🏭 Pattern Aziendale - Composite Pattern

📝 Descrizione

Per gestire la struttura aziendale descritta, il pattern più appropriato è il Composite Pattern. Questo pattern permette di creare una gerarchia di oggetti dove sia gli oggetti singoli (Employee) che le composizioni di oggetti (Manager) possono essere trattati in modo uniforme.

Perché il Composite Pattern?

- Permette di rappresentare gerarchie parte-tutto
- Consente di trattare oggetti singoli e composizioni in modo uniforme
- Facilita l'aggiunta di nuovi tipi di dipendenti
- Gestisce facilmente la doppia responsabilità (es. Engineer che è sia AdministrativeManager che ProjectManager)

⚖️ Vantaggi e Svantaggi

Vantaggi

- 🟢 Struttura flessibile e facilmente estendibile
- 🟢 Semplifica la gestione delle gerarchie
- 🟢 Codice più pulito e mantenibile

Svantaggi

- 🔴 Può diventare troppo generico
- 🔴 Potenziale complessità nella gestione delle responsabilità multiple

💻 Implementazione

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

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

    @abstractmethod
```

```
    def get_daily_tasks(self) -> List[str]:
        pass

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

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

class AdministrativeManager(Engineer):
    def __init__(self, name: str):
        super().__init__(name)
        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.name}" for emp in
self.managed_employees])
        return tasks

class ProjectManager(Engineer):
    def __init__(self, name: str):
        super().__init__(name)
        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.name}" for emp in
self.project_members])
        return tasks

# Output di esempio
def main():
    # Creazione dipendenti
    eng1 = Engineer("Mario Rossi")
    eng2 = Engineer("Luigi Verdi")

    # Creazione manager
    admin_manager = AdministrativeManager("Giovanni Bianchi")
```

```
    project_manager = ProjectManager("Anna Neri")

    # Assegnazione task
    eng1.add_task("Implementare nuova feature")
    eng2.add_task("Fix bug #123")

    # Gestione gerarchie
    admin_manager.add_employee(eng1)
    admin_manager.add_employee(eng2)

    project_manager.add_project_member(eng1)
    project_manager.add_task("Revisione codice")

    # Stampa dei task giornalieri
    print(f"\nTask di {eng1.name}:")
    print("\n".join(eng1.get_daily_tasks()))

    print(f"\nTask di {admin_manager.name}:")
    print("\n".join(admin_manager.get_daily_tasks()))

    print(f"\nTask di {project_manager.name}:")
    print("\n".join(project_manager.get_daily_tasks()))

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

🎯 Output dell'implementazione

```
Task di Mario Rossi:
Sviluppo: Implementare nuova feature

Task di Giovanni Bianchi:
Gestione amministrativa: Mario Rossi
Gestione amministrativa: Luigi Verdi

Task di Anna Neri:
Sviluppo: Revisione codice
Supervisione progetto per: Mario Rossi
```

Il Composite Pattern è stato scelto perché:

1. Permette di gestire la gerarchia aziendale in modo naturale
2. Consente di trattare sia dipendenti singoli che manager in modo uniforme

3. Supporta la composizione di ruoli (un Engineer può essere sia AdministrativeManager che ProjectManager)
4. Facilita l'aggiunta di nuovi tipi di dipendenti senza modificare il codice esistente