

Modelo, Visão e Controle – MVC

O padrão de arquitetura MVC aplicado a projetos Java

1. Introdução

O **MVC** (*Model-View-Controller*, ou Modelo-Visão-Controle) é um padrão de arquitetura de software que organiza uma aplicação em três camadas com responsabilidades bem definidas: o **Modelo**, responsável pelos dados e regras de negócio; a **Visão**, responsável pela apresentação das informações ao usuário; e o **Controlador**, responsável por receber as ações do usuário e coordenar a comunicação entre o Modelo e a Visão. Essa separação torna o código mais organizado, testável e fácil de manter.

2. Fluxo de Comunicação no MVC

De forma resumida, o usuário interage com a **Visão**; a Visão repassa essa ação ao **Controlador**; o Controlador consulta ou atualiza o **Modelo**; e, por fim, o Modelo notifica (direta ou indiretamente) a Visão para que ela seja atualizada com os novos dados.

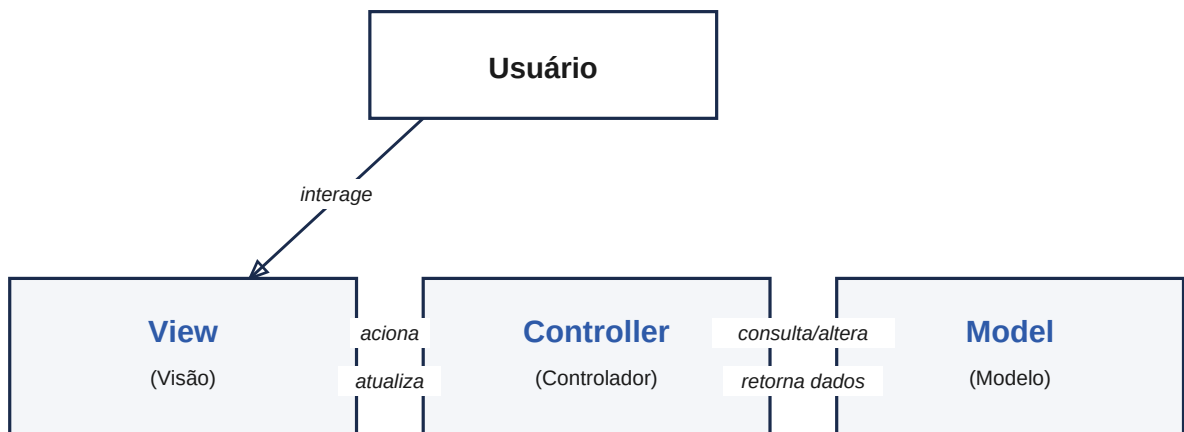


Figura 1 — Fluxo típico de comunicação entre Usuário, View, Controller e Model no padrão MVC.

3. Model (Modelo)

O **Model** representa os dados da aplicação e as regras de negócio associadas a eles. Ele não conhece a interface do usuário nem como os dados serão exibidos — sua única responsabilidade é manter o estado da aplicação de forma consistente.

```
public class Produto {
    private int id;
    private String nome;
    private double preco;

    public Produto(int id, String nome, double preco) {
        if (preco < 0) {
            throw new IllegalArgumentException("Preço não pode ser negativo.");
        }
        this.id = id;
        this.nome = nome;
        this.preco = preco;
    }

    public int getId() { return id; }
    public String getNome() { return nome; }
    public double getPreco() { return preco; }
    public void setPreco(double preco) { this.preco = preco; }
}
```

4. View (Visão)

A **View** é responsável apenas por **exibir** informações ao usuário e capturar suas ações (como cliques ou entradas de texto), sem conter nenhuma regra de negócio. Em uma aplicação de console, a View pode ser uma classe dedicada a imprimir e ler dados do terminal.

```

import java.util.List;
import java.util.Scanner;

public class ProdutoView {
    private Scanner scanner = new Scanner(System.in);

    public void exibirMenu() {
        System.out.println("1 - Cadastrar produto");
        System.out.println("2 - Listar produtos");
        System.out.println("3 - Sair");
    }

    public int lerOpcao() {
        return Integer.parseInt(scanner.nextLine());
    }

    public String lerNome() {
        System.out.print("Nome do produto: ");
        return scanner.nextLine();
    }

    public double lerPreco() {
        System.out.print("Preço do produto: ");
        return Double.parseDouble(scanner.nextLine());
    }

    public void exibirProdutos(List<Produto> produtos) {
        for (Produto p : produtos) {
            String linha = p.getId() + " - " + p.getNome() + " (R$ " + p.getPreco() + ")";
            System.out.println(linha);
        }
    }

    public void exibirMensagem(String mensagem) {
        System.out.println(mensagem);
    }
}

```

5. Controller (Controlador)

O **Controller** recebe as ações vindas da View, decide o que fazer com base nelas — geralmente chamando métodos do Model — e determina como a View deve ser atualizada em seguida. É o controlador que conecta as duas outras camadas, sem que a View precise conhecer diretamente o Model, nem vice-versa.

```

import java.util.ArrayList;
import java.util.List;

public class ProdutoController {
    private List<Produto> produtos = new ArrayList<>();
    private ProdutoView view;
    private int proximoId = 1;

    public ProdutoController(ProdutoView view) {
        this.view = view;
    }

    public void iniciar() {
        int opcao;
        do {
            view.exibirMenu();
            opcao = view.lerOpcao();

            switch (opcao) {
                case 1 -> cadastrarProduto();
                case 2 -> listarProdutos();
                case 3 -> view.exibirMensagem("Encerrando...");
                default -> view.exibirMensagem("Opção inválida.");
            }
        } while (opcao != 3);
    }

    private void cadastrarProduto() {
        String nome = view.lerNome();
        double preco = view.lerPreco();

        Produto produto = new Produto(proximoId++, nome, preco);
        produtos.add(produto); // interação com o Model

        view.exibirMensagem("Produto cadastrado com sucesso!");
    }

    private void listarProdutos() {
        view.exibirProdutos(produtos); // atualiza a View com dados do Model
    }
}

```

Por fim, a classe principal apenas cria a View e o Controller e inicia a aplicação:

```

public class Main {
    public static void main(String[] args) {
        ProdutoView view = new ProdutoView();
        ProdutoController controller = new ProdutoController(view);
        controller.iniciar();
    }
}

```

6. Vantagens do Padrão MVC

- **Separação de responsabilidades:** cada camada tem uma função clara, facilitando a compreensão do sistema.
- **Facilidade de manutenção:** alterar a aparência da aplicação (View) não exige mexer nas regras de negócio (Model), e vice-versa.

- **Reutilização:** um mesmo Model pode ser utilizado por diferentes Views (por exemplo, uma interface de console e uma interface gráfica).
- **Testabilidade:** o Model e o Controller podem ser testados isoladamente, sem depender da interface gráfica.

7. MVC em Frameworks Java

O exemplo apresentado usa uma implementação manual e simplificada do MVC, útil para compreender o padrão. Na prática, frameworks Java já oferecem uma estrutura MVC pronta: o **Spring MVC**, muito usado em aplicações web, organiza os Controllers como classes anotadas com `@Controller`, as Views geralmente como templates HTML (Thymeleaf, JSP) e o Model como objetos de domínio ou DTOs. Já em aplicações desktop, o **JavaFX** também segue uma estrutura próxima ao MVC, separando arquivos FXML (View), classes de Controller e classes de Model.

8. Conclusão

O padrão MVC organiza uma aplicação em camadas com responsabilidades bem delimitadas, tornando o código mais fácil de entender, testar e evoluir ao longo do tempo. Como visto no exemplo do cadastro de produtos, mesmo em uma aplicação simples de console é possível aplicar os princípios do MVC, separando claramente os dados (Produto), a apresentação (ProdutoView) e o fluxo de controle (ProdutoController) — uma prática que se estende naturalmente para aplicações web e desktop construídas com frameworks Java mais robustos.