📝 Database Adapter Pattern Implementation

🎯 Analisi del Problema

Il problema richiede la creazione di una libreria che permetta di interagire con diversi tipi di database mantenendo un'interfaccia uniforme. Le caratteristiche principali sono:

- Interfaccia unificata per diversi database
- Facilità di cambio database
- Estensibilità per nuovi database

🎨 Pattern Scelto: Adapter

Ho scelto il pattern Adapter perché:

1. Permette di uniformare interfacce diverse
2. Facilita l'integrazione di nuovi sistemi
3. Mantiene il codice client indipendente dall'implementazione

Vantaggi ✅

- Separazione delle responsabilità
- Facilità di manutenzione
- Estensibilità
- Riusabilità del codice

Svantaggi ❌

- Complessità aggiuntiva
- Overhead di codice
- Possibile impatto sulle performance

💻 Implementazione

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

# Interfaccia target
class DatabaseInterface(ABC):
    @abstractmethod
```

```python
    def find(self, params: Dict) -> Any:
        pass

    @abstractmethod
    def create(self, data: Dict) -> Any:
        pass

# Adapter concreti per diversi database
class MongoDBAdapter(DatabaseInterface):
    def __init__(self, connection_string: str):
        self.connection_string = connection_string
        print(f"Connesso a MongoDB: {connection_string}")

    def find(self, params: Dict) -> Any:
        print(f"MongoDB: Ricerca con parametri {params}")
        return {"result": "MongoDB data"}

    def create(self, data: Dict) -> Any:
        print(f"MongoDB: Creazione record {data}")
        return {"id": "123", "data": data}

class PostgreSQLAdapter(DatabaseInterface):
    def __init__(self, connection_string: str):
        self.connection_string = connection_string
        print(f"Connesso a PostgreSQL: {connection_string}")

    def find(self, params: Dict) -> Any:
        print(f"PostgreSQL: Ricerca con parametri {params}")
        return {"result": "PostgreSQL data"}

    def create(self, data: Dict) -> Any:
        print(f"PostgreSQL: Creazione record {data}")
        return {"id": "456", "data": data}

# Client che usa l'adapter
class DatabaseClient:
    def __init__(self, db_adapter: DatabaseInterface):
        self.db = db_adapter

    def search_data(self, params: Dict) -> Any:
        return self.db.find(params)

    def insert_data(self, data: Dict) -> Any:
        return self.db.create(data)

# Esempio di utilizzo
```

```python
def main():
    # Uso MongoDB
    mongo_db = MongoDBAdapter("mongodb://localhost:27017")
    client = DatabaseClient(mongo_db)

    print("\nTest con MongoDB:")
    client.search_data({"user": "mario"})
    client.insert_data({"name": "Mario", "age": 30})

    # Cambio a PostgreSQL
    postgres_db = PostgreSQLAdapter("postgresql://localhost:5432")
    client = DatabaseClient(postgres_db)

    print("\nTest con PostgreSQL:")
    client.search_data({"user": "luigi"})
    client.insert_data({"name": "Luigi", "age": 25})

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

🎬 Output dell'implementazione

```
Connesso a MongoDB: mongodb://localhost:27017

Test con MongoDB:
MongoDB: Ricerca con parametri {'user': 'mario'}
MongoDB: Creazione record {'name': 'Mario', 'age': 30}

Connesso a PostgreSQL: postgresql://localhost:5432

Test con PostgreSQL:
PostgreSQL: Ricerca con parametri {'user': 'luigi'}
PostgreSQL: Creazione record {'name': 'Luigi', 'age': 25}
```

🔍 Spiegazione del Pattern

L'Adapter pattern è stato scelto perché:

1. `DatabaseInterface` definisce l'interfaccia target comune
2. Gli adapter (`MongoDBAdapter` e `PostgreSQLAdapter`) implementano questa interfaccia
3. Il client può utilizzare qualsiasi database attraverso l'interfaccia comune
4. Nuovi database possono essere facilmente aggiunti creando nuovi adapter

Il flusso delle operazioni è:

1. La libreria base definisce l'interfaccia e gestisce il client
2. Le integrazioni specifiche (adapter) implementano la logica per ogni database
3. Il client interagisce solo con l'interfaccia comune