Padrão DAO

Data Access Object

Aula 07 · Camada de persistência desacoplada e reutilizável

Práticas de POO

Ciência da Computação · Graduação

`interface DAO · DAOImpl · DAOFactory · Generics · PostgreSQL`

AULA 07

O que você vai aprender hoje

01

O problema do código repetido

Entender por que SQL espalhado é um problema.

02

O que é o padrão DAO

Conhecer o conceito e seus três pilares.

03

Interface + Implementação

Separar o contrato (DAO) da implementação (DAOImpl).

04

DAOFactory

Centralizar a criação dos DAOs.

05

DAO genérico com Generics

Reutilizar CRUD entre entidades (avançado).

06

Transações e boas práticas

Commit/rollback e tratamento de exceções.

INICIANTE

O problema: SQL espalhado

Sem o DAO, o código de acesso ao banco fica **repetido** em várias classes.
Qualquer mudança no banco exige alterar tudo.

SQL duplicado em cada tela

```
01 public class TelaProduto extends JFrame {
02     void salvar() throws SQLException {
03         Connection c = DriverManager.getConnection(url, u, s);
04         Statement st = c.createStatement();
05         st.executeUpdate("INSERT INTO produto ...");
06     }
07 }
08 public class TelaPedido extends JFrame {
09     void salvar() throws SQLException {
10         Connection c = DriverManager.getConnection(url, u, s);
11         // o mesmo código repetido outra vez!
12     }
13 }
```

INICIANTE

A solução: centralizar no DAO

O DAO **centraliza** todo o acesso a dados em um único lugar.
As demais classes só chamam os métodos do DAO.

SEM DAO

- SQL repetido em cada tela
- Mudança no banco = alterar tudo
- Difícil de testar

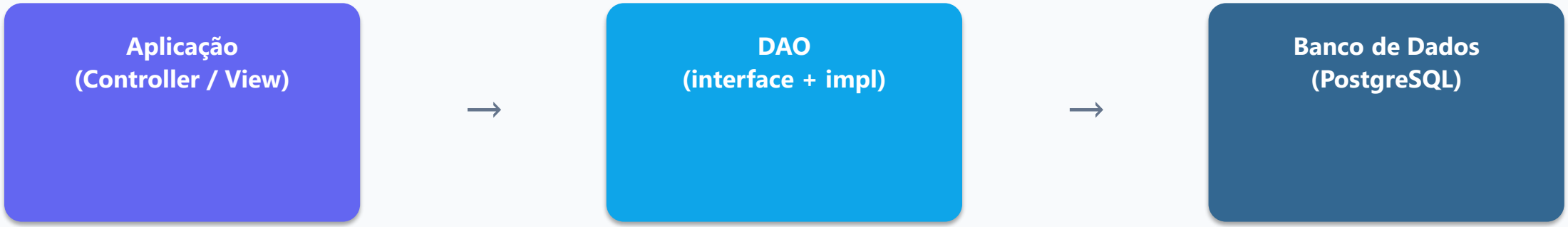
COM DAO

- SQL em um único lugar
- Mudança no banco = 1 classe
- Fácil de testar e reutilizar

CONCEITO

O que é o padrão DAO?

DAO (Data Access Object) é um padrão que **isola o acesso a dados** em uma camada própria, escondendo os detalhes do banco (SQL, conexão, driver).



A aplicação nunca fala com o banco diretamente

Ela chama métodos do DAO (inserir, listar, excluir) e o DAO cuida do SQL e da conexão.

CONCEITO

Benefícios do padrão DAO

Desacoplamento

A regra de negócio não depende do banco de dados.

Reutilização

O mesmo DAO é usado por várias telas e controllers.

Manutenção

Mudou o banco? Altera apenas a implementação do DAO.

Testabilidade

É fácil trocar o DAO por uma versão de teste (mock).

ESTRUTURA

Os 3 componentes do padrão

Interface DAO

Define o contrato: quais operações existem (inserir, listar...).

Implementação

DAOImpl: escreve o SQL e a conexão de verdade.

DAOFactory

Fábrica que cria e entrega as implementações prontas.

POR QUE SEPARAR INTERFACE E IMPLEMENTAÇÃO?

A aplicação depende apenas da interface. Assim, podemos trocar a implementação (outro banco, outro driver) sem mudar quem usa.

ESTRUTURA

ConnectionFactory: a base de tudo

Antes do DAO, centralizamos a conexão em uma fábrica. Todo DAO usa este único ponto de acesso.

ConnectionFactory.java

```java
package util;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {
    private static final String URL = "jdbc:postgresql://localhost:5432/escola";
    private static final String USER = "postgres";
    private static final String PASS = "123456";

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USER, PASS);
    }
}
```

ESTRUTURA

A interface ProdutoDAO: o contrato

A interface declara **o que** o DAO faz, sem dizer **como**.
Quem usa só conhece estes métodos.

ProdutoDAO.java — contrato

```java
package dao;
import java.util.List;
import model.Produto;

public interface ProdutoDAO {
    boolean inserir(Produto p);
    boolean atualizar(Produto p);
    boolean excluir(int id);
    Produto buscarPorId(int id);
    List<Produto> listar();
}
```

ESTRUTURA

Organização em pacotes

Estrutura do projeto

```
src/
├── model/
│   └── Produto.java            // entidade (dados)
├── dao/
│   ├── ProdutoDAO.java         // interface (contrato)
│   ├── ProdutoDAOImpl.java     // implementação (SQL)
│   └── DAOFactory.java         // fábrica de DAOs
└── util/
    └── ConnectionFactory.java  // conexão centralizada
```

MODEL

Produto.java: a entidade de dados

A entidade representa os **dados** da tabela.
Ela não contém SQL nem lógica de acesso — apenas atributos e getters/setters.

Produto.java — entidade

```java
package model;
public class Produto {
    private int id;
    private String nome;
    private double preco;
    private int estoque;
    public Produto(String nome, double preco, int estoque) {
        this.nome = nome;
        this.preco = preco;
        this.estoque = estoque;
    }
    // getters e setters omitidos
}
```

IMPLEMENTAÇÃO

ProdutoDAOImpl: a classe que faz o SQL

A implementação **cumpre o contrato** da interface.
É aqui que vive todo o JDBC: conexão, SQL e ResultSet.

ProdutoDAOImpl.java — estrutura

```java
package dao;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
import model.Produto;

public class ProdutoDAOImpl implements ProdutoDAO {
    // cada método usa try-with-resources
    // e a ConnectionFactory para obter a conexão
}
```

IMPLEMENTAÇÃO

inserir(): criando um registro

O método recebe a entidade e usa **PreparedStatement** para enviar os valores com segurança.

ProdutoDAOImpl.java — inserir()

```java
@Override
public boolean inserir(Produto p) {
    String sql = "INSERT INTO produto (nome,preco,estoque) VALUES (?,?,?)";
    try (Connection c = ConnectionFactory.getConnection();
         PreparedStatement ps = c.prepareStatement(sql)) {
        ps.setString(1, p.getNome());
        ps.setDouble(2, p.getPreco());
        ps.setInt(3, p.getEstoque());
        return ps.executeUpdate() > 0;
    } catch (SQLException e) { e.printStackTrace(); return false; }
}
```

IMPLEMENTAÇÃO

atualizar(): modificando um registro

O UPDATE usa **4 parâmetros**: os 3 campos novos e o id na cláusula WHERE.

ProdutoDAOImpl.java — atualizar()

```java
@Override
public boolean atualizar(Produto p) {
    String sql = "UPDATE produto SET nome=?, preco=?, estoque=? WHERE id=?";
    try (Connection c = ConnectionFactory.getConnection();
         PreparedStatement ps = c.prepareStatement(sql)) {
        ps.setString(1, p.getNome());
        ps.setDouble(2, p.getPreco());
        ps.setInt(3, p.getEstoque());
        ps.setInt(4, p.getId());
        return ps.executeUpdate() > 0;
    } catch (SQLException e) { e.printStackTrace(); return false; }
}
```

IMPLEMENTAÇÃO

excluir(): removendo um registro

O DELETE recebe apenas o **id** e o usa no WHERE.

ProdutoDAOImpl.java — excluir()

```java
@Override
public boolean excluir(int id) {
    String sql = "DELETE FROM produto WHERE id=?";
    try (Connection c = ConnectionFactory.getConnection();
         PreparedStatement ps = c.prepareStatement(sql)) {
        ps.setInt(1, id);
        return ps.executeUpdate() > 0;
    } catch (SQLException e) { e.printStackTrace(); return false; }
}
```

IMPLEMENTAÇÃO

buscarPorId(): lendo um registro

Um SELECT que retorna **um objeto**, convertido pelo método montar().

ProdutoDAOImpl.java — buscarPorId()

```java
@Override
public Produto buscarPorId(int id) {
    String sql = "SELECT * FROM produto WHERE id=?";
    try (Connection c = ConnectionFactory.getConnection();
         PreparedStatement ps = c.prepareStatement(sql)) {
        ps.setInt(1, id);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) return montar(rs);
    } catch (SQLException e) { e.printStackTrace(); }
    return null;
}
```

IMPLEMENTAÇÃO

listar(): lendo todos os registros

Percorre o **ResultSet** com next() e monta uma lista de entidades.

ProdutoDAOImpl.java — listar()

```java
@Override
public List<Produto> listar() {
    List<Produto> lista = new ArrayList<>();
    String sql = "SELECT * FROM produto ORDER BY nome";
    try (Connection c = ConnectionFactory.getConnection();
         PreparedStatement ps = c.prepareStatement(sql)) {
        ResultSet rs = ps.executeQuery();
        while (rs.next()) lista.add(montar(rs));
    } catch (SQLException e) { e.printStackTrace(); }
    return lista;
}
```

IMPLEMENTAÇÃO

montar(): ResultSet → objeto

Método privado que **mapeia cada linha** do ResultSet para um objeto Produto.
Evita repetição de código.

ProdutoDAOImpl.java — montar()

```
01  private Produto montar(ResultSet rs) throws SQLException {
02      Produto p = new Produto(rs.getString("nome"),
03              rs.getDouble("preco"), rs.getInt("estoque"));
04      p.setId(rs.getInt("id"));
05      return p;
06  }
```

INTEGRAÇÃO

O Controller usa apenas a interface

O Controller depende da **interface** ProdutoDAO, nunca da implementação.
Isso é o desacoplamento do padrão.

ProdutoController.java

```java
package controller;
import dao.ProdutoDAO;
import dao.ProdutoDAOImpl;
import model.Produto;
public class ProdutoController {
    private final ProdutoDAO dao = new ProdutoDAOImpl();

    public boolean salvar(Produto p) {
        if (p.getNome() == null || p.getNome().isBlank()) return false;
        if (p.getPreco() <= 0) return false;
        return p.getId() == 0 ? dao.inserir(p) : dao.atualizar(p);
    }
}
```

INTEGRAÇÃO

Testando o DAO completo

Veja como o código cliente fica **limpo**: nenhum SQL aparece aqui.

TesteDAO.java

```java
public static void main(String[] args) {
    ProdutoDAO dao = new ProdutoDAOImpl();
    dao.inserir(new Produto("Notebook", 3500.0, 10));
    dao.inserir(new Produto("Mouse", 80.0, 25));

    for (Produto p : dao.listar())
        System.out.println(p.getId() + " " + p.getNome());

    Produto p = dao.buscarPorId(1);
    p.setPreco(3200.0);
    dao.atualizar(p);
    dao.excluir(2);
}
```

AVANÇADO

DAO genérico: escreva o CRUD uma vez só

Com **Generics**, criamos um DAO "pai" que implementa o CRUD uma única vez.
Cada entidade só herda e reaproveita.



Vantagem

Adicionar uma nova entidade (ex.: ClienteDAO) não exige reescrever INSERT/SELECT/UPDATE/DELETE — basta herdar o DAO genérico e informar a tabela.

AVANÇADO

GenericDAO: o contrato genérico

A interface usa o tipo **T** como coringa: ela serve para qualquer entidade.

GenericDAO.java

```java
package dao.generico;
import java.util.List;

public interface GenericDAO<T> {
    boolean inserir(T entidade);
    boolean atualizar(T entidade);
    boolean excluir(int id);
    T buscarPorId(int id);
    List<T> listar();
}
```

AVANÇADO

GenericDAOImpl: a classe abstrata

Ela implementa o CRUD e deixa **dois métodos abstratos** para o DAO concreto preencher.

GenericDAOImpl.java — estrutura

```java
package dao.generico;
import java.sql.*;
import util.ConnectionFactory;

public abstract class GenericDAOImpl<T> implements GenericDAO<T> {
    protected abstract String nomeTabela();

    protected abstract void preencher(PreparedStatement ps, T e)
            throws SQLException;
}
```

AVANÇADO

GenericDAOImpl: inserir() reutilizável

O método chama **preencher()**, que o DAO concreto implementa.
O SQL é montado com **nomeTabela()**

.

GenericDAOImpl.java — inserir()

```java
@Override
public boolean inserir(T e) {
    String sql = "INSERT INTO " + nomeTabela() + " VALUES (DEFAULT,?,?)";
    try (Connection c = ConnectionFactory.getConnection();
         PreparedStatement ps = c.prepareStatement(sql)) {
        preencher(ps, e);
        return ps.executeUpdate() > 0;
    } catch (SQLException ex) {
        throw new DAOException("Erro ao inserir", ex);
    }
}
```

AVANÇADO

ProdutoDAO: herdando o genérico

O DAO concreto só informa a **tabela** e como **preencher** os parâmetros.
Todo o resto vem herdado.

ProdutoDAO.java

```java
package dao;
import dao.generico.GenericDAOImpl;
import java.sql.*;
import model.Produto;
public class ProdutoDAO extends GenericDAOImpl<Produto> {
    @Override protected String nomeTabela() { return "produto"; }
    @Override
    protected void preencher(PreparedStatement ps, Produto p) throws SQLException {
        ps.setString(1, p.getNome());
        ps.setDouble(2, p.getPreco());
        ps.setInt(3, p.getEstoque());
    }
}
```

AVANÇADO

DAOFactory: criando DAOs num só lugar

A fábrica **centraliza a criação** dos DAOs.
O padrão usado aqui é **Singleton** (uma instância por DAO).

DAOFactory.java

```java
package dao;

public class DAOFactory {
    private static ProdutoDAO produtoDAO;

    public static ProdutoDAO getProdutoDAO() {
        if (produtoDAO == null) produtoDAO = new ProdutoDAO();
        return produtoDAO;
    }
}
```

AVANÇADO

Transações: tudo ou nada

Com **setAutoCommit(false)**, várias operações viram uma unidade: ou todas confirmam (commit) ou todas desfazem (rollback).

Transação com commit/rollback

```
01  public boolean transferirEstoque(int origem, int destino, int qtd) {
02      String sql = "UPDATE produto SET estoque = estoque + ? WHERE id = ?";
03      try (Connection c = ConnectionFactory.getConnection()) {
04          c.setAutoCommit(false);
05          try (PreparedStatement ps = c.prepareStatement(sql)) {
06              ps.setInt(1, -qtd); ps.setInt(2, origem);  ps.executeUpdate();
07              ps.setInt(1, +qtd); ps.setInt(2, destino); ps.executeUpdate();
08              c.commit();
09              return true;
10          } catch (SQLException e) { c.rollback(); return false; }
11      } catch (SQLException e) { throw new DAOException(e); }
12  }
```

AVANÇADO

DAOException: tratando erros com robustez

Em vez de só imprimir o erro, o DAO lança uma **exceção própria**, deixando a decisão para as camadas superiores.

DAOException.java + uso

```java
public class DAOException extends RuntimeException {
    public DAOException(String msg, Throwable causa) {
        super(msg, causa);
    }
}

// uso no DAO: em vez de apenas imprimir o erro...
catch (SQLException e) {
    throw new DAOException("Erro ao inserir produto", e);
}
```

BOAS PRÁTICAS

Escrevendo DAOs com qualidade

Use try-with-resources

Fecha Connection, Statement e ResultSet automaticamente.

Prefira PreparedStatement

Evita SQL Injection e permite parâmetros dinâmicos.

Uma exceção própria

Lance DAOException em vez de engolir SQLException.

Centralize na Factory

Crie os DAOs em um único ponto (DAOFactory).

Reutilize com Generics

Um DAO genérico evita repetir o CRUD para cada entidade.

Transações quando necessário

Use commit/rollback para operações atômicas.

ATENÇÃO

Erros comuns ao usar DAO

Esquecer de fechar recursos

Conexões abertas esgotam o banco. Use try-with-resources.

Concatenar valores no SQL

Abre brecha para SQL Injection. Use os ? do PreparedStatement.

Engolir a SQLException

Um catch vazio esconde o erro. Lance DAOException.

SQL na View ou no Controller

O acesso a dados deve ficar só no DAO (camada Model).

Índice de parâmetro errado

Os ? começam em 1 e seguem a ordem do SQL.

Ignorar o retorno de executeUpdate

O int indica se a operação realmente afetou linhas.

MAPA

O padrão DAO em uma visão

`Interface (ProdutoDAO)`	Declara as operações (o quê), sem nenhum SQL.
`Implementação (ProdutoDAOImpl)`	Executa o SQL e a conexão via JDBC (o como).
`Fábrica (DAOFactory)`	Centraliza e entrega as instâncias dos DAOs.
`Genérico (GenericDAO<T>)`	Reaproveita o CRUD entre entidades com Generics.

Resultado: a aplicação depende só da interface — trocar o banco não quebra nada.

PRÁTICA

Exercícios: aplique o padrão DAO

EXERCÍCIO 1 Fácil

ClienteDAO: interface + implementação

Crie a tabela cliente (id SERIAL, nome VARCHAR, email VARCHAR) no PostgreSQL. Defina a interface ClienteDAO com inserir, atualizar, excluir, buscarPorId e listar, e implemente ClienteDAOImpl usando PreparedStatement e try-with-resources, seguindo o modelo de ProdutoDAOImpl.

DICAS

- Reutilize a ConnectionFactory da aula
- Um método por vez: teste inserir antes de listar
- Lance DAOException em vez de imprimir o erro

PRÁTICA

Exercício 2: DAO genérico em ação

EXERCÍCIO 2 Médio

Herdando o GenericDAOImpl

Crie a tabela pedido (id SERIAL, descricao VARCHAR, valor NUMERIC). Faça PedidoDAO herdar GenericDAOImpl<Pedido>, implementando apenas nomeTabela() e preencher(). Obtenha os DAOs pela DAOFactory e teste listar() e buscarPorId() no main.

REQUISITOS

- Nenhum SQL repetido: herde o CRUD pronto
- Adicione getPedidoDAO() na DAOFactory
- Compare o tamanho com um DAO sem Generics

PRÁTICA

Exercício 3: desafio com transação

EXERCÍCIO 3 Desafio

Venda atômica: tudo ou nada

Implemente registrarVenda(produtoId, qtd): em uma única transação, diminua o estoque do produto (UPDATE) e insira a venda (INSERT). Use setAutoCommit(false), commit() no sucesso e rollback() no erro, lançando DAOException. Monte um menu no console (Scanner) que chame os DAOs.

DESAFIOS EXTRAS

- Valide estoque antes de vender (buscarPorId)
- Imprima o estado antes e depois da transação
- Trate a exceção no menu, não no DAO

Resumo da aula

Sua persistência agora é desacoplada

- Padrão DAO: separar o acesso a dados
- Interface × implementação (contrato)
- DAOFactory e DAO genérico com Generics
- Transações: commit e rollback
- Tratamento robusto com DAOException

PRÓXIMA AULA

Integrando Swing + JDBC + MVC

Vamos juntar tudo: uma aplicação gráfica completa, com DAOs, transações e o padrão MVC organizando as camadas.

```
view + controller + dao
```

Obrigado!

Aula 07 · Padrão DAO (Data Access Object)

Práticas de POO

Prof. Me. Célio Ricardo Castelano · Ciência da Computação