2025-2 / Lorenzon : OO - Design Patterns - MVC

Design Patterns – MVC

O padrão MVC (**Model–View–Controller**) é um padrão **arquitetural** (não está no catálogo GoF, mas é um clássico de design de software) usado para separar responsabilidades em aplicações.

Ele divide o sistema em três partes principais:

- **Model (Modelo):** representa os dados e a lógica de negócio;
- **View (Visão):** exibe a interface ao usuário e apresenta os dados do Model. Quando falamos interface, não estamos falando de "interface" par abstração de contratos em Java, por exemplo;
- **Controller (Controlador):** recebe as ações do usuário, coordena a comunicação entre View e Model e decide o fluxo da aplicação.

A ideia central do MVC é permitir que a interface de usuário seja separada da lógica de negócio, o que torna o código mais organizado, manutenível e testável.

É utilizado quando se deseja organizar o sistema em camadas bem definidas. Em aplicações com interface de usuário complexa, para não misturar código de lógica com apresentação. Desta forma facilita manutenção e testes, isolando responsabilidades (SOLID, SRP).

É importante observar que ele permite que a mesma lógica de negócio (Model) pode ser usada por diferentes tipos de interface (Views diferentes).

Benefícios

- **Separação de responsabilidades:** cada camada tem uma função clara;
- **Reuso:** o mesmo Model pode ser utilizado com múltiplas Views;
- **Flexibilidade:** a View pode mudar sem alterar a lógica de negócio;
- **Testabilidade:** torna mais simples escrever testes unitários para as regras de negócio.

Evolução

O MVC clássico nasceu no Smalltalk nos anos 70 e foi amplamente usado em desktops (Swing, JavaFX, .NET) e em frameworks Web (Spring MVC, Django, Rails, ASP.NET MVC). Hoje, o MVC ainda é usado, mas em muitos casos foi adaptado:

- **MVP (Model–View–Presenter):** comum em Android clássico;
- **MVVM (Model–View–ViewModel):** usado em Angular, Vue, React, WPF.

Em arquiteturas modernas (Clean Architecture, Hexagonal, DDD), o princípio de separar responsabilidades continua, mas de forma mais robusta. No frontend moderno (React, Vue, Angular), embora não se fale explicitamente em MVC, a separação de responsabilidades se inspira nele.

O MVC nasceu antes do DDD, SOLID e Clean Architecture. Estas práticas trouxeram uma visão mais ampla e robusta para estruturar sistemas grandes. Introduziram camadas como Repositórios, Serviços, Use Cases, que de certa forma

cumprem papéis semelhantes ao ViewModel no MVVM, ou seja, desacoplar a lógica da apresentação e deixar o código mais testável.

O MVVM já antecipava práticas de baixo acoplamento e responsabilidade única, princípios que hoje chamamos de SOLID. Repositórios e Serviços que você tem usado funcionam como parte dessa evolução:

- O **Repository** separa o acesso a dados (persistência).
- O **Service**/Use Case concentra a regra de negócio.
- Isso reflete o mesmo espírito do **ViewModel**, separar lógica da interface, só que de forma mais genérica e aplicável a qualquer domínio.

Em DDD, o foco é no domínio e suas regras, mas a ideia de "cada camada tem uma responsabilidade" é a mesma que motivou o MVC/MVVM.

Foi tudo meio que absorvido por práticas como DDD, SOLID, Clean Architecture. Já atendia os princípios, mas ganhou outros nomes e estruturas para projetos mais complexos. Quando trabalhamos hoje com camadas de repositórios, serviços e controladores, estamos aplicando na prática uma evolução do que era o MVC/MVVM: *"O que antes chamávamos de MVC/MVVM hoje aparece em forma de camadas mais organizadas, mas a essência é a mesma: separar responsabilidades para evitar acoplamento."*

Para exemplificar uma ideia de MVC uma aplicação console, observe estes códigos a seguir. Perceba a **injeção** (composição) das classes e não utilização de herança. Tudo é baseado em **contratos**, não em classes concretas. Isso dá maior flexibilidade: você pode trocar a View para uma versão gráfica ou web, sem mexer no Controller. Também facilita mockar [1]os componentes em testes unitários (ex.: criar uma FakeAlunoView). Se conecta diretamente com princípios SOLID (principalmente D – Dependency Inversion Principle).

Interfaces

O modelo, que separa os dados:

```
public interface EstudanteModel {

    String getNome();

    int getIdade();

    void setNome(String nome);

}
```

[1] Mockar vem de mock, que em inglês significa "simular/fingir". No contexto de programação e testes, mockar significa criar um objeto falso que simula o comportamento de uma classe ou interface real. Esse objeto não executa a lógica real, mas apenas responde de forma controlada para que possamos testar outra parte do sistema isoladamente. Por que usar mocks?

- Isolamento: testar uma classe sem depender de banco de dados, rede, APIs externas etc.
- Velocidade: testes ficam mais rápidos, porque não chamam recursos reais;
- Controle: podemos programar o mock para devolver respostas específicas (ex.: simular erro de rede);
- Precisão: conseguimos verificar se um método foi chamado (ex.: se o Controller chamou a View corretamente).

A visão, que se preocupa com a exibição:

```
public interface EstudadeView {
    void exibirDetalhesAluno(EstudanteModel estudante);
}
```

O controlador, que agrupa as ações do conjunto:

```
public interface EstudanteControllerInterface {
    void atualizarNome(String nome);
    //void exibirAluno(); Isso é coisa pra view
}
```

A classe de entidade então, pode ser Aluno, Professor, qualquer coisa do contexto, desde que implemente o contrato, exemplo de Aluno:

```
public class Aluno implements EstudanteModel {
    private String nome;
    private int idade;
    public Aluno(String nome, int idade) {
        setNome(nome);
        this.idade = idade;
    }
    @Override
    public String getNome() {
        return nome;
    }
    @Override
    public int getIdade() {
        return idade;
    }
    @Override
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

Veja agora, por exemplo, ao invés de criarmos o toString() no Aluno, reescrevendo-o na classe, esta ação é da visão:

```
public class AlunoConsoleView implements EstudadeView {
    @Override
    public void exibirDetalhesAluno(EstudanteModel estudade) {
        System.out.println("Estudante: " + estudante.getNome() + ", Idade: " +
                           estudante.getIdade());

    }
}
```

No controle, trata-se as injeções e as ações sobre o contrato. OU seja, a programação é para a implementação.

```
public class EstudanteController implements EstudanteControllerInterface {
    private EstudanteModel model;
    private EstudanteView view;

    public EstudanteController(EstudanteModel model, EstudanteView view) {
        this.model = model;
        this.view = view;
    }

    public void atualizarNome(String nome) {
        model.setNome (nome);
    }

    public void exibirEstudante() {
        view.exibirDetalhesAluno(model);
    }
}
```

E, por fim, nossa aplicação:

```
public class App {

    public static void main(String[] args) {

        EstudanteModel model = new Aluno("Maria", 20);

        EstudanteView view = new AlunoConsoleView();

        EstudanteControllerInterface controller = new

                              EstudanteController(model, view);


        view.exibirDetalhesAluno (model);    // Maria, 20

        controller.atualizarNome("João");

        view.exibirDetalhesAluno(model);    // João, 20

    }

}
```

Nesta organização a orientação a objetos é mais fiel: a View lida diretamente com o objeto, não com dados soltos. Há extensibilidade, se o Model crescer (novos atributos), a assinatura dos métodos não precisa mudar. Reduzimos a duplicação passando para o Controller o objeto inteiro, evitando passar cada campo manualmente. Nesta realidade, estamos mais próximo do que frameworks reais fazem (ex.: Spring MVC entrega o objeto inteiro para a View).

No exemplo, se o Controller dependesse diretamente de uma classe concreta, teríamos forte acoplamento. Mas como ele recebe um contrato (EstudanteView), podemos injetar qualquer implementação:

- Uma EstudanteConsoleView que imprime no console;
- Uma EstudanteSwingView que abre uma janela gráfica;
- Uma EstudanteWebView que renderiza HTML;
- Ou até uma MockEstudanteView para testes unitários.

Assim, estamos praticando programar para interface, não para implementação, porque o Controller não depende de uma classe fixa, mas de um contrato que pode ser cumprido de várias formas.

“No caso do MVC, programar para a **interface** permite que o Controller não dependa de uma implementação fixa da View. Isso garante baixo acoplamento, facilidade de manutenção e possibilidade de trocar a interface (console, GUI, web) sem alterar a lógica do Controller. Se programássemos para a implementação, ficaríamos presos a uma classe específica, reduzindo a flexibilidade e violando princípios como o Open/Closed e a Inversão de Dependência do SOLID.”