Introduzione ai design pattern

Python

Cosa sono i Design Pattern

I Design Pattern sono soluzioni riutilizzabili a problemi ricorrenti nell'ambito della progettazione del software.

Non si tratta di frammenti di codice pronti all'uso, bensì di schemi concettuali che descrivono ruoli e responsabilità tra oggetti, definendo interfacce chiare e collaborazioni efficaci.

Grazie ai pattern, è possibile migliorare la manutenibilità, la flessibilità e la leggibilità del codice, riducendo l'accoppiamento e favorendo l'estendibilità del sistema.

Origini dei Design Pattern

Il concetto di Design Pattern nel software è nato ufficialmente con il libro “Design Patterns: Elements of Reusable Object-Oriented Software” (1994) di Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, noti come “Gang of Four” (GoF).

I quattro autori hanno raccolto e codificato le migliori pratiche emerse nella comunità C++ e Smalltalk, fornendo una tassonomia di 23 pattern suddivisi in tre categorie (Creazionali, Strutturali e Comportamentali).

Categoria	Scopo principale	Fase di applicazione	Esempi tipici
Creazionali	Separare la logica di creazione degli oggetti dal resto del codice, garantendo flessibilità e controllo sul processo di istanziazione.	Quando servono nuovi oggetti, in modo dinamico o condizionato.	Singleton, Factory Method, Abstract Factory, Builder, Prototype
Strutturali	Definire come classi e oggetti si compongono per formare strutture più grandi, ottimizzando la relazione tra componenti.	Quando serve comporre o estendere strutture ad hoc senza cambiare il loro funzionamento interno.	Adapter, Decorator, Facade, Composite, Proxy, Bridge, Flyweight
Comportamentali	Gestire le interazioni e la responsabilità tra oggetti, definendo algoritmi, flussi di controllo e protocolli di comunicazione.	Quando è necessario orchestrare la logica di esecuzione, la comunicazione o il flusso di dati tra oggetti.	Observer, Strategy, Command, Iterator, State, Mediator, Visitor, Template Method, Chain of Responsibility

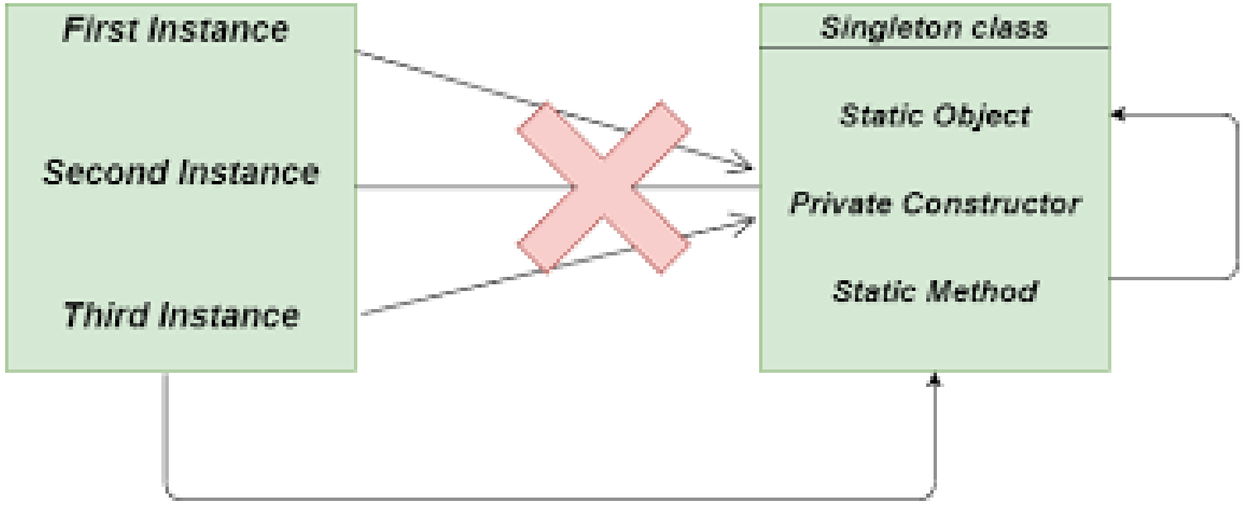
I 5 principali Design Pattern in Python che vedremo sono

- Singleton – Creazionale
- Factory Method – Creazionale
- Decorator – Strutturale
- Adapter – Strutturale
- Observer – Comportamentale
- Strategy – Comportamentale

Singleton (Creazionale)

Il pattern Singleton garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a essa.

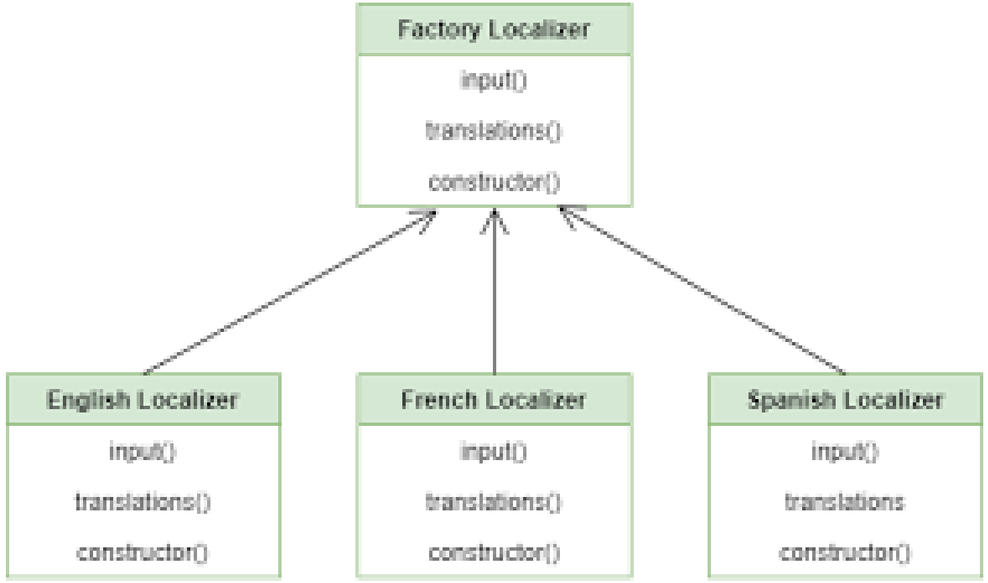
Quando usarlo: quando serve una sola risorsa condivisa, come ad esempio una connessione al database, un logger o un gestore di configurazione.



Factory Method (Creazionale)

Il Factory Method delega la creazione di oggetti a sottoclassi o metodi separati, evitando l'uso diretto del costruttore (__init__).

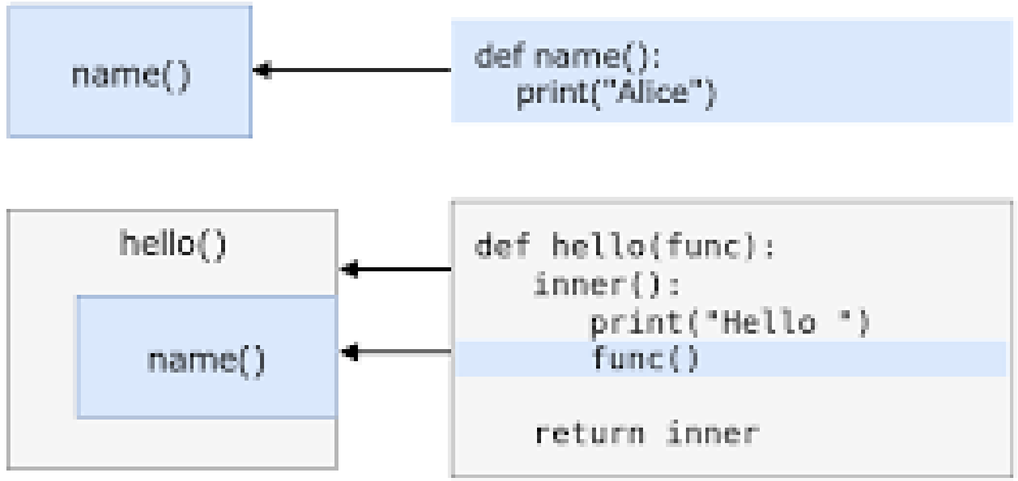
Quando usarlo: quando vuoi disaccoppiare la creazione dell'oggetto dalla sua implementazione, utile quando l'oggetto può variare in base al contesto o a parametri.



Decorator (Strutturale)

Il Decorator consente di aggiungere dinamicamente nuove funzionalità a un oggetto senza modificarne la struttura originale.

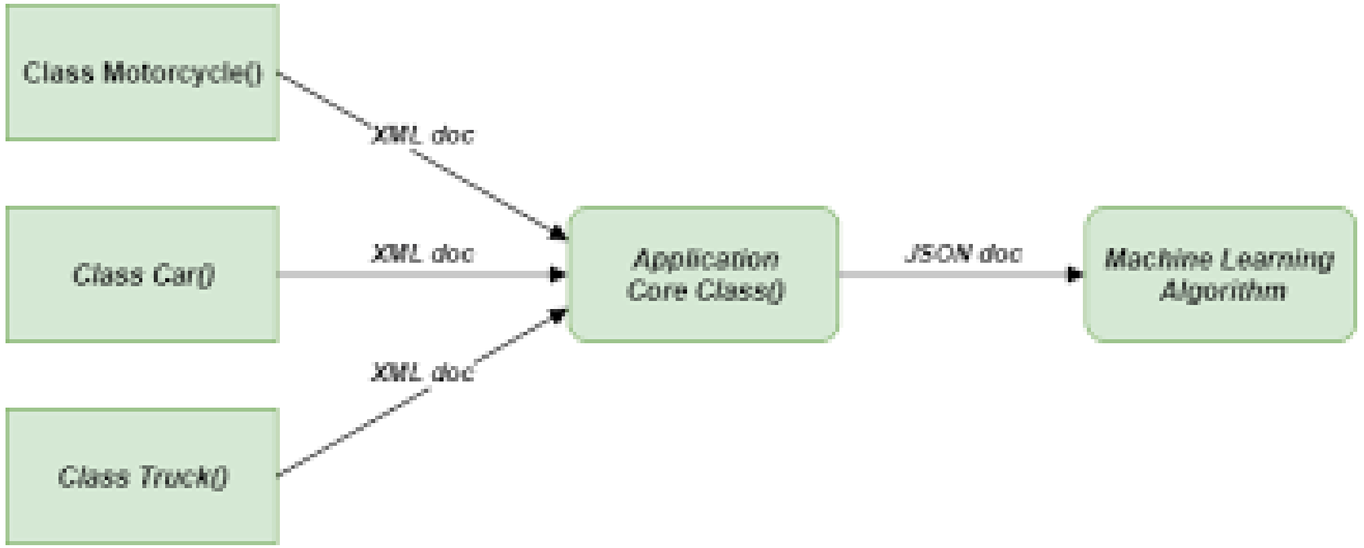
Quando usarlo: per estendere il comportamento di oggetti (es. funzioni o metodi) in modo modulare e flessibile, ad esempio per logging, validazione o timing.



Adapter (Strutturale)

L’Adapter permette a classi con interfacce incompatibili di lavorare insieme, agendo come un ponte tra di esse.

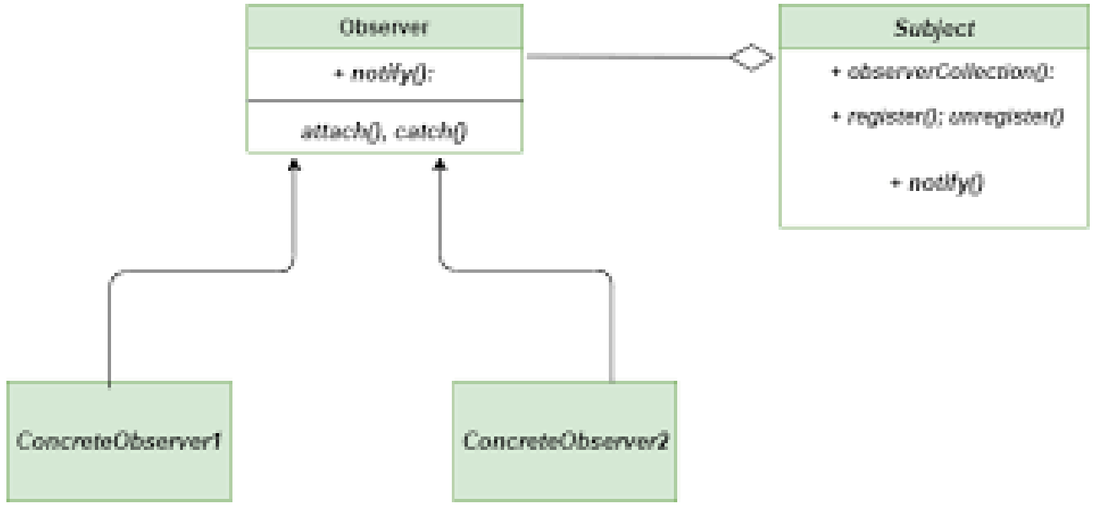
Quando usarlo: quando vuoi integrare codice esistente con nuove interfacce o librerie senza modificarlo, adattandolo a nuovi usi.



Observer (Comportamentale)

Il pattern Observer definisce una relazione di dipendenza uno-a-molti tra oggetti, in cui un cambiamento di stato di un oggetto (soggetto) notifica automaticamente tutti gli osservatori.

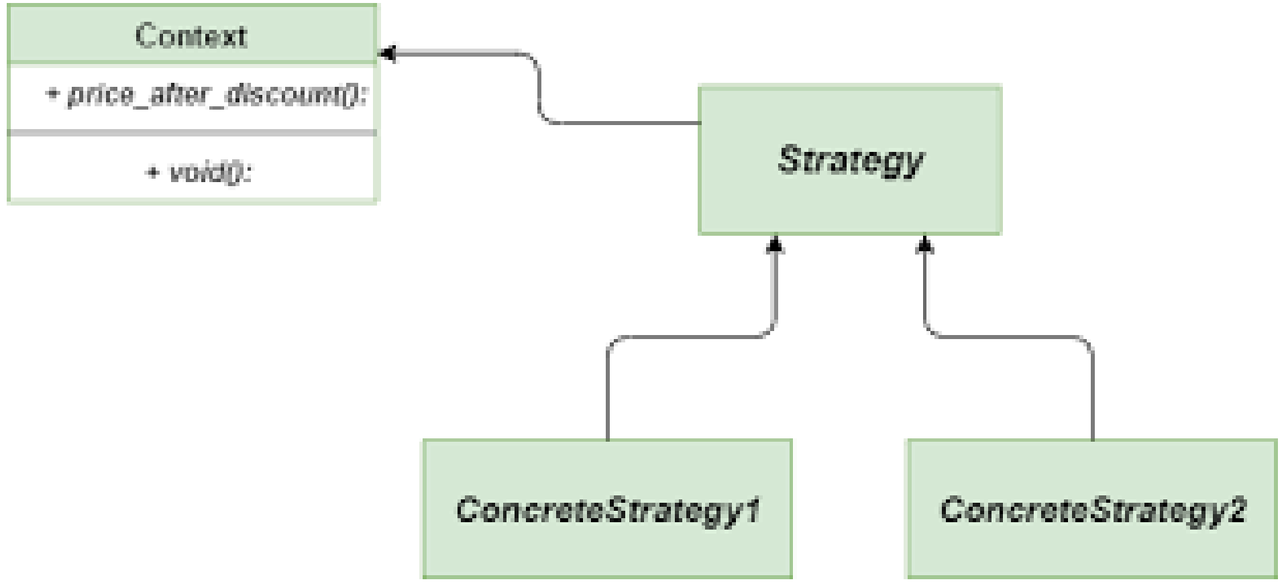
Quando usarlo: quando vari oggetti devono reagire ai cambiamenti di uno stato centrale, come in interfacce grafiche, sistemi di notifica o motori di eventi.



Strategy (Comportamentale)

Lo Strategy consente di definire una famiglia di algoritmi, incapsularli in classi diverse e renderli intercambiabili durante l'esecuzione.

Quando usarlo: quando hai diverse varianti di un algoritmo e vuoi selezionarlo a runtime senza usare condizioni if o switch.



L'utilizzo dei design pattern, pur essendo una pratica utile per migliorare la struttura, la manutenzione e la riusabilità del codice, comporta alcuni rischi, il principale è l'introduzione di complessità non necessaria: applicare un pattern dove non serve può rendere il codice più difficile da leggere, mantenere o estendere.

Un altro rischio è il sovraccarico progettuale, cioè l'over-engineering, che si verifica quando si anticipano esigenze future non confermate.

Inoltre, alcuni pattern possono introdurre un overhead computazionale, incidendo negativamente sulle prestazioni, soprattutto se impiegati in contesti dove la semplicità e la velocità sono prioritarie.

Infine, un pattern mal compreso o implementato in modo scorretto può causare problemi di architettura, rendendo il sistema rigido o inefficiente.

È quindi fondamentale applicarli solo quando risolvono un problema reale e in modo proporzionato alla complessità del progetto.

Buon Davanti a tutti