🎭 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

```
from abc import ABC, abstractmethod
from typing import List

class Employee(ABC):
    @abstractmethod
    def get_daily_tasks(self) -> List[str]:
```

```
        pass

    @abstractmethod
    def get_role(self) -> str:
        pass

class Engineer(Employee):
    def __init__(self, name: str):
        self.name = name
        self._tasks = []

    def add_task(self, task: str):
        self._tasks.append(task)

    def get_daily_tasks(self) -> List[str]:
        return [f"Sviluppo: {task}" for task in self._tasks]

    def get_role(self) -> str:
        return "Ingegnere"

class EmployeeDecorator(Employee):
    def __init__(self, employee: Employee):
        self._employee = employee

    def get_daily_tasks(self) -> List[str]:
        return self._employee.get_daily_tasks()

    def get_role(self) -> str:
        return self._employee.get_role()

class ProjectManagerDecorator(EmployeeDecorator):
    def __init__(self, employee: Employee):
        super().__init__(employee)
        self.project_members = []

    def add_project_member(self, employee: Employee):
        self.project_members.append(employee)

    def get_daily_tasks(self) -> List[str]:
        tasks = super().get_daily_tasks()
        tasks.extend([f"Supervisione progetto per: {emp.get_role()}" for emp
in self.project_members])
        return tasks

    def get_role(self) -> str:
        return f"{self._employee.get_role()}, Project Manager"
```

```python
class AdministrativeManagerDecorator(EmployeeDecorator):
    def __init__(self, employee: Employee):
        super().__init__(employee)
        self.managed_employees = []

    def add_employee(self, employee: Employee):
        self.managed_employees.append(employee)

    def get_daily_tasks(self) -> List[str]:
        tasks = super().get_daily_tasks()
        tasks.extend([f"Gestione amministrativa: {emp.get_role()}" for emp in
self.managed_employees])
        return tasks

    def get_role(self) -> str:
        return f"{self._employee.get_role()}, Administrative Manager"

# Output di esempio
def main():
    # Creazione ingegneri base
    eng1 = Engineer("Mario Rossi")
    eng2 = Engineer("Luigi Verdi")
    eng3 = Engineer("Paolo Bianchi")

    # Aggiunta di task base
    eng1.add_task("Implementare nuova feature")
    eng2.add_task("Fix bug #123")

    # Decorazione con ruoli aggiuntivi
    project_manager = ProjectManagerDecorator(eng1)
    project_manager.add_project_member(eng2)
    project_manager.add_project_member(eng3)

    # Aggiunta del ruolo amministrativo allo stesso ingegnere
    admin_project_manager = AdministrativeManagerDecorator(project_manager)
    admin_project_manager.add_employee(eng2)
    admin_project_manager.add_employee(eng3)

    # Stampa dei risultati
    print(f"\nRuolo completo: {admin_project_manager.get_role()}")
    print("\nTask giornalieri:")
    for task in admin_project_manager.get_daily_tasks():
        print(f"- {task}")
```

```
if __name__ == "__main__":
    main()
```

🎯 Output dell'implementazione

```
Ruolo completo: Ingegnere, Project Manager, Administrative Manager

Task giornalieri:
- Sviluppo: Implementare nuova feature
- Supervisione progetto per: Ingegnere
- Supervisione progetto per: Ingegnere
- Gestione amministrativa: Ingegnere
- Gestione amministrativa: Ingegnere
```

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