Soluzione Gestione Accessori Aspirapolvere 🧹

📝 Descrizione

Questo progetto implementa un sistema per gestire gli accessori specifici per diversi modelli di aspirapolvere utilizzando l' Abstract Factory. Ogni modello di aspirapolvere è dotato di tre accessori dedicati, ovverospazzola, 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

🧠 Ragionamento sulla Scelta del Pattern

Perché Abstract Factory?

L'Abstract Factory è stato scelto come pattern ottimale per questo scenario per i seguenti motivi:

1. **Famiglie di Prodotti Correlati**:
   - Ogni modello di aspirapolvere richiede una famiglia specifica di accessori (spazzola, filtro, sacchetto)
   - Gli accessori devono essere compatibili tra loro all'interno dello stesso modello
2. **Incapsulamento della Creazione**:
   - Nasconde i dettagli di implementazione delle classi concrete
   - Centralizza la logica di creazione degli accessori per modello
3. **Garanzia di Compatibilità**:

- Assicura che vengano creati solo accessori compatibili per ciascun modello
- Previene errori di abbinamento tra accessori e modelli

Vantaggi

1. **Manutenibilità**:
   - Facile aggiungere nuovi modelli di aspirapolvere
   - Modifiche localizzate nelle factory concrete
2. **Estensibilità**:
   - Semplice introduzione di nuovi tipi di accessori
   - Supporto per nuovi modelli senza modificare il codice esistente
3. **Coerenza**:
   - Garantisce la creazione di set completi di accessori compatibili
   - Riduce gli errori di configurazione

Svantaggi

1. **Complessità**:
   - Richiede la creazione di molte interfacce e classi
   - Può risultare eccessivo per sistemi semplici
2. **Rigidità della Struttura**:
   - L'aggiunta di nuovi tipi di accessori richiede modifiche all'interfaccia della factory
   - Tutte le factory concrete devono implementare i nuovi metodi

🖼️ Ottimizzazione Thumbnail

🎯 Pattern Scelto: 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

🧠 Ragionamento

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

✅ Vantaggi

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

❌ Svantaggi

- Maggiore complessità iniziale

🚀 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

🏭 Preventivatore

🎯 Pattern Scelto: Composite

📝 Descrizione del Problema

Dovete costruire un preventivatore. Questo preventivatore è diviso in diverse sezioni, ogni sezione può contenere settosezioni e/o elementi base (accessori, servizi, ecc)
Ogni sezione/sottosezione/elemento può avere una quantità e uno sconto. Gli elementi base derivano da un listino e hanno un prezzo base di partenza.
Il software deve mostrare per ogni sezione/sottosezione il suo prezzo base e il prezzo applicato lo sconto.
Il prezzo base di una sezione/sottosezione è dato dalla somma dei prezzi base dei suoi elementi o sottosezioni. Il prezzo scontato è calcolato usando come base la somma dei prezzi scontati delle sue sottosezioni e elementi, sulla quale poi viene applicato lo sconto della sezione.
Il software mostra poi un costo totale con e senza sconti applicati

🧠 Ragionamento

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

- Abbiamo una struttura ad albero con sezioni, sottosezioni ed elementi base
- 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 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

🎭 Azienda e Ruoli

🎯 Pattern Scelto: Decorator

📝 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.

🧠 Ragionamento

- 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

📝 TYpeORM

🎯 Pattern Scelto: Unione di Strategy + Factory Method

📓 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

🧠 Ragionamento

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
3. Permettono di selezionare dinamicamente il database da utilizzare e di creare gli oggetti database in modo flessibile in caso di cambio struttura del progetto.

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 avendo più classi da gestire

💻 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"}
```

```python
# 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

🎯 Pattern Scelto: Facade

📝 Analisi del Problema

Vi viene chiesto di andare ad aggiornare un unico file excel su Google Drive ogni volta che un ordine viene aggiunto, modificato o eliminato. Ad ogni ordine corrisponde una riga.
La libreria che vi permette di scrivere su Drive lavora a basso livello, per ogni dato che dovete andare a scrivere vi serve il link del file (sempre lo stesso), il nome del foglio (sempre lo

stesso), le coordinate su cui lavorare, e infine i dati da scrivere.
Ogni ordine ha il suo codice univoco, e deve rimanere all'oscuro del fatto che lo state esportando su Drive.
Possiamo riassumere i passaggi delle 3 operazioni da svolgere:

- Aggiunta ordine
- Modifica ordine
- eliminare la riga

🧠 Ragionamento

Ho scelto il Facade pattern per questi motivi:

- Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem. ➡️ Questo è il caso d'uso tipico del Facade pattern e il problema che vogliamo risolvere rispecchia questa descrizione.
- Fornisce un'interfaccia semplificata per il sistema complesso di calcolo preventivi
- Nasconde la complessità del sistema sottostante
- Riduce le dipendenze tra il client e i sottosistemi
- Facilita l'utilizzo del sistema di preventivazione

✅ Vantaggi

- Semplifica l'interfaccia per il client
- Disaccoppia il sottosistema dai client
- Fornisce un punto di accesso unificato
- Migliora la manutenibilità del codice

❌ Svantaggi

- Può introdurre un livello di indirezione non necessario se non gestito correttamente

💻 Implementazione

```
# Sottosistemi
class CalcolatoreIVA:
    def calcola_iva(self, importo: float) -> float:
        return importo * 0.22

class CalcolatoreBase:
    def calcola_base(self, dati: dict) -> float:
        return dati.get('importo_base', 0)
```

```python
# Facade
class PreventivatoreFacade:
    def __init__(self):
        self._calcolatore_iva = CalcolatoreIVA()
        self._calcolatore_base = CalcolatoreBase()

    def calcola_preventivo(self, dati: dict) -> float:
        importo_base = self._calcolatore_base.calcola_base(dati)
        iva = self._calcolatore_iva.calcola_iva(importo_base)
        return importo_base + iva
```

Il codice implementa un sistema di preventivazione che utilizza il pattern Facade per nascondere la complessità del calcolo dei preventivi. La facade fornisce un'interfaccia semplice per il client, mentre gestisce internamente l'interazione con i vari sottosistemi di calcolo.