Dato che il file task.txt è vuoto, creerò un esempio generico di un preventivatore utilizzando il pattern Strategy, che è ideale per gestire diversi algoritmi di calcolo dei preventivi.

📋 Preventivatore v2 - Documentation

🎯 Pattern Scelto: Strategy

Il pattern Strategy è stato scelto per le seguenti ragioni:

- Permette di definire una famiglia di algoritmi di calcolo preventivi
- Rende gli algoritmi intercambiabili
- Isola la logica di calcolo dal resto dell'applicazione
- Facilita l'aggiunta di nuove strategie di calcolo

✅ Vantaggi

- Flessibilità nel cambiare l'algoritmo di calcolo a runtime
- Separazione delle responsabilità
- Facile aggiunta di nuove strategie
- Eliminazione di condizioni multiple switch/if

❌ Svantaggi

- Aumenta il numero di oggetti nel sistema
- Il client deve conoscere le differenze tra le strategie
- Overhead di comunicazione tra Strategy e Context

💻 Implementazione

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

# Strategy Interface
class StrategiaPreventivo(ABC):
    @abstractmethod
    def calcola_preventivo(self, dati: Dict) -> float:
        pass

# Concrete Strategies
class PreventivoBase(StrategiaPreventivo):
```

```python
    def calcola_preventivo(self, dati: Dict) -> float:
        base = dati.get('importo_base', 0)
        return base * 1.22  # IVA 22%

class PreventivoSconto(StrategiaPreventivo):
    def calcola_preventivo(self, dati: Dict) -> float:
        base = dati.get('importo_base', 0)
        sconto = dati.get('sconto', 0)
        return (base * (1 - sconto)) * 1.22

class PreventivoPremium(StrategiaPreventivo):
    def calcola_preventivo(self, dati: Dict) -> float:
        base = dati.get('importo_base', 0)
        extra = dati.get('servizi_extra', 0)
        return (base + extra) * 1.22

# Context
class Preventivatore:
    def __init__(self, strategia: StrategiaPreventivo):
        self._strategia = strategia

    def set_strategia(self, strategia: StrategiaPreventivo):
        self._strategia = strategia

    def genera_preventivo(self, dati: Dict) -> float:
        return self._strategia.calcola_preventivo(dati)

# Output di esempio
if __name__ == "__main__":
    dati_base = {"importo_base": 1000}
    dati_sconto = {"importo_base": 1000, "sconto": 0.1}
    dati_premium = {"importo_base": 1000, "servizi_extra": 200}

    preventivatore = Preventivatore(PreventivoBase())

    print("Preventivo Base:", preventivatore.genera_preventivo(dati_base))

    preventivatore.set_strategia(PreventivoSconto())
    print("Preventivo con Sconto:",
preventivatore.genera_preventivo(dati_sconto))

    preventivatore.set_strategia(PreventivoPremium())
    print("Preventivo Premium:",
preventivatore.genera_preventivo(dati_premium))
```

🖥️ Output

```
Preventivo Base: 1220.0
Preventivo con Sconto: 1098.0
Preventivo Premium: 1464.0
```

Il codice implementa un sistema flessibile per il calcolo dei preventivi con diverse strategie:

- PreventivoBase: calcola il prezzo base + IVA
- PreventivoSconto: applica uno sconto prima dell'IVA
- PreventivoPremium: aggiunge servizi extra al prezzo base

La classe Preventivatore funge da contesto e può utilizzare qualsiasi strategia di calcolo in modo intercambiabile.