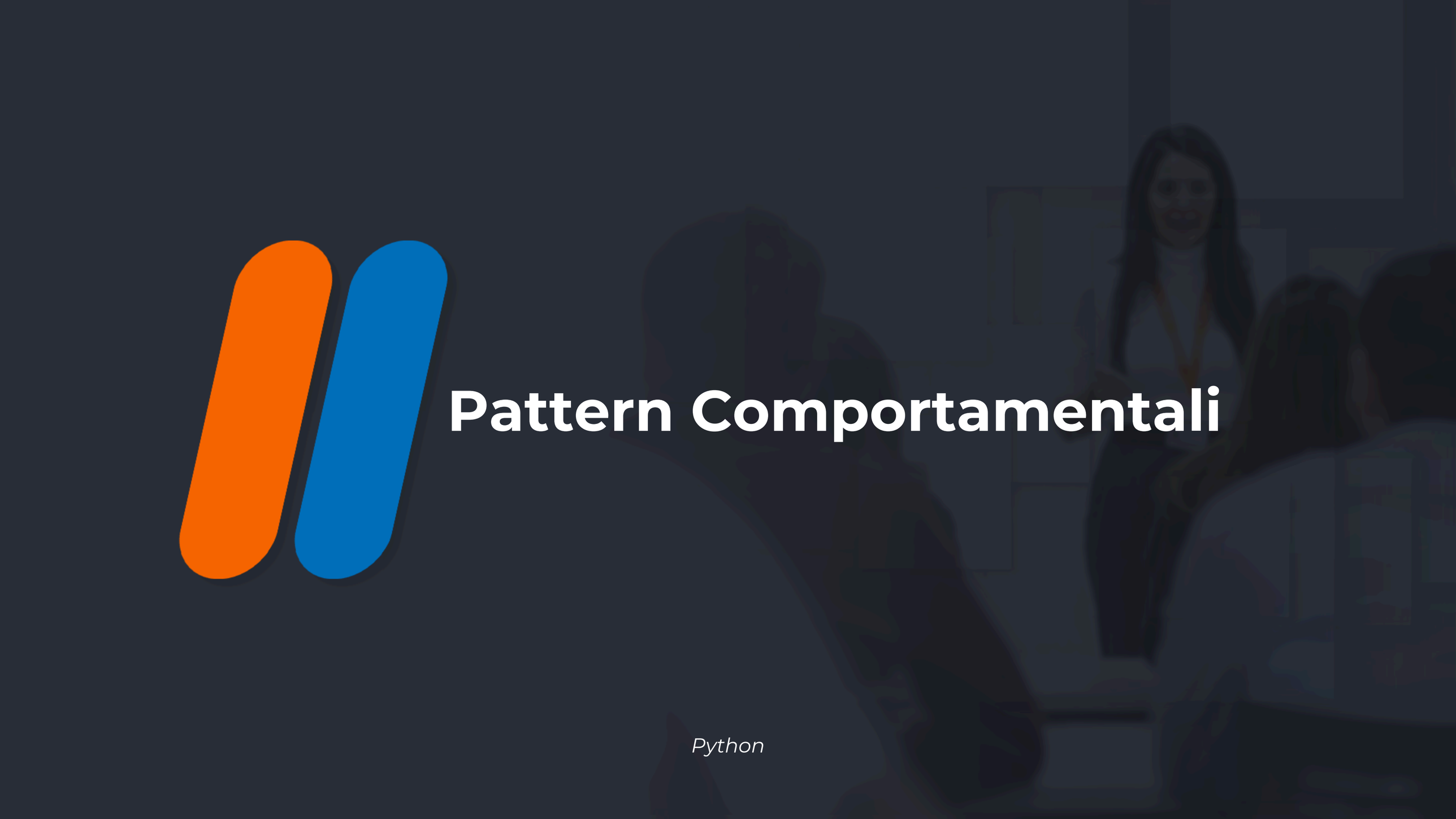


Pattern Comportamentali

Python

I pattern comportamentali si concentrano su come gli oggetti interagiscono e comunicano tra loro, definendo le responsabilità e i flussi di comunicazione tra oggetti.

Mentre i pattern strutturali si occupano della composizione delle classi, i comportamentali sono più focalizzati sulla dinamica del sistema: chi fa cosa, quando e come.

Obiettivi principali dei pattern comportamentali:

- Separare responsabilità complesse tra oggetti diversi.
- Rendere le interazioni flessibili, evitando codice rigido o accoppiato.
- Rendere più facili l’estensione, la modifica o la sostituzione di comportamenti.
- Fornire alternative eleganti all’uso massiccio di condizionali o cicli.

Caratteristiche comuni dei pattern comportamentali

Caratteristica	Spiegazione breve
Comunicazione decentrata	Gli oggetti comunicano senza conoscersi direttamente.
Separazione delle responsabilità	Le decisioni vengono delegate a oggetti o funzioni dedicate.
Inversione del controllo	Gli oggetti si affidano ad altri per gestire comportamenti specifici.
Estensibilità	È facile aggiungere nuovi comportamenti senza modificare codice esistente.

Pattern Comportamentali più comuni

Pattern	Descrizione breve
Observer	Un oggetto notifica automaticamente tutti gli altri interessati quando cambia stato.
Strategy	Incapsula algoritmi diversi e li rende intercambiabili.
Command	Incapsula una richiesta come oggetto, permettendo undo/redo e queue di comandi.
State	Permette a un oggetto di cambiare comportamento al variare del suo stato interno.
Template Method	Definisce il skeleton di un algoritmo, delegando i dettagli alle sottoclassi.
Chain of Responsibility	Passa una richiesta attraverso una catena di gestori fino a quando uno la gestisce.

Observer Pattern (Comportamentale)

Il pattern Observer definisce una relazione uno-a-molti tra oggetti, in cui quando un oggetto cambia stato, tutti gli altri oggetti interessati (osservatori) vengono notificati automaticamente.

È molto utile quando bisogna mantenere sincronizzati più oggetti senza creare un forte accoppiamento tra loro.

Caratteristiche principali

- **Relazione uno-a-molti**

Un “soggetto” può notificare più osservatori contemporaneamente.

- **Disaccoppiamento**

Il soggetto non conosce i dettagli degli osservatori, solo che devono essere notificati.

- **Aggiornamenti automatici**

Gli osservatori ricevono le modifiche in tempo reale.

- **Estendibilità**

È facile aggiungere nuovi osservatori senza toccare il codice del soggetto.

```python
# Classe Soggetto (Publisher)
class Subject:
    def __init__(self):
        self._observers = []  # Lista degli osservatori

    def aggiungi_observer(self, observer):
        self._observers.append(observer)

    def rimuovi_observer(self, observer):
        self._observers.remove(observer)

    def notifica_observers(self, messaggio):
        for observer in self._observers:
            observer.aggiorna(messaggio)

# Classe Osservatore (Subscriber)
class Observer:
    def aggiorna(self, messaggio):
        pass  # Interfaccia generica

# Osservatore concreti
class UtenteApp(Observer):
    def __init__(self, nome):
        self.nome = nome

    def aggiorna(self, messaggio):
        print(f"[{self.nome}] Notifica ricevuta: {messaggio}")

class Logger(Observer):
    def aggiorna(self, messaggio):
        print(f"[LOGGER] Registrazione evento: {messaggio}")

# Uso del pattern Observer
if __name__ == "__main__":
    # Creo il soggetto
    notiziario = Subject()

    # Creo osservatori
    utente1 = UtenteApp("Alice")
    utente2 = UtenteApp("Bob")
    logger = Logger()

    # Registro gli osservatori
    notiziario.aggiungi_observer(utente1)
    notiziario.aggiungi_observer(utente2)
    notiziario.aggiungi_observer(logger)

    # Invio una notifica a tutti
    notiziario.notifica_observers("Nuova versione dell'app disponibile!")
```

Spiegazione del codice

- **Subject è il soggetto osservato: tiene traccia degli osservatori e li notifica quando c'è un evento.**
- **Observer è un'interfaccia base per gli osservatori.**
- **UtenteApp e Logger sono osservatori concreti che reagiscono diversamente alla notifica.**
- **notiziario è l'oggetto centrale: quando chiamiamo notifica_observers, tutti gli iscritti vengono aggiornati.**

Strategy Pattern (Comportamentale)

Lo Strategy Pattern permette di selezionare dinamicamente un algoritmo tra più disponibili, incapsulandoli in classi separate, tutte intercambiabili attraverso un'interfaccia comune.

È utile quando un oggetto ha bisogno di eseguire diversi comportamenti che possono variare in base al contesto, evitando quindi catene di if/else o switch.

Caratteristiche principali

- **Comportamento intercambiabile a runtime**

Si può cambiare l’algoritmo anche mentre il programma è in esecuzione.

- **Disaccoppiamento della logica dal contesto**

L’oggetto principale non conosce i dettagli degli algoritmi concreti.

- **Aggiunta facile di nuove strategie**

Basta creare una nuova classe che implementa l’interfaccia.

- **Codice pulito e aperto a estensioni**

Segue il principio Open/Closed: puoi estendere senza modificare.

```python
# Interfaccia Strategy
class StrategiaPagamento:
    def paga(self, importo):
        pass

# Strategie concrete
class PagamentoConCarta(StrategiaPagamento):
    def paga(self, importo):
        print(f"Pagamento di €{importo} effettuato con Carta di Credito.")

class PagamentoConPayPal(StrategiaPagamento):
    def paga(self, importo):
        print(f"Pagamento di €{importo} effettuato con PayPal.")

class PagamentoConBitcoin(StrategiaPagamento):
    def paga(self, importo):
        print(f"Pagamento di €{importo} effettuato con Bitcoin.")

# Classe Contesto
class Carrello:
    def __init__(self):
        self._strategia = None

    def set_strategia_pagamento(self, strategia: StrategiaPagamento):
        self._strategia = strategia

    def checkout(self, importo):
        if not self._strategia:
            print("Errore: Nessuna strategia di pagamento selezionata.")
            return
        self._strategia.paga(importo)

# Esempio d'uso
carrello = Carrello()

# Imposto la strategia a PayPal
carrello.set_strategia_pagamento(PagamentoConPayPal())
carrello.checkout(49.99)

# Cambio la strategia a Bitcoin
carrello.set_strategia_pagamento(PagamentoConBitcoin())
carrello.checkout(149.99)

```

Spiegazione del codice

- **StrategiaPagamento è l'interfaccia comune per tutte le strategie di pagamento.**
- **PagamentoConCarta, PagamentoConPayPal e PagamentoConBitcoin sono strategie concrete.**
- **Carrello è il contesto che usa una strategia ma non sa quale: la riceve dall'esterno.**
- **Il comportamento (checkout) cambia dinamicamente a seconda della strategia selezionata.**

Command Pattern (Comportamentale)

Il Command Pattern incapsula una richiesta come oggetto, permettendo di parametrizzare gli invocatori con operazioni diverse, mettere in coda, disaccoppiare o annullare comandi in modo flessibile.

È usato per scenari in cui si vuole disaccoppiare chi invoca un'azione da chi la esegue, come nei sistemi undo/redo, nei controller GUI, nei bot, o nelle macro.

Caratteristiche principali

- **Disaccoppiamento tra invocazione e esecuzione**

Il client non sa chi esegue l’azione, sa solo che può “invocarla”.

- **Comandi come oggetti**

Le azioni diventano oggetti riutilizzabili, modificabili e storicizzabili.

- **Supporto per undo/redo, logging, macro**

Ogni comando può essere archiviato, annullato o rieseguito.

- **Parametrizzazione flessibile**

I comandi possono essere creati e passati dinamicamente.

```python
# Receiver: oggetto che sa eseguire le azioni
class Lampada:
    def accendi(self):
        print("La lampada è accesa.")

    def spegni(self):
        print("La lampada è spenta.")

# Interfaccia Command
class Comando:
    def esegui(self):
        pass

# Comandi concreti
class ComandoAccendi(Comando):
    def __init__(self, lampada):
        self.lampada = lampada

    def esegui(self):
        self.lampada.accendi()

class ComandoSpegni(Comando):
    def __init__(self, lampada):
        self.lampada = lampada

    def esegui(self):
        self.lampada.spegni()

# Invoker: l'oggetto che invoca il comando (es. un bottone)
class Pulsante:
    def __init__(self, comando):
        self.comando = comando

    def premi(self):
        self.comando.esegui()

# Uso del pattern Command
lampada = Lampada()

# Creo i comandi
comando_accendi = ComandoAccendi(lampada)
comando_spegni = ComandoSpegni(lampada)

# Associo i comandi ai pulsanti
pulsante_on = Pulsante(comando_accendi)
pulsante_off = Pulsante(comando_spegni)

# Simulazione dell'uso
pulsante_on.premi()   # Output: La lampada è accesa.
pulsante_off.premi()  # Output: La lampada è spenta.
```

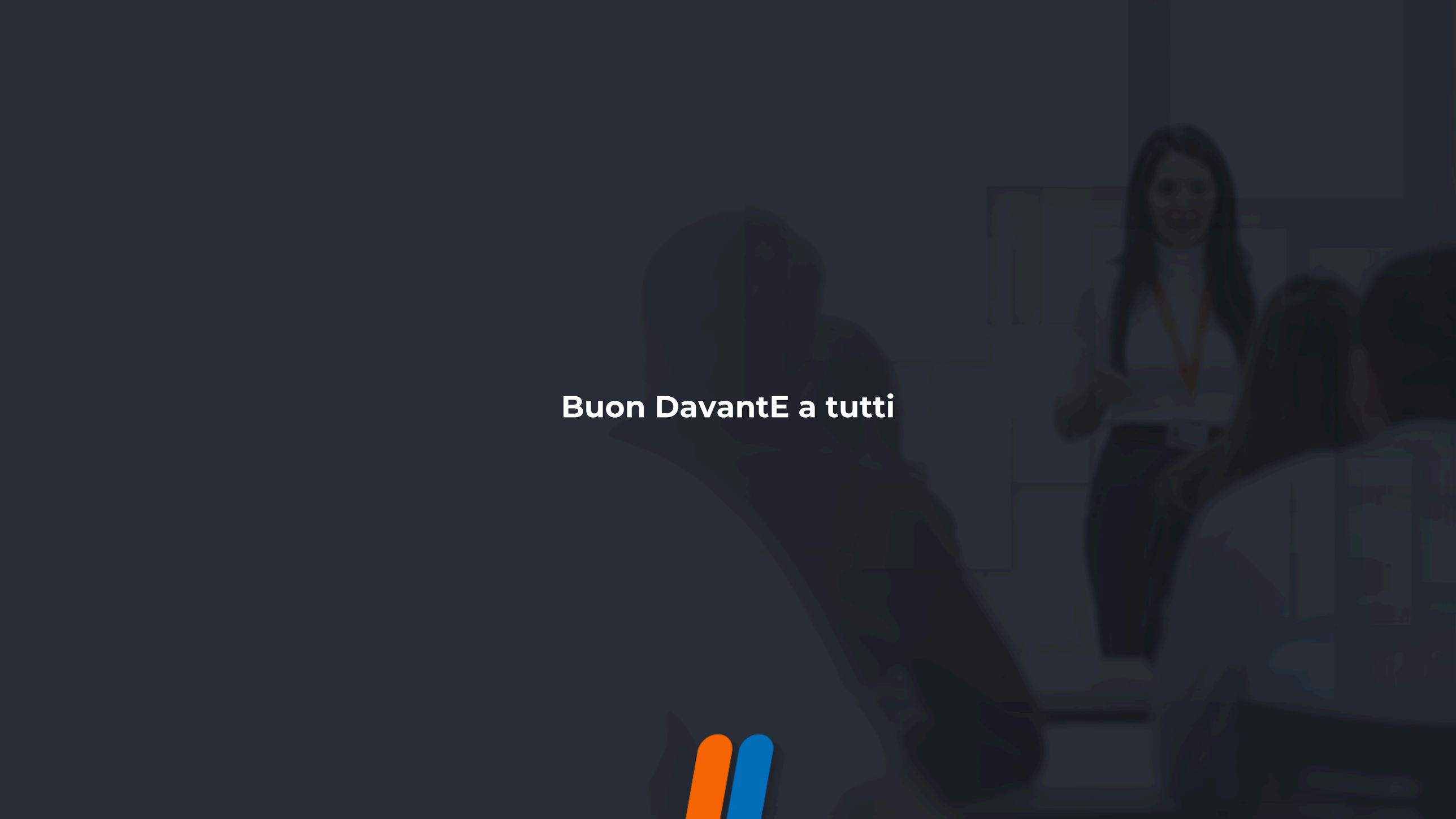
Spiegazione del codice

- Lampada è il receiver: l'oggetto che fa il lavoro vero.
- Comando è l’interfaccia base per tutti i comandi.
- ComandoAccendi e ComandoSpegni sono comandi concreti che invocano metodi sulla lampada.
- Pulsante è l’invoker, che richiama il comando senza conoscere i dettagli dell’azione.

Conclusione sui Pattern Comportamentali

I pattern comportamentali si concentrano su come gli oggetti interagiscono e collaborano tra loro per svolgere compiti complessi, mantenendo bassa coesione e alta flessibilità.

Sono fondamentali per organizzare la logica dinamica e le relazioni tra componenti in modo estendibile e modulare.


Buon DavantE a tutti