



Pattern Aziendale - Decorator Pattern



Descrizione

Il Decorator Pattern è la scelta migliore per questo scenario aziendale perché permette di aggiungere dinamicamente nuove responsabilità agli oggetti. Nel nostro caso, un ingegnere può assumere ruoli aggiuntivi (Project Manager e/o Administrative Manager) mantenendo le sue responsabilità base.

Perché il Decorator Pattern?

- Permette di aggiungere responsabilità in modo dinamico
- Mantiene il principio Single Responsibility
- Evita la proliferazione di sottoclassi
- Supporta la composizione di ruoli in modo flessibile



Vantaggi e Svantaggi

Vantaggi

- ● Maggiore flessibilità rispetto all'ereditarietà
- ● Aggiunta/rimozione di responsabilità a runtime
- ● Rispetta il principio Open/Closed
- ● Evita classi sovraccariche di funzionalità

Svantaggi

- ● Può risultare in molti oggetti piccoli e simili
- ● L'ordine di decorazione può essere importante
- ● Può essere più complesso da debuggare



Implementazione

```
# Esempio semplificato del pattern
class Employee(ABC):
    @abstractmethod
    def get_daily_tasks(self) -> List[str]:
        pass
```

```
class Engineer(Employee):
    def get_daily_tasks(self) -> List[str]:
        return ["Sviluppo task"]

class ProjectManagerDecorator(EmployeeDecorator):
    def get_daily_tasks(self) -> List[str]:
        return super().get_daily_tasks() + ["Gestione progetto"]
```

Output dell'implementazione

Ruolo completo: Ingegnere, Project Manager

Task giornalieri:

- Sviluppo task
- Gestione progetto

Il Decorator Pattern è stato scelto perché:

1. Permette di aggiungere responsabilità in modo dinamico e flessibile
2. Mantiene la separazione delle responsabilità
3. Evita la necessità di creare sottoclassi per ogni combinazione di ruoli
4. Supporta la composizione di ruoli in modo trasparente e modulare