Soluzione Gestione Accessori Aspirapolvere 🧹

📝 Descrizione

Questo progetto implementa un sistema per gestire gli accessori specifici per diversi modelli di aspirapolvere utilizzando il design pattern Abstract Factory. Ogni modello di aspirapolvere è dotato di tre accessori dedicati: spazzola, filtro e sacchetto.

📐 Struttura del Progetto

```
aspirapolvere_project/
├── factories/
│   ├── abc456_factory.py     # Factory concreta per il modello ABC456
│   ├── accessorio_factory.py # Interfaccia astratta delle factory
│   └── xyz123_factory.py     # Factory concreta per il modello XYZ123
├── models/
│   ├── aspirapolvere.py      # Classe principale Aspirapolvere
│   ├── accessori/
│   │   ├── filtro.py        # Classi per i filtri
│   │   ├── sacchetto.py     # Classi per i sacchetti
│   │   ├── spazzola.py      # Classi per le spazzole
│   │   └── __init__.py      # File package accessori
│   └── __init__.py          # File package models
└── main.py                   # Punto di ingresso dell'applicazione
```

⚙️ Requisiti di Sistema

- Python 3.7 o versioni successive
- Ambiente di sviluppo con pip configurato

🚀 Avvio del Progetto

1. Aprire il terminale
2. Navigare alla directory del progetto
3. Eseguire:

```
python main.py
```

📋 Output di Esempio

```
Aspirapolvere: SuperClean XYZ123
✓ Spazzola compatibile con il modello XYZ123
✓ Filtro compatibile con il modello XYZ123
✓ Sacchetto compatibile con il modello XYZ123

Aspirapolvere: PowerVac ABC456
✓ Spazzola compatibile con il modello ABC456
✓ Filtro compatibile con il modello ABC456
✓ Sacchetto compatibile con il modello ABC456
```

🔧 Design Pattern

- Abstract Factory Pattern per la creazione di famiglie di accessori compatibili

📚 Note Aggiuntive

Il progetto è strutturato per essere facilmente estendibile con nuovi modelli di aspirapolvere e relativi accessori.

Sviluppato come esempio di implementazione del pattern Abstract Factory

🖼️ Ottimizzazione Thumbnail con Proxy

📝 Descrizione del Problema

Il sistema deve gestire le immagini profilo degli utenti con le seguenti necessità:

- Ottimizzare il trasferimento delle immagini riducendone la risoluzione
- Implementare un sistema di cache temporanea (20 minuti)
- Generare thumbnail al primo accesso per utenti esistenti
- Generare thumbnail all'upload per nuovi utenti

🔧 Pattern utilizzato: Proxy

🎯 Perché il Pattern Proxy?

Il Pattern Proxy è la scelta ideale per questo scenario per diversi motivi:

1️⃣ Controllo degli Accessi

- Gestisce l'accesso alle immagini originali
- Implementa la logica di caching
- Controlla la generazione delle thumbnail

2️⃣ Lazy Loading

- Genera le thumbnail solo quando necessario
- Ottimizza le risorse del sistema
- Riduce il carico iniziale

3️⃣ Caching

- Memorizza temporaneamente le immagini accedute di frequente
- Riduce il numero di accessi allo storage
- Migliora le performance del sistema

Componenti Principali:

1. 🎨 **IImageService** (Interface)
   - Definisce il contratto base per tutti i servizi
2. 💾 **RealImageService**
   - Gestisce l'accesso diretto allo storage
   - Recupera le immagini originali
3. 📦 **CacheImageProxy**
   - Implementa la cache temporanea
   - Gestisce la scadenza delle immagini
   - Pulisce automaticamente la cache
4. 🖼️ **ThumbnailProxy**
   - Gestisce la creazione delle thumbnail
   - Memorizza le versioni ridotte
   - Ottimizza il trasferimento

💡 Vantaggi della Soluzione

1. **Separazione delle Responsabilità**
   - Ogni proxy ha un compito specifico
   - Codice più organizzato e manutenibile
2. **Ottimizzazione delle Risorse**

- Riduzione del traffico di rete
- Minore utilizzo dello storage
- Migliori performance

3. **Flessibilità**
    - Facile aggiungere nuove funzionalità
    - Semplice modifica dei comportamenti esistenti

🚀 Esempio di Utilizzo

```
# Esempio base di utilizzo
real_service = RealImageService(storage_path)
cache_proxy = CacheImageProxy(real_service)
thumbnail_proxy = ThumbnailProxy(cache_proxy)

# Richiesta immagine
image = thumbnail_proxy.get_image("user_123_profile.jpg")
```

📈 Conclusioni

Il Pattern Proxy fornisce una soluzione elegante e efficiente per:

- Ottimizzare le risorse
- Migliorare le performance
- Mantenere il codice organizzato
- Gestire in modo trasparente la complessità

🏭 Pattern Composite per il Preventivatore

📝 Descrizione della Soluzione

Per questo problema, il pattern Composite è la scelta ideale perché:

- Abbiamo una struttura ad albero con sezioni, sottosezioni ed elementi base
- Vogliamo trattare sia gli elementi singoli che i gruppi di elementi in modo uniforme
- Il calcolo dei prezzi segue una logica ricorsiva dove il prezzo di una sezione dipende dai suoi componenti

✅ Vantaggi

- Permette di trattare oggetti singoli e composizioni di oggetti in modo uniforme
- Facilita l'aggiunta di nuovi tipi di componenti

- Semplifica la struttura del client
- Rende più facile aggiungere nuove funzionalità all'intera struttura

❌ Svantaggi

- Può rendere il design troppo generico
- Può essere difficile limitare i tipi di componenti che possono essere aggiunti
- Richiede una buona comprensione della ricorsione per l'implementazione

🧬 Struttura del Pattern

- `ComponentePreventivo`: Interfaccia base per tutti gli elementi
- `ElementoBase`: Rappresenta gli elementi foglia (prodotti, servizi)
- `Sezione`: Rappresenta i nodi compositi (sezioni e sottosezioni)

💻 Implementazione Semplificata

```
class ComponentePreventivo(ABC):
    def __init__(self, nome: str, sconto: float = 0):
        self.nome = nome
        self.sconto = sconto

class ElementoBase(ComponentePreventivo):
    def __init__(self, nome: str, prezzo: float, sconto: float = 0):
        super().__init__(nome, sconto)
        self.prezzo = prezzo

class Sezione(ComponentePreventivo):
    def __init__(self, nome: str, sconto: float = 0):
        super().__init__(nome, sconto)
        self.componenti = []
```

🚀 Esempio di Utilizzo

```
preventivo = Sezione("Preventivo Completo")
sez_macchina = Sezione("Macchina base", 0.05)
sez_macchina.aggiungi_componente(ElementoBase("Macchina", 20000))
```

📊 Output di Esempio

```
Prezzo base totale: €20000.00
Prezzo scontato totale: €19000.00
```

📝 Note Implementative

- La struttura permette di aggiungere facilmente nuove sezioni e sottosezioni
- Gli sconti vengono applicati in cascata, rispettando la gerarchia
- Il calcolo dei prezzi è completamente trasparente per il client
- La manutenibilità è elevata grazie alla separazione delle responsabilità

🎭 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

📝 Database Pattern Implementation

🎯 Analisi del Problema

Il problema richiede la creazione di una libreria per interagire con diversi database mantenendo un'interfaccia uniforme. Le caratteristiche chiave sono:

- Selezione dinamica del database
- Interfaccia unificata
- Estensibilità per nuovi database

🎨 Pattern Scelti: Strategy + Factory Method

Ho scelto la combinazione di Strategy e Factory Method perché:

1. Strategy: Permette di definire una famiglia di algoritmi intercambiabili
2. Factory Method: Gestisce la creazione degli oggetti database in modo flessibile

Vantaggi ✅

- Maggiore flessibilità nella selezione del database
- Incapsulamento della logica di creazione
- Facilità di aggiungere nuove strategie
- Separazione tra creazione e utilizzo

Svantaggi ❌

- Maggiore complessità iniziale
- Più classi da gestire
- Overhead di performance minimo

💻 Implementazione

```
# Strategy Pattern - Interfaccia base
class DatabaseStrategy(ABC):
    @abstractmethod
    def find(self, params: Dict) -> Any:
        pass

# Esempio di strategia concreta
class MongoDBStrategy(DatabaseStrategy):
    def find(self, params: Dict) -> Any:
        print(f"MongoDB: Ricerca con parametri {params}")
        return {"result": "MongoDB data"}

# Factory Method base
class DatabaseFactory(ABC):
    @abstractmethod
    def create_database(self, connection_string: str) -> DatabaseStrategy:
        pass
```

🎬 Output dell'implementazione

```
Connesso a MongoDB: mongodb://localhost:27017
MongoDB: Ricerca con parametri {'user': 'mario'}
```

🔍 Spiegazione dei Pattern

Strategy Pattern

- Definisce una famiglia di algoritmi (strategie di database)
- Rende gli algoritmi intercambiabili
- Permette la selezione dell'algoritmo a runtime

Factory Method Pattern

- Incapsula la creazione degli oggetti database
- Permette di estendere facilmente con nuovi tipi di database
- Mantiene il codice pulito e organizzato

Il flusso delle operazioni è:

1. La factory crea l'implementazione appropriata del database
2. Il context utilizza la strategia selezionata
3. Le operazioni vengono eseguite attraverso l'interfaccia comune

📋 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

```python
from abc import ABC, abstractmethod

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

# Esempio di Concrete Strategy
class PreventivoBase(StrategiaPreventivo):
    def calcola_preventivo(self, dati: dict) -> float:
        return dati.get('importo_base', 0) * 1.22  # IVA 22%

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

🖥️ Output

```
Preventivo Base: 1220.0
```

Il codice implementa un sistema flessibile per il calcolo dei preventivi con diverse strategie. L'esempio mostra solo la strategia base che calcola il prezzo con IVA, ma il sistema è progettato per supportare facilmente l'aggiunta di nuove strategie di calcolo.