Pattern Strutturale

Python

I pattern strutturali (structural patterns) fanno parte della classificazione dei Design Pattern descritti nel celebre libro "Design Patterns: Elements of Reusable Object-Oriented Software" (GoF – Gang of Four).

Il loro obiettivo principale è semplificare la composizione delle classi e degli oggetti, fornendo modi flessibili ed efficienti per organizzare le relazioni tra entità software.

Spiegazione Teorica dei Pattern Strutturali

I pattern strutturali si concentrano su come classi e oggetti possono essere combinati per formare strutture più grandi, mantenendo basso l’accoppiamento e alta la flessibilità del sistema.

In pratica, aiutano a costruire architetture modulari, in cui i singoli componenti possono essere facilmente modificati o estesi senza impattare l'intero sistema.

Questi pattern:

- Incapsulano l'uso delle interfacce e l’ereditarietà per adattare o comporre oggetti.
- Possono astrarre la complessità delle dipendenze tra oggetti.
- Sono utili per integrare librerie o componenti esistenti senza modificarli direttamente.

Caratteristiche principali dei Pattern Strutturali

- Adattamento: consentono di far collaborare classi con interfacce diverse (es. Adapter).
- Composizione: promuovono la creazione di oggetti complessi da oggetti più semplici (es. Composite).
- Decoupling: riducono il legame diretto tra le classi (es. Facade, Bridge).
- Flessibilità: facilitano l’estensione delle funzionalità senza alterare il codice esistente.
- Riusabilità: promuovono il riutilizzo di componenti preesistenti integrandoli in nuovi contesti.

NOME	FUNZIONE
Adapter	Converte l’interfaccia di una classe in un’altra attesa dal client.
Bridge	Separa un'astrazione dalla sua implementazione, permettendo di modificarle indipendentemente.
Composite	Compone oggetti in strutture ad albero per rappresentare gerarchie parte-tutto.
Decorator	Aggiunge dinamicamente responsabilità a un oggetto.
Facade	Fornisce un’interfaccia semplificata a un insieme complesso di classi o API.
Flyweight	Minimizza l’uso di memoria condividendo oggetti simili.
Proxy	Fornisce un sostituto o rappresentante di un altro oggetto per controllarne l’accesso.

Adapter Pattern (Strutturale)

Il pattern Adapter (o “Adattatore”) consente di far collaborare classi con interfacce incompatibili, convertendo l’interfaccia di una classe in quella attesa dal client.

In pratica, l’adapter fa da “ponte” tra un'interfaccia esistente e una richiesta diversa, senza modificare il codice originale.

È spesso usato per integrare codice legacy, librerie esterne o sistemi che non possono essere riscritti.

Caratteristiche principali

- **Compatibilità tra classi incompatibili**

Permette a classi esistenti con interfacce diverse di lavorare insieme.

- **Nessuna modifica al codice esistente**

L'adapter lavora esternamente alla classe da adattare.

- **Flessibilità con il codice legacy o di terze parti**

Utile per integrare codice non modificabile.

- **Uso sia con ereditarietà che composizione**

L’adapter può estendere o contenere l’oggetto da adattare.

```python
# Classe con interfaccia già esistente (aspettata dal client)
class LettoreAudio:
    def riproduci(self, nome_file):
        print(f"Riproduzione del file audio: {nome_file}")

# Classe incompatibile (es. una libreria legacy)
class LettoreMp4:
    def play_video(self, file_video):
        print(f"Riproduzione del video MP4: {file_video}")

# Adapter che adatta LettoreMp4 all'interfaccia di LettoreAudio
class Mp4ToAudioAdapter(LettoreAudio):
    def __init__(self, lettore_mp4):
        self.lettore_mp4 = lettore_mp4

    def riproduci(self, nome_file):
        # Conversione dell'interfaccia: da riproduci() a play_video()
        self.lettore_mp4.play_video(nome_file)

# Codice client
def client_riproduci_audio(lettore: LettoreAudio, file):
    lettore.riproduci(file)

# Uso normale con LettoreAudio
lettore_audio = LettoreAudio()
client_riproduci_audio(lettore_audio, "musica.mp3")

# Uso di una classe incompatibile adattata
lettore_legacy = LettoreMp4()
adapter = Mp4ToAudioAdapter(lettore_legacy)
client_riproduci_audio(adapter, "video.mp4")
```

Spiegazione del codice

- **LettoreAudio è la classe che il client si aspetta, con un metodo riproduci.**
- **LettoreMp4 è una classe esistente e incompatibile, che ha un metodo play_video.**
- **Mp4ToAudioAdapter adatta LettoreMp4 per poter essere usato come un LettoreAudio.**
- **Il client (client_riproduci_audio) non sa se sta usando un adattatore o una classe nativa.**

Decorator Pattern (Strutturale)

Il Decorator Pattern consente di aggiungere dinamicamente comportamenti (funzionalità) a un oggetto senza modificarne la classe originale.

È una valida alternativa all'ereditarietà per estendere il comportamento, soprattutto quando vogliamo combinare diverse funzionalità a runtime.

In Python, grazie al supporto per le funzioni come oggetti, questo pattern è molto naturale e può essere implementato sia su oggetti che su funzioni tramite @decorator.

Caratteristiche principali

- **Estensione flessibile senza sottoclassare**

Permette di aggiungere funzionalità senza creare sottoclassi multiple.

- **Combinabilità**

I decorator possono essere concatenati tra loro, permettendo diverse combinazioni di comportamento.

- **Comportamento dinamico**

La decorazione può essere applicata o meno a runtime, secondo necessità.

- **Separazione delle responsabilità**

Ogni decorator aggiunge una specifica responsabilità in modo modulare.

```python
# Classe base (componente)
class Notificatore:
    def invia(self, messaggio):
        print(f"Invio notifica base: {messaggio}")

# Decorator base astratto
class NotificatoreDecorator(Notificatore):
    def __init__(self, notificatore):
        self._notificatore = notificatore

    def invia(self, messaggio):
        self._notificatore.invia(messaggio)

# Decorator concreto: invio via email
class NotificatoreEmail(NotificatoreDecorator):
    def invia(self, messaggio):
        super().invia(messaggio)
        print(f"Inviata anche email con messaggio: {messaggio}")

# Decorator concreto: invio via SMS
class NotificatoreSMS(NotificatoreDecorator):
    def invia(self, messaggio):
        super().invia(messaggio)
        print(f"Inviato anche SMS con messaggio: {messaggio}")

# Uso del decorator
notificatore_base = Notificatore()
notificatore_con_email = NotificatoreEmail(notificatore_base)
notificatore_completo = NotificatoreSMS(notificatore_con_email)

# Invio con catena di decoratori
notificatore_completo.invia("Attenzione! Aggiornamento richiesto.")
```

Spiegazione del codice

- **Notificatore è la classe componente base.**
- **NotificatoreDecorator è la classe base per i decorator, che tiene un riferimento al notificatore da decorare.**
- **NotificatoreEmail e NotificatoreSMS sono decorator concreti che estendono il comportamento.**
- **L’oggetto finale (notificatore_completo) ha un comportamento combinato: invia una notifica base, un’email e un SMS.**

Facade Pattern (Strutturale)

Il Facade Pattern fornisce un’interfaccia semplificata a un sottosistema complesso, rendendo più semplice per il client l’accesso alle funzionalità.

Serve a nascondere la complessità del sistema interno, offrendo un’unica porta d’ingresso “facile” alle operazioni frequenti o comuni.

È utile nei sistemi grandi, modulati in molti componenti, oppure quando si vuole fornire un’interfaccia “pulita” per un gruppo di classi.

Caratteristiche principali

- **Semplifica l’interazione con sistemi complessi**

Nasconde le logiche interne offrendo un'interfaccia amichevole.

- **Riduce il coupling (accoppiamento)**

Il client dipende solo dal Facade, non dalle singole classi interne.

- **Organizzazione modulare**

Aiuta a separare il codice client dalla logica interna del sistema.

- **Interfaccia unica, uso frequente**

Racchiude e organizza funzionalità usate comunemente.

```python
# Sottosistemi complessi
class CPU:
    def avvia(self):
        print("CPU avviata")

class Memoria:
    def carica(self, posizione, dati):
        print(f"Caricamento dei dati '{dati}' alla posizione {posizione}")

class Disco:
    def leggi(self, settore):
        print(f"Lettura dati dal settore {settore}")
        return f"dati_{settore}"

# Facade che semplifica l'uso del sistema
class ComputerFacade:
    def __init__(self):
        self.cpu = CPU()
        self.memoria = Memoria()
        self.disco = Disco()

    def avvia_computer(self):
        print("Avvio del computer in corso...")
        dati = self.disco.leggi(0)
        self.memoria.carica(0, dati)
        self.cpu.avvia()
        print("Computer avviato con successo.")

# Codice client semplificato
computer = ComputerFacade()
computer.avvia_computer()
```

Spiegazione del codice

- Le classi CPU, Memoria e Disco rappresentano componenti interni del sistema.
- ComputerFacade è il facade, che coordina l’interazione tra questi componenti.
- Il client (computer.avvia_computer()) non ha bisogno di conoscere i dettagli di leggi, carica, avvia.


Buon DavantE a tutti