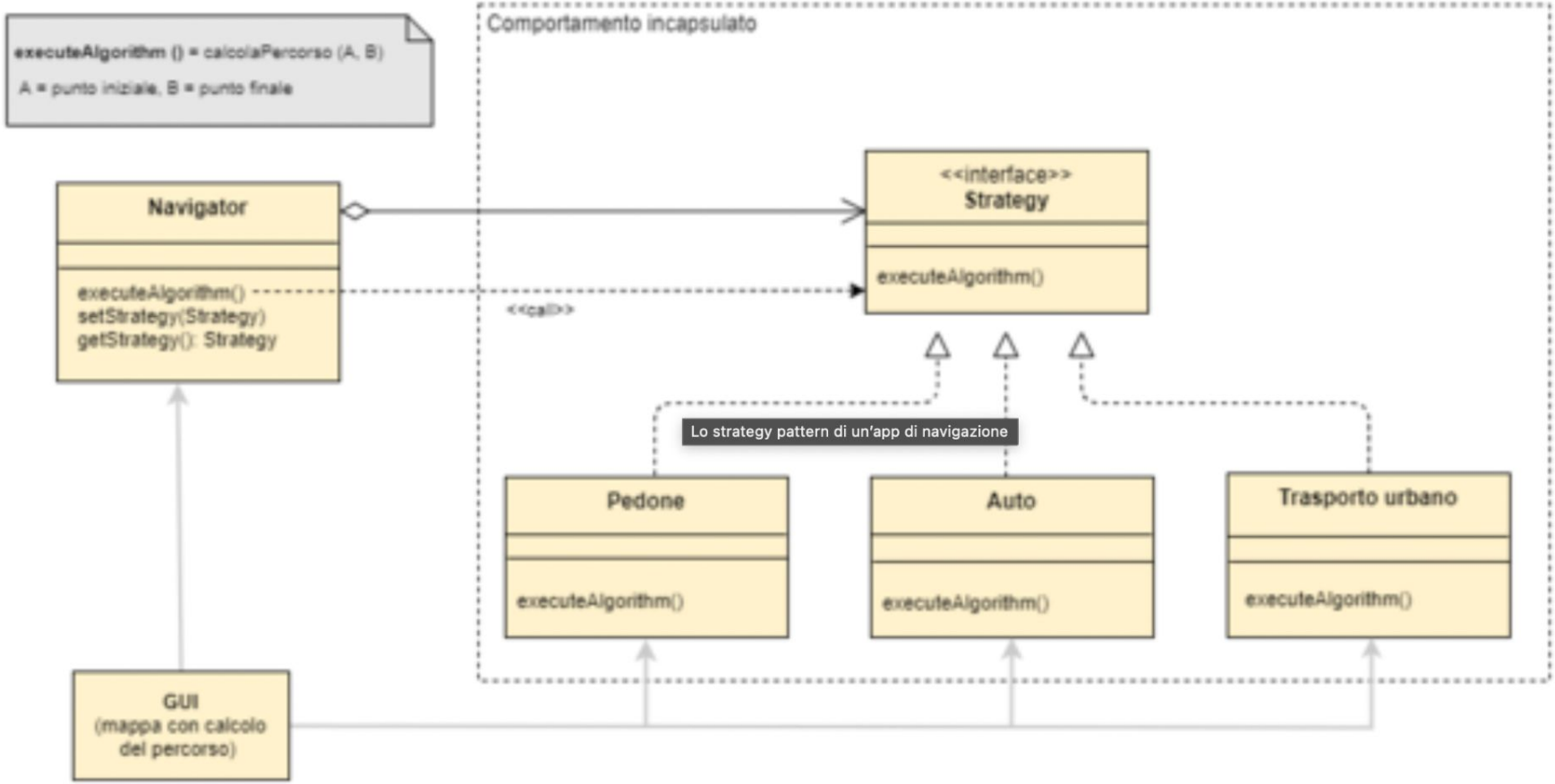
Strategy pattern spiegato con un esempio

Lo schema progettuale della strategia di un'app di navigazione in UML: il contesto riceve un comando dal client dell'app (tasto della GUI) e quindi imposta la strategia richiesta. L'interazione con gli algoritmi di strategia incorporati (ConcreteStrategy) avviene tramite l'interfaccia della strategia. Per poter richiedere una strategia concreta, il client deve conoscere tutte le implementazioni.

Nel nostro esempio, il **client** è l'interfaccia utente grafica (Graphical User Inferface, GUI) di un'app di navigazione con comandi per il calcolo di percorso. Se l'utente effettua una selezione tramite un comando, viene calcolato un percorso specifico. Il **contesto (classe Navigator)** ha il compito di calcolare e visualizzare una serie di punti di controllo sulla mappa. La classe Navigator ha un metodo specifico per cambiare la strategia di calcolo percorso attiva. Ciò significa che è possibile passare facilmente da una modalità di trasporto all'altra utilizzando i comandi del client.

Se, ad esempio, viene attivato un comando corrispondente con il **tasto pedone sul client**, viene richiesto il servizio "Calcola percorso pedonale" (ConcreteStrategyA). Il metodo *executeAlgorithm()* (nel nostro esempio il metodo: *calcolaPercorso (A, B)*) accetta un'origine e una destinazione e produce una raccolta dei punti di controllo della rotta. Il contesto riceve il comando del client e, **in base alle linee guida definite in precedenza (Policy)**, decide la strategia appropriata (*setStrategy: Pedone).* Tramite *Call* delega invece la richiesta all'oggetto-strategia e alla sua interfaccia.

Con *getStrategy()* si memorizza la strategia attualmente selezionata nel contesto (classe del navigatore). I **risultati dei calcoli ConcreteStrategy** confluiscono nell'ulteriore elaborazione e nella visualizzazione grafica del percorso nella app di navigazione. Se l'utente sceglie un percorso diverso, ad esempio facendo clic sul pulsante "Auto" in un secondo momento, il contesto cambia nella strategia richiesta (ConcreteStrategyB) e avvia un nuovo calcolo tramite un'ulteriore *Call*. Alla fine della procedura, viene emessa **una descrizione del percorso modificata per il mezzo di trasporto auto**.


executeAlgorithm () = calcolaPercorso (A, B)
A = punto iniziale, B = punto finale
Comportamento incapsulato
Navigator
executeAlgorithm()
setStrategy(Strategy)
getStrategy(): Strategy
<<interface>>
Strategy
executeAlgorithm()
<<call>>
Lo strategy pattern di un'app di navigazione
Pedone
executeAlgorithm()
Auto
executeAlgorithm()
Trasporto urbano
executeAlgorithm()
GUI
(mappa con calcolo
del percorso)

Modelli comportamentali: Visitor Pattern

Modelli comportamentali: Visitor pattern

La programmazione orientata agli oggetti (OOP), che viene classificata come sottosezione del paradigma di programmazione imperativo ha assunto molta importanza negli ultimi anni. L'opzione di **descrivere tutte le componenti di un progetto software come oggetto**, il cui comportamento può essere definito tramite le relative classi, offre dei vantaggi decisivi rispetto ad altri stili di programmazione. Soprattutto la possibilità di poter riutilizzare componenti di programma senza alcun problema è un argomento decisivo che porta molti sviluppatori a decidere di utilizzare la OOP.

Per rendere ancora più semplice questa **riusabilità**, così come l'implementazione e la possibilità di effettuare dei test degli oggetti incorporati, i modelli di progettazione GoF sono stati presentati nel libro "Design Patterns: Elements of Reusable Object Oriented Software". Tra questi oltre 20 design pattern troviamo anche il cosiddetto Visitor pattern o Visitor design pattern (in italiano: schema progettuale di visitatore), che verrà dettagliatamente descritto nei seguenti paragrafi.

Visitor pattern

Il Visitor design pattern, in breve Visitor pattern, rappresenta un modello risolutivo per separare un **algoritmo dalla struttura di oggetti** a cui è applicato. Descrive un modo per inserire nuove operazioni in strutture di oggetti preesistenti, senza che queste strutture debbano essere modificate per questo. Grazie a questa proprietà, il Visitor pattern rappresenta una possibile opzione per l'**applicazione del principio aperto/chiuso** (OCP). Questo principio dello sviluppo software orientato agli oggetti si basa sul fatto che le unità software, come moduli, classi o metodi, sono contemporaneamente aperte (open) per delle estensioni e chiuse (closed) alle modifiche.

Il Visitor pattern è uno dei 23 modelli di progettazione (categoria: modello comportamentale descritti e pubblicati nel **1994** dagli informatici Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Dato che i quattro sono conosciuti nel mondo degli sviluppatori anche come "Gang of Four", abbreviato in GoF, per il suddetto pattern è stato coniato il nome di **schema progettuale GoF**.

Qual è lo scopo del Visitor design pattern?

Se la struttura a oggetti è composta da molte classi slegate tra loro con una costante richiesta di nuove operazioni, è difficile **per lo sviluppatore dover implementare una nuova sottoclasse per ogni nuova operazione**. Il risultato è un sistema con diverse classi di nodi, difficile non solo da capire ma anche da mantenere e modificare. L'istanza decisiva del Visitor pattern, il **visitatore (visitor)**, permette di aggiungere **nuove funzioni virtuali** a una famiglia di classi, senza dover modificare queste ultime.

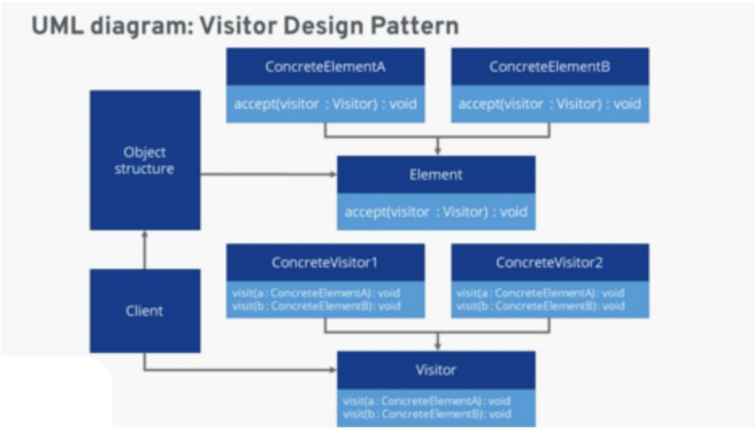
Lo schema progettuale di visitatore prevede che un tale **oggetto visitatore abbia** una definizione separata, con l'obiettivo di implementare un'operazione eseguita su un elemento o più elementi della struttura di oggetti. I **client** che hanno accesso alla struttura di oggetti richiamano il relativo metodo di elementi come "**accept(visitor)**", che delegano la richiesta all'oggetto visitatore accettato. Ne consegue che l'oggetto visitatore può eseguire la relativa operazione.

Le funzioni virtuali o metodi definiscono le funzioni obiettivo eseguibili, per le quali non è necessario che l'obiettivo sia già conosciuto al momento della compilazione. Rappresentano uno strumento importante della lingua orientata agli oggetti.

Rappresentazione grafica del Visitor pattern (UML)

L’interazione tra gli elementi dati e gli oggetti visitatori inclusi secondo il Visitor design pattern può essere spiegata al meglio con una rappresentazione grafica delle relazioni e delle procedure di un possibile software orientato agli oggetti. A tal fine è perfetto il linguaggio di modellazione UML (Unified Modeling Language), che per questo motivo è stato utilizzato anche nel seguente diagramma di classi per il **Visitor pattern**.

Spiegazione del Visitor pattern come soluzione campione per l’estensione flessibile di elementi in software orientati agli oggetti con l’aiuto del linguaggio di modellazione UML (Unified Modeling Language)



I vantaggi e gli svantaggi del Visitor pattern

VANTAGGI:

Il Visitor pattern rappresenta una via già collaudata e ben funzionante per **estendere** delle unità esistenti di un software orientato agli oggetti. In caso si debba aggiungere una nuova operazione, si può procedere tranquillamente tramite la definizione di un nuovo visitatore. Questo modo di procedere consente inoltre di **centralizzare** ogni **codice funzionale**: la relativa implementazione di un'operazione si trova centralmente nella classe visitatore e non deve essere completata in aggiunta nelle altre singole classi. Il vantaggio chiave di un software con il Visitor design pattern insomma, sta nel fatto che **il codice sorgente alla base degli oggetti utilizzati non deve essere costantemente adeguato**. La logica si suddivide invece tra il visitatore, che agisce come sostituto, e le classi di visitatori.

I vantaggi e gli svantaggi del Visitor pattern

SVANTAGGI:

Naturalmente anche lo schema progettuale di visitatore non è perfetto in ogni suo punto. Chi lavora secondo i principi di questo modello deve essere consapevole che anche a seguito di minime **modifiche alla classe di un elemento**, nella maggior parte dei casi sono necessari adeguamenti anche nelle classi di visitatori attribuite. Inoltre, non si risparmia **lavoro aggiuntivo per la successiva introduzione di nuovi elementi**, perché anche per questi vanno implementati metodi visitatore, che a loro volta si devono integrare nelle classi ConcreteVisitor. L'eccellente possibilità di ampliare le unità di software è **legata ad un certo impegno.**

Dove si utilizza il visitor pattern?

Il Visitor design pattern può semplificare notevolmente i **compiti ricorrenti nello sviluppo di software**. Soprattutto per gli sviluppatori che seguono il paradigma di programmazione orientato agli oggetti, sarebbe utile confrontarsi con questo schema progettuale. Dalla sua presentazione nel 1994, il modello si è affermato nell'ambiente di programmazione, e il **tipo di progetto software non svolge un ruolo decisivo in linea di massima** per il fattore d'uso del pattern. Anche relativamente ai **linguaggi di programmazione che ne traggono vantaggio**, non ci sono limitazioni concrete di base per il principio di modellazione, a meno che sia stato tarato in modo particolare per il paradigma orientato agli oggetti.

Il Visitor pattern svolge un ruolo elementare tra gli altri nei seguenti popolari **linguaggi** di programmazione:

- C++
- C#
- Java
- PHP
- Python
- JavaScript
- Golang

Esempi pratici dell’impiego del Visitor pattern

Non è per niente semplice capire i vantaggi e lo scopo del Visitor pattern per un osservatore esterno. Ma chi studia programmazione viene automaticamente a contatto con la modellazione e la sua applicazione.

Per creare un’analogia con la vita reale, per il Visitor pattern si utilizza spesso l’esempio di un **viaggio in taxi**: **un cliente prenota un taxi** che arriva, su richiesta, fino alla porta di casa sua. Una volta che la persona è seduta all’interno del taxi “in visita”, quest’ultimo (o il tassista) è **completamente responsabile del trasporto della persona**.

Spesso per illustrare il funzionamento del Visitor pattern si usa anche l’immagine della **spesa al supermercato**: la persona che deve effettuare gli acquisti mette nel **carrello** ciò che desidera comprare, che metaforicamente rappresenta il **set di elementi** della struttura di oggetti. Una volta alla cassa, il cassiere è il visitatore, che scansiona i prezzi e il peso dei singoli prodotti (o elementi) scelti, per calcolare il **costo finale**.

Conclusioni

I design pattern mettono a disposizione modelli preimpostati con i quali risolvere un problema esplicito, attingendo da un **progetto affidabile**. Il modello si costruisce su un software design già esistente e coinvolge l'utente nella soluzione futura del processo di progettazione. Inoltre, gli schemi progettuali **non sono legati a un linguaggio di programmazione** e sono quindi applicabili a livello universale.

```
8   // Dear programmer:
9   // When I wrote this code, only god and
10  // I knew how it worked.
11  // Now, only god knows it!
12  //
13  // Therefore, if you are trying to optimize
14  // this routine and it fails (most surely),
15  // please increase this counter as a
16  // warning for the next person:
17  //
18  // total_hours_wasted_here = 254
19  //
20
```