



Pattern Creazionali

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.



Singleton (Pattern Creazionale)

Il pattern Singleton è un design pattern creazionale il cui obiettivo è assicurarsi che una classe abbia una sola istanza e fornisca un punto di accesso globale a quell'istanza.

È spesso utilizzato per gestire risorse condivise come connessioni al database, file di log, o configurazioni applicative, dove creare più istanze sarebbe inefficiente o porterebbe a conflitti.



Caratteristiche principali

- **Istanza unica garantita**

Assicura che non possano esistere più istanze della stessa classe durante l'esecuzione del programma.

- **Accesso globale controllato**

L'oggetto può essere raggiunto ovunque nel codice, ma la creazione è sotto controllo.

- **Lazy Initialization (opzionale)**

L'istanza viene creata solo al primo accesso, migliorando le performance iniziali.

- **Memoria condivisa**

Tutti i riferimenti puntano alla stessa area di memoria: se modifiche, modifiche per tutti.



```
1. # Esempio di Singleton in Python con metodo __new__
2.
3. class LoggerSingleton:
4.     _istanza = None # Attributo di classe per tenere l'unica istanza
5.
6.     def __new__(cls):
7.         if cls._istanza is None:
8.             print("Creo una nuova istanza del Logger")
9.             cls._istanza = super(LoggerSingleton, cls).__new__(cls)
10.        return cls._istanza
11.
12.    def __init__(self):
13.        self.logs = []
14.
15.    def scrivi_log(self, messaggio):
16.        self.logs.append(messaggio)
17.
18.    def mostra_logs(self):
19.        for log in self.logs:
20.            print(log)
21.
22. # Creazione di due oggetti apparentemente diversi
23. logger1 = LoggerSingleton()
24. logger2 = LoggerSingleton()
25.
26. logger1.scrivi_log("Avvio del programma")
27. logger2.scrivi_log("Errore in riga 42")
28.
29. # Verifica che entrambi siano lo stesso oggetto
30. print(logger1 is logger2) # Output: True
31.
32. # Mostra i log da uno dei due
33. logger1.mostra_logs()
```

Spiegazione codice

- La classe `LoggerSingleton` ridefinisce `__new__` per controllare la creazione dell'oggetto.
- Se `_istanza` è `None`, viene creata una nuova istanza, altrimenti viene restituita quella già creata.
- Il metodo `scrivi_log` aggiunge un messaggio alla lista dei log condivisi.
- Entrambe le variabili `logger1` e `logger2` puntano alla stessa istanza e quindi condividono i dati.



Factory Method

Il Factory Method è un pattern creazionale che definisce un'interfaccia per creare oggetti, ma lascia alle sottoclassi la decisione di quale classe concreta istanziare.

Serve a disaccoppiare la creazione dell'oggetto dal suo utilizzo, seguendo il principio di inversione delle dipendenze.

È particolarmente utile quando si vuole lasciare la libertà di estendere le classi concrete senza modificare il codice client.



Caratteristiche principali

- **Disaccoppiamento tra creazione e utilizzo**

Il codice client lavora con l'interfaccia o classe astratta, non con classi specifiche.

- **Estendibilità facilitata**

Nuove classi concrete possono essere aggiunte senza modificare il codice esistente.

- **Separazione della logica di istanziazione**

Ogni sottoclasse decide come e cosa creare.

- **Polimorfismo**

Si basa sull'uso di ereditarietà e override.



```
1.# Classe prodotto: definisce l'interfaccia comune
2.class Pizza:
3.    def prepara(self):
4.        pass
5.
6.# Classi concrete che estendono Pizza
7.class Margherita(Pizza):
8.    def prepara(self):
9.        print("Preparazione della pizza Margherita...")
10.
11.class Diavola(Pizza):
12.    def prepara(self):
13.        print("Preparazione della pizza Diavola...")
14.
15.# Classe creatore astratto (potrebbe anche essere un'interfaccia)
16.class Pizzeria:
17.    def crea_pizza(self):
18.        pass
19.
20.# Sottoclassi concrete della factory
21.class PizzeriaMargherita(Pizzeria):
22.    def crea_pizza(self):
23.        return Margherita()
24.
25.class PizzeriaDiavola(Pizzeria):
26.    def crea_pizza(self):
27.        return Diavola()
28.
29.# Funzione client che usa il Factory Method
30.def ordina_pizza(pizzeria: Pizzeria):
31.    pizza = pizzeria.crea_pizza()
32.    pizza.prepara()
33.
34.# Uso del Factory Method
35.ordina_pizza(PizzeriaMargherita()) # Output: Preparazione della pizza Margherita...
36.ordina_pizza(PizzeriaDiavola())  # Output: Preparazione della pizza Diavola...
37.
```

Spiegazione codice

- Pizza è la classe prodotto con un metodo astratto `prepara()`.
- Margherita e Diavola sono le implementazioni concrete.
- Pizzeria è la factory astratta che definisce `crea_pizza`.
- Le sottoclassi `PizzeriaMargherita` e `PizzeriaDiavola` definiscono cosa viene istanziato.
- Il client (`ordina_pizza`) lavora solo con l'interfaccia `Pizzeria` e `Pizza`, senza preoccuparsi della loro classe concreta.



Builder Pattern

Il Builder Pattern serve per costruire oggetti complessi passo dopo passo, separando il processo di costruzione dalla rappresentazione finale.

È ideale quando un oggetto può essere costruito in modi diversi o ha molti parametri opzionali che renderebbero poco chiaro l'uso del costruttore tradizionale.



Caratteristiche principali

- **Separazione tra costruzione e rappresentazione**

Il builder gestisce i dettagli di costruzione, mentre il prodotto finale rimane indipendente.

- **Costruzione passo-passo**

L'oggetto può essere costruito con metodi separati che aggiungono/modificano parti.

- **Supporto a oggetti immutabili e configurabili**

Permette di creare oggetti con solo i parametri desiderati, anche in versioni diverse.

- **Più costruttori logici per uno stesso oggetto**

Con diverse implementazioni di builder si possono avere versioni diverse dello stesso oggetto.



```

1. # Prodotto finale
2. class Computer:
3.     def __init__(self):
4.         self.cpu = None
5.         self.ram = None
6.         self.storage = None
7.
8.     def descrizione(self):
9.         print(f"CPU: {self.cpu}, RAM: {self.ram}, Storage: {self.storage}")
10.
11. # Builder astratto
12. class ComputerBuilder:
13.     def reset(self):
14.         pass
15.     def set_cpu(self, cpu):
16.         pass
17.     def set_ram(self, ram):
18.         pass
19.     def set_storage(self, storage):
20.         pass
21.     def get_result(self):
22.         pass
23.
24. # Builder concreto
25. class GamingComputerBuilder(ComputerBuilder):
26.     def __init__(self):
27.         self.reset()
28.
29.     def reset(self):
30.         self.computer = Computer()
31.
32.     def set_cpu(self, cpu="Intel i9"):
33.         self.computer.cpu = cpu
34.
35.     def set_ram(self, ram="32GB"):
36.         self.computer.ram = ram
37.
38.     def set_storage(self, storage="1TB SSD"):
39.         self.computer.storage = storage
40.
41.     def get_result(self):
42.         return self.computer
43.
44. # Director: coordina la costruzione
45. class Director:
46.     def __init__(self, builder: ComputerBuilder):
47.         self.builder = builder
48.
49.     def costruisci_pc_gaming(self):
50.         self.builder.reset()
51.         self.builder.set_cpu()
52.         self.builder.set_ram()
53.         self.builder.set_storage()
54.
55. # Utilizzo del builder pattern
56. builder = GamingComputerBuilder()
57. director = Director(builder)
58.
59. director.costruisci_pc_gaming()
60. pc_gaming = builder.get_result()
61. pc_gaming.descrizione()

```

Spiegazione codice

- La classe **Computer** è il prodotto finale.
- **ComputerBuilder** è l'interfaccia del builder, astratta.
- **GamingComputerBuilder** è un builder concreto che sa costruire un PC da gaming.
- Il **Director** guida la costruzione del prodotto in un certo ordine (CPU → RAM → Storage).
- Il client usa il builder tramite il director e alla fine ottiene l'oggetto con `get_result()`



I Design Pattern Creazionali sono strumenti fondamentali per organizzare e semplificare la creazione di oggetti complessi o variabili nel tempo.

In Python, il loro utilizzo migliora la manutenibilità, la leggibilità e l'estendibilità del codice, specialmente in progetti di medie o grandi dimensioni.

- **Il Singleton garantisce un'unica istanza, ideale per gestire risorse condivise.**
- **Il Factory Method permette di delegare la creazione di oggetti a sottoclassi, mantenendo flessibile il codice.**
- **Il Builder separa costruzione e rappresentazione, utile per oggetti con molte configurazioni.**

Questi pattern promuovono principi solidi di progettazione come inversione delle dipendenze, apertura/chiusura e responsabilità singola, rendendo il software più robusto, modulare e pronto a evolversi.



Buon Davante a tutti

