**Cos’è il pattern DAO?**

Il *DAO (Data Access Object)* è un pattern progettuale che separa la logica di business (business logic) dalla logica di accesso ai dati (data access logic). L'obiettivo principale è evitare che i componenti della logica di business accedano direttamente ai dati (ad esempio, al database). Questo miglioramento dell'architettura porta a un codice più **manutenibile e scalabile**, in cui modifiche alla persistenza dei dati possono essere gestite senza impattare la logica di business.

Grazie al DAO, se dovessimo cambiare il tipo di database (per esempio, passando da Oracle a MySQL) o la tecnologia di accesso ai dati (come da JDBC a JPA/Hibernate), non sarebbe necessario modificare il codice applicativo principale. Sarà sufficiente aggiornare le implementazioni DAO.

**Concetti chiave del pattern DAO**

1. **Modello dati (Model)**: Per ogni tabella del database, viene creata una classe model (POJO) che rappresenta le entità con cui interagiamo.
2. **Interfaccia DAO**: Per ogni tabella, si definisce un'interfaccia DAO che contiene tutti i metodi CRUD (Create, Read, Update, Delete) per la gestione dei dati di quella tabella. Questa interfaccia funge da contratto, stabilendo i metodi che verranno utilizzati per interagire con il database.
3. **Implementazione DAO**: L'interfaccia DAO è implementata da una classe che si occupa di interagire direttamente con il database, utilizzando la tecnologia di persistenza scelta (come JDBC, JPA o Hibernate).
4. **Metodi CRUD**: Questi metodi rappresentano le operazioni principali sul database:
   * **C**reate: Inserisce nuovi dati nel database.
   * **R**ead: Recupera i dati esistenti.
   * **U**pdate: Aggiorna i dati esistenti.
   * **D**elete: Rimuove i dati dal database.

Questi metodi sono presenti in ciascun DAO, e chiunque debba accedere ai dati utilizzerà l’interfaccia DAO corrispondente, senza accedere al database direttamente.

**Architettura del pattern DAO**

1. **Logica di business**: È la logica applicativa, specifica dell’applicazione, che si occupa di elaborare e utilizzare i dati. Si interfaccia con lo strato DAO per ottenere o manipolare i dati necessari.
2. **Strato DAO**: Si occupa dell'accesso ai dati, fornendo metodi CRUD per ciascuna entità. Interagisce con la persistenza dei dati, che può essere un database, e gestisce il collegamento con tecnologie come JDBC o Hibernate.
3. **Database (persistenza)**: È il sistema di memorizzazione dei dati. I DAO fungono da intermediari tra il database e la logica di business, incapsulando le operazioni di accesso ai dati e rendendo l’applicazione più modulare.

**Vantaggi dell'uso del pattern DAO**

* **Manutenibilità**: Isolando la logica di accesso ai dati, è più semplice aggiornare o sostituire la tecnologia di persistenza senza modificare il resto dell'applicazione.
* **Riutilizzabilità**: I DAO possono essere riutilizzati in più punti dell'applicazione, riducendo il codice duplicato.
* **Testabilità**: L'uso di interfacce rende più semplice testare il codice della logica di business, poiché è possibile simulare i DAO senza accedere al database reale.

**Esempio di utilizzo del pattern DAO in una libreria**

Supponiamo di dover gestire una libreria. L'entità principale sarà il **Libro**, e quindi:

1. Creiamo una classe Libro che rappresenta il modello con i campi necessari (come titolo e isbn).

package model;

​

public class Libro {

  private String titolo;

  private String isbn;

​

   // Costruttore, getter e setter

  public Libro(String titolo, String isbn) {

      this.titolo = titolo;

      this.isbn = isbn;

   }

​

  public String getTitolo() {

      return titolo;

   }

​

  public void setTitolo(String titolo) {

      this.titolo = titolo;

   }

​

  public String getIsbn() {

      return isbn;

   }

​

  public void setIsbn(String isbn) {

      this.isbn = isbn;

   }

}

​

1. Definiamo un’interfaccia LibroDao con i metodi CRUD

package dao;

​

import model.Libro;

import java.util.List;

​

public interface LibroDao {

  void aggiungiLibro(Libro libro);

  Libro cercaLibroPerIsbn(String isbn);

  List<Libro> trovaTutti();

   boolean aggiornaLibro(Libro libro);

   boolean eliminaLibro(String isbn);

}

​

1. Implementiamo LibroDao in una classe LibroDaoImpl che conterrà la logica per interagire con il database.

package dao.impl;

​

import dao.LibroDao;

import model.Libro;

import java.util.ArrayList;

import java.util.List;

​

public class LibroDaoImpl implements LibroDao {

  private List<Libro> libri = new ArrayList<>();

​

  @Override

  public void aggiungiLibro(Libro libro) {

      libri.add(libro);

   }

​

  @Override

  public Libro cercaLibroPerIsbn(String isbn) {

      for (Libro libro : libri) {

          if (libro.getIsbn().equals(isbn)) {

              return libro;

           }

       }

      return null;

   }

​

  @Override

  public List<Libro> trovaTutti() {

      return new ArrayList<>(libri);

   }

​

  @Override

  public boolean aggiornaLibro(Libro libro) {

      for (int i = 0; i < libri.size(); i++) {

          if (libri.get(i).getIsbn().equals(libro.getIsbn())) {

              libri.**set**(i, libro);

              return true;

           }

       }

      return false;

   }

​

  @Override

  public boolean eliminaLibro(String isbn) {

      return libri.removeIf(libro -> libro.getIsbn().equals(isbn));

   }

}

​

**In sintesi:**

Il pattern DAO:

* Incapsula i dati in oggetti dedicati, noti come POJO (Plain Old Java Object).
* Fornisce un modo per trasferire oggetti attraverso la rete tra client e server.
* Separa la logica di business dall'accesso ai dati, migliorando modularità, manutenibilità e testabilità.

**Cos’è il pattern DTO?**

Il **pattern DTO (Data Transfer Object)** è un design pattern usato per trasferire dati tra diversi strati di un’applicazione, come ad esempio tra il client e il server, o tra diversi moduli. L'idea principale dietro il DTO è di avere un oggetto leggero e specializzato per il trasferimento dei dati, riducendo la quantità di comunicazioni necessarie e migliorando l'efficienza dell'applicazione.

**Perché usare il pattern DTO?**

Il pattern DTO è utile quando:

* Dobbiamo trasferire dati attraverso la rete, come in un'applicazione distribuita o client-server.
* Vogliamo evitare di esporre direttamente le entità del database (i model) agli altri strati dell'applicazione, specialmente nei sistemi che usano il pattern DAO.
* Vogliamo aumentare l'efficienza della comunicazione, raggruppando i dati necessari in un unico oggetto.

Usando i DTO, possiamo quindi definire un’interfaccia di scambio dati senza esporre la logica o la struttura dei dati interni.

**Differenze tra DAO e DTO**

* **DAO**: Accede e modifica direttamente i dati nel database. La sua funzione principale è fornire metodi CRUD per manipolare le entità di persistenza.
* **DTO**: Trasporta dati tra i livelli di un'applicazione. Non ha logica di business o accesso ai dati, ma solo proprietà (attributi) e, in alcuni casi, metodi di conversione (ad esempio, da e verso il model).

**Componenti principali del pattern DTO**

1. **Classi DTO**: Ogni DTO è una classe con attributi che rappresentano i dati da trasferire. È solitamente una classe leggera, senza metodi di business e solo con getter, setter e, talvolta, costruttori. Un DTO può aggregare dati da più oggetti model.
2. **Mapper**: È comune avere una classe Mapper che si occupa di convertire le entità (model) in DTO e viceversa. Questo consente di trasformare i dati recuperati dai DAO in oggetti DTO da restituire al client, e di convertire i DTO ricevuti dal client in model da salvare nel database.

**Vantaggi dell’uso dei DTO**

* **Migliore efficienza**: Riduce il numero di chiamate tra client e server, specialmente in ambienti remoti.
* **Sicurezza dei dati**: Consente di nascondere i dettagli interni delle entità di persistenza, esponendo solo i dati necessari.
* **Migliore manutenibilità**: Modifiche al database o alla logica di business non influiscono direttamente sui DTO, poiché i dati sono incapsulati e separati.

**Esempio di utilizzo del pattern DTO**

Immaginiamo di dover gestire una libreria con il pattern DAO per accedere al database e il pattern DTO per trasferire i dati tra i livelli dell’applicazione.

Supponiamo di avere un'entità Libro con alcuni campi di dettaglio, ma al client serve solo una parte delle informazioni. Creeremo quindi un DTO per trasferire solo i dati rilevanti.

**Classe model**: Libro

package model;

​

public class Libro {

  private String titolo;

  private String isbn;

  private String autore;

  private int annoPubblicazione;

​

   // Costruttore, getter e setter

}

​ **Classe DTO:** LibroDTO

package dto;

​

public class LibroDTO {

  private String titolo;

  private String isbn;

​

  public LibroDTO(String titolo, String isbn) {

      this.titolo = titolo;

      this.isbn = isbn;

   }

​

   // Getter e setter

  public String getTitolo() {

      return titolo;

   }

​

  public String getIsbn() {

      return isbn;

   }

}

​

**Classe Mapper:** LibroMapper

package mapper;

​

import model.Libro;

import dto.LibroDTO;

​

public class LibroMapper {

​

  public static LibroDTO toDTO(Libro libro) {

      return new LibroDTO(libro.getTitolo(), libro.getIsbn());

   }

​

  public static Libro toModel(LibroDTO dto) {

      return new Libro(dto.getTitolo(), dto.getIsbn(), null, 0); // campi opzionali come autore e anno

   }

}

**DAO e logica di utilizzo**: Nell'applicazione, il DAO recupera l'oggetto Libro dal database, poi la classe Mapper lo trasforma in un LibroDTO per essere trasferito al client.

**In sintesi**

Il pattern DTO:

* Incapsula e trasferisce dati: È usato per trasferire dati tra i livelli di un’applicazione, separando la logica di persistenza dalla logica di business.
* Evita l'accesso diretto alle entità: Protegge i dati interni dell’applicazione esponendo solo ciò che è necessario.
* Ottimizza le comunicazioni: Riduce le interazioni necessarie tra client e server in applicazioni distribuite.

Il pattern DTO, abbinato al pattern DAO, contribuisce a una struttura solida, manutenibile e sicura per la gestione dei dati in applicazioni complesse.