Padrão MVC

Model · View · Controller

Aula 06 · Model = entidade (POJO); View limpa; Controller orquestra (e persiste, sem DAO)

Práticas de POO

Ciência da Computação · Graduação

```
model = POJO · view = Swing · controller = regras + JDBC (até o DAO da Aula 07)
```

AULA 06

O que você vai aprender hoje

01 **Separar responsabilidades**
View, Model e Controller com papéis bem definidos.

02 **Model = entidade (POJO)**
Apenas dados: atributos + getters/setters. Sem lógica, sem SQL.

03 **View limpa**
Swing só exibe e captura; nunca faz SQL nem regra de negócio.

04 **Controller orquestrador**
Valida (regras) e, NA FALTA DO DAO, executa o JDBC.

05 **Fluxo de uma requisição**
View → Controller → banco → de volta à View.

06 **Caminho para o DAO**
Por que o JDBC no Controller é um degrau até a Aula 07.

INICIANTE

O problema: tudo misturado na View

Sem camadas, a tela acumula **SQL + regra + apresentação**.
Qualquer mudança vira um caos.

O que NÃO fazer

```
01  // ❌ ANTI-PADRÃO: SQL e regra DENTRO da View
02  btnSalvar.addActionListener(e -> {
03      if (txtNome.getText().isBlank()) { /* valida aqui */ }
04      Connection c = DriverManager.getConnection(url, u, s);
05      PreparedStatement ps = c.prepareStatement(
06          "INSERT INTO aluno (nome, email) VALUES (?,?)");
07      ps.setString(1, txtNome.getText());   // View conhecendo o banco!
08      ps.executeUpdate();
09  });
```

CONCEITO

MVC: três papéis, três camadas

MODEL	VIEW	CONTROLLER
Entidade (POJO): só dados. Sem lógica, sem SQL.	Swing: exibe e captura. Nunca faz SQL nem regra.	Orquestra: valida (regras) e, sem DAO, executa o JDBC.

Convenção adotada no curso

Usamos o Model 'anêmico' (POJO): a entidade só carrega dados.

As regras e a orquestração ficam no Controller — e o acesso a dados, sem DAO, também.

MODEL

Model = a entidade (POJO), nada mais

O Model representa os **dados** do domínio.
Nesta convenção ele **não** contém lógica de negócio nem acesso a banco.

O QUE O MODEL TEM

- Atributos privados (id, nome, email...)
- Getters e setters
- Construtores e toString()

O QUE O MODEL NÃO TEM

- SQL / JDBC / Hibernate
- Validações de negócio
- Chamadas à View ou ao Controller

MODEL

Aluno.java: o POJO puro

model/Aluno.java

```java
package model;
// MODEL = entidade (POJO): só dados. Sem lógica e sem banco.
public class Aluno {
    private Integer id;
    private String nome;
    private String email;
    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }
    public String getNome() { return nome; }
    public void setNome(String n) { this.nome = n; }
    public String getEmail() { return email; }
    public void setEmail(String e) { this.email = e; }
}
```

VIEW

View: exibir e capturar, só isso

A View monta a tela Swing e **delega tudo ao Controller**.
Ela não decide nada e não conhece o banco.

RESPONSABILIDADES

- Montar componentes (campos, botões, lista)
- Capturar entradas e cliques
- Exibir os dados que o Controller traz

PROIBIDO NA VIEW

- SQL / DriverManager / PreparedStatement
- Validações de regra de negócio
- Criar ou manipular entidades diretamente

VIEW

AlunoView: estrutura (1/5)

view/AlunoView.java — estrutura

```java
package view;
import javax.swing.*;
import java.util.ArrayList;
import java.util.List;
import model.Aluno;
import controller.AlunoController;
public class AlunoView extends JFrame {
    private JTextField txtNome, txtEmail;
    private JButton btnSalvar, btnAtualizar, btnExcluir;
    private JList<String> lista;
    private List<Aluno> atual = new ArrayList<>();
    private final AlunoController ctrl;
    // VIEW: só exibe/captura. NUNCA faz SQL nem regra de negócio.
}
```

VIEW

AlunoView: initComponents (2/5)

view/AlunoView.java — initComponents()

```
01      private void initComponents() {
02          setLayout(new BorderLayout());
03          txtNome = new JTextField(12); txtEmail = new JTextField(12);
04          btnSalvar = new JButton("Salvar"); btnAtualizar = new JButton("Atualizar");
05          btnExcluir = new JButton("Excluir"); lista = new JList<>();
06          JPanel topo = new JPanel();
07          topo.add(new JLabel("Nome:")); topo.add(txtNome);
08          topo.add(new JLabel("Email:")); topo.add(txtEmail);
09          topo.add(btnSalvar); topo.add(btnAtualizar); topo.add(btnExcluir);
10          add(topo, BorderLayout.NORTH);
11          add(new JScrollPane(lista), BorderLayout.CENTER);
12      }
```

VIEW

AlunoView: construtor e ações (3/5)

view/AlunoView.java — construtor + acoes()

```
01      public AlunoView(AlunoController ctrl) {
02          super("Cadastro de Alunos (MVC)");
03          this.ctrl = ctrl;
04          initComponents();
05          acoes();
06      }
07      private void acoes() {
08          btnSalvar.addActionListener(e -> salvar());
09          btnAtualizar.addActionListener(e -> listar());
10          btnExcluir.addActionListener(e -> excluir());
11      }
```

VIEW

AlunoView: delegando ao Controller (4/5)

view/AlunoView.java — salvar() e excluir()

```
01      private void salvar() {
02          ctrl.salvar(txtNome.getText(), txtEmail.getText());
03          txtNome.setText(""); txtEmail.setText("");
04          listar();
05      }
06      private void excluir() {
07          int i = lista.getSelectedIndex();
08          if (i >= 0) ctrl.excluir(atual.get(i).getId());
09          listar();
10      }
```

VIEW

AlunoView: listar + Main (5/5)

view/AlunoView.java — listar() + Main.java

```
01      private void listar() {
02          atual = ctrl.listar();
03          DefaultListModel<String> m = new DefaultListModel<>();
04          for (Aluno a : atual) m.addElement(a.getNome() + " | " + a.getEmail());
05          lista.setModel(m);
06      }
07  }
08  public class Main {
09      public static void main(String[] args) {
10          AlunoController ctrl = new AlunoController();
11          SwingUtilities.invokeLater(() -> new AlunoView(ctrl).setVisible(true));
12      }
13  }
```

CONTROLLER

Controller: o orquestrador

O Controller recebe da View, **valida (regras)** e coordena o fluxo.
Sem DAO, ele também executa o JDBC.

SEMPRE DO CONTROLLER

- Receber chamadas da View
- Aplicar regras de negócio (validar)
- Decidir o que a View exibe depois

NESTA AULA (SEM DAO)

- Abrir a Connection (JDBC)
- Executar INSERT/SELECT/DELETE
- Na Aula 07 isso vira o DAO

CONTROLLER

AlunoController: estrutura e conexão (1/4)

controller/AlunoController.java — estrutura

```java
package controller;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
import model.Aluno;
// CONTROLLER orquestra. SEM DAO, ele também acessa o banco (Aula 07 extrai p/ DAO).
public class AlunoController {
    private static final String URL = "jdbc:postgresql://localhost:5432/escola";
    private static final String USER = "postgres";
    private static final String PASS = "123456";
    private Connection con() throws SQLException {
        return DriverManager.getConnection(URL, USER, PASS);
    }
```

CONTROLLER

Regras + salvar (2/4)

controller/AlunoController.java — validar() + salvar()

```
01      private boolean validar(Aluno a) {   // REGRA de negócio fica aqui
02          return a.getNome()!=null && !a.getNome().isBlank() && a.getEmail()!=null &&
a.getEmail().contains("@");
03      }
04      public void salvar(String nome, String email) {
05          Aluno a = new Aluno();
06          a.setNome(nome); a.setEmail(email);
07          if (!validar(a)) return;
08          String sql = "INSERT INTO aluno (nome, email) VALUES (?,?)";
09          try (Connection c = con(); PreparedStatement ps = c.prepareStatement(sql)) {
10              ps.setString(1, a.getNome()); ps.setString(2, a.getEmail());
11              ps.executeUpdate();
12          } catch (SQLException e) { throw new RuntimeException(e); }
13      }
```

CONTROLLER

listar(): lendo do banco (3/4)

controller/AlunoController.java — listar()

```java
public List<Aluno> listar() {
    List<Aluno> l = new ArrayList<>();
    String sql = "SELECT id, nome, email FROM aluno ORDER BY nome";
    try (Connection c = con(); PreparedStatement ps = c.prepareStatement(sql)) {
        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            Aluno a = new Aluno();
            a.setId(rs.getInt("id"));
            a.setNome(rs.getString("nome"));
            a.setEmail(rs.getString("email"));
            l.add(a);
        }
    } catch (SQLException e) { throw new RuntimeException(e); }
    return l;
}
```

CONTROLLER

excluir(): fechando a classe (4/4)

controller/AlunoController.java — excluir()

```
01      public void excluir(int id) {
02          String sql = "DELETE FROM aluno WHERE id = ?";
03          try (Connection c = con(); PreparedStatement ps = c.prepareStatement(sql)) {
04              ps.setInt(1, id);
05              ps.executeUpdate();
06          } catch (SQLException e) { throw new RuntimeException(e); }
07      }
08  }
```

ARQUITETURA

Onde mora o acesso a dados?

Sem camada DAO, o **Controller acumula orquestração + JDBC**.
É um degrau didático: mantém a View limpa e centraliza tudo.



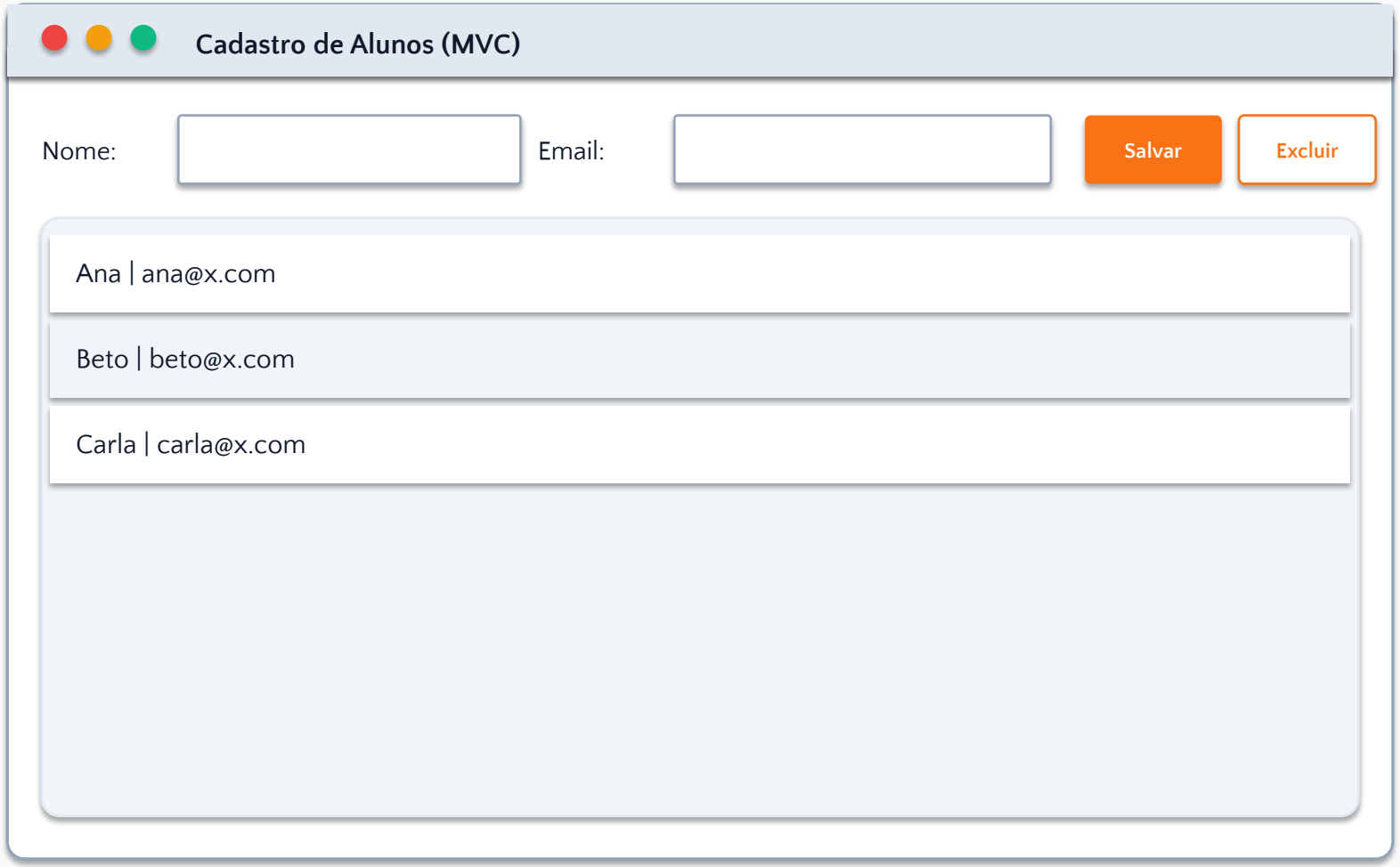
Por que aceitamos (por enquanto)?

Porque não há DAO nesta aula: alguém precisa falar com o banco, e o Controller é o único lugar coerente (a View jamais).

Na Aula 07 extraímos esse JDBC para um DAO, deixando o Controller fino.

RESULTADO

A aplicação MVC rodando



FLUXO

O caminho de um clique em Salvar

1 **View.salvar()**

lê txtNome/txtEmail

2 **Controller.salvar()**

monta o Aluno

3 **validar()**

aplica a regra de negócio

4 **JDBC INSERT**

grava no PostgreSQL

5 **Controller.listar()**

recarrega os dados

6 **View.listar()**

atualiza a JList

BOAS PRÁTICAS

MVC bem aplicado

Model = POJO puro

Só dados; zero SQL e zero regra na entidade.

View fina

Só exibe/captura; delega 100% ao Controller.

Controller orquestra

Valida e coordena; sem DAO, faz o JDBC.

Injeção do Controller

View recebe o Controller no construtor (testável).

Constantes de conexão

URL/USER/PASS como static final no Controller.

Rumo ao DAO

Na Aula 07, extraia o JDBC do Controller para um DAO.

ATENÇÃO

Erros comuns no MVC

SQL dentro da View
Acopla a tela ao banco e quebra o padrão.

Regra de negócio na View
Validação deve viver no Controller (ou Service).

Model com JDBC
A entidade POJO nunca abre conexão.

View criando entidades
Quem monta o Aluno é o Controller.

Controller chamando a View
O fluxo é View→Controller; o Controller só retorna dados.

Esquecer de atualizar a View
Depois de salvar/excluir, chame listar().

PRÁTICA

Exercícios: monte seu MVC

EXERCÍCIO 1 — Fácil

Campo novo

Adicione um campo 'telefone' no POJO, na View e no INSERT/SELECT do Controller, mantendo as camadas separadas.

EXERCÍCIO 2 — Médio

Validação no Controller

Crie validar() que exige email com '@' e nome com 3+ caracteres; a View apenas mostra um JOptionPane de erro.

EXERCÍCIO 3 — Desafio

Segunda entidade

Crie a entidade Curso (POJO) e um CursoController com seu próprio JDBC; liste cursos em um JComboBox na AlunoView.

Resumo da aula

MVC com Model-POJO

- Model = entidade (POJO), só dados
- View fina: exibe e captura
- Controller: regras + orquestração
- Sem DAO, o Controller faz o JDBC
- Fluxo View → Controller → banco

PRÓXIMA AULA

Padrão DAO

Vamos tirar o JDBC de dentro do Controller e isolá-lo em classes DAO, deixando o Controller fino e o projeto testável.

```
interface + Impl + Factory
```

Obrigado!

Aula 06 · Padrão MVC (Model–POJO · View · Controller)

Práticas de POO

Prof. Me. Célio Ricardo Castelano · Ciência da Computação