

# CLEAN CODE COM **SPRING** **BOOT**

Fundamentos e Primeiros  
Passos em Projetos Java  
com Spring

Géssica Mendonça

# Introdução: Por que Clean Code importa?

Imagine abrir um projeto após seis meses e não entender nada do que você mesma escreveu.

Métodos com nomes genéricos, classes gigantescas, códigos repetidos, exceções genéricas, e um Service1 que não faz ideia do que está servindo.

Isso não é raro, é o padrão em muitos projetos que crescem rápido e sem cuidado.

Clean Code é mais do que escrever “código bonito”. É escrever código legível, previsível e fácil de manter. Especialmente em projetos Java com Spring Boot, onde a facilidade de começar pode esconder armadilhas arquiteturais no médio e longo prazo.

Este eBook é um guia prático e direto. Em vez de teoria extensa, você encontrará aqui:

- Exemplos reais de código com o que fazer e o que evitar
- Dicas que você pode aplicar hoje no seu projeto
- Princípios como SRP, boas práticas de nomenclatura e organização
- Um checklist final para manter seu código limpo todos os dias

Nos próximos capítulos, você vai aprender como transformar seu projeto em um legado que outros desenvolvedores (e você no futuro) terão prazer em dar manutenção.

Código limpo não é só sobre funcionar é sobre continuar funcionando sem sofrimento.

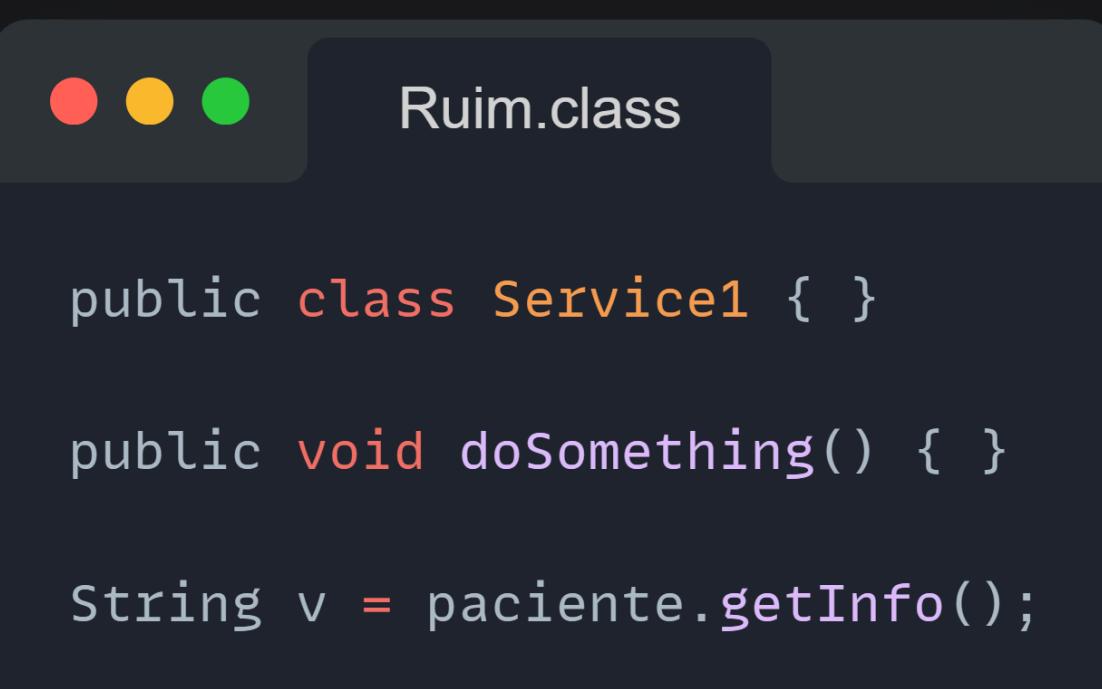
# Capítulo 1: Nomeando como um Jedi

Nomes são a primeira coisa que lemos no código. Um bom nome explica por si só o que aquela classe, método ou variável faz. Um nome ruim, por outro lado, obriga o leitor a adivinhar ou seguir chamadas para entender o que está acontecendo.

Código limpo começa com nomes limpos.

Evite nomes genéricos

Um erro comum é usar nomes curtos ou genéricos demais:



The image shows a dark-themed code editor window. In the top bar, there are three colored circular icons (red, yellow, green) followed by the text "Ruim.class". The main code area contains the following Java code:

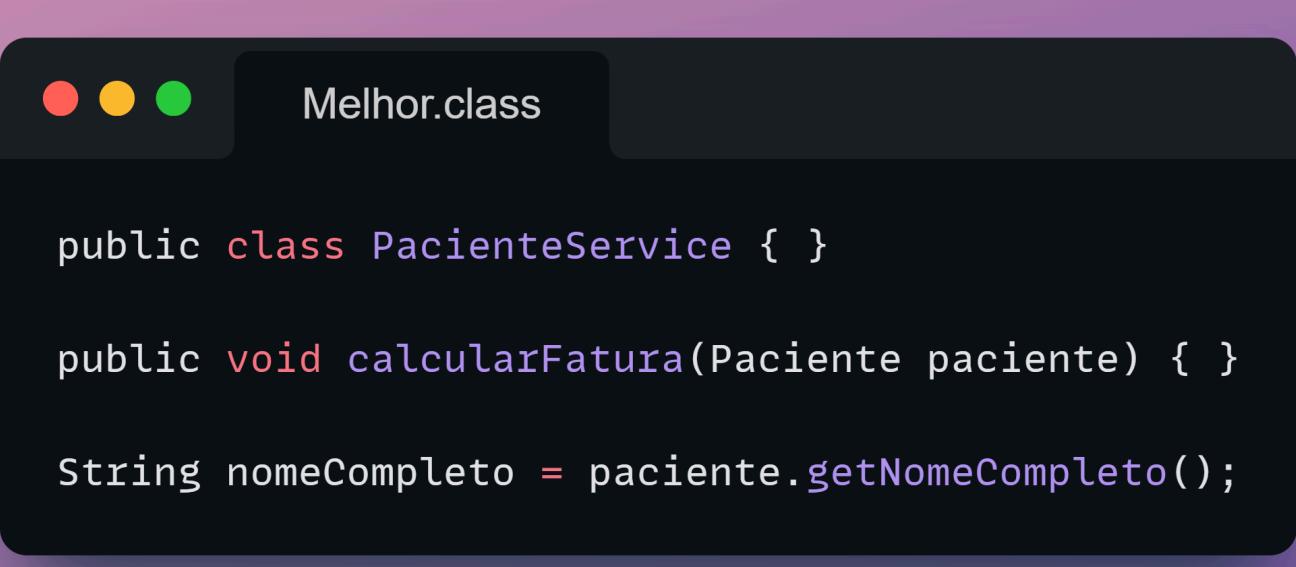
```
public class Service1 { }

public void doSomething() { }

String v = paciente.getInfo();
```

Esses nomes não dizem nada. O dev precisa abrir a classe ou o método para entender o que está acontecendo.

Agora compare com:



```
public class PacienteService { }

public void calcularFatura(Paciente paciente) { }

String nomeCompleto = paciente.getNomeCompleto();
```

Cada nome comunica uma intenção clara.  
Quem lê não precisa adivinhar.

## Dê contexto aos nomes

Evite variáveis soltas como data, status ou valor. Pergunte: "data de quê?", "status de quem?", "valor de onde?".



Exemplo.class

```
// Ruim  
String status = "ATIVO";  
  
// Melhor  
String statusDoPlano = "ATIVO";
```

# Não repita nomes desnecessariamente

Quando o contexto já está claro, você não precisa repetir:



Exemplo.class

```
// Evite
public class Paciente {
    private String nomeDoPaciente;
    private String cpfDoPaciente;
}
```

```
// Melhor
public class Paciente {
    private String nome;
    private String cpf;
}
```

# Nomeie métodos como ações

Métodos devem ser verbos ou expressar claramente uma ação ou pergunta.



Exemplo.class

```
// Ruim  
pacienteService.validar();  
  
// Melhor  
pacienteService.validarCadastro(paciente);
```

Evite nomes vagos como executar, processar, handle. Substitua por ações específicas do domínio:



Exemplo.class

```
// Melhor  
pacienteService.realizarTriagem(paciente);  
relatorioService.gerarFaturamentoMensal();
```

# Classes como substantivos

Classes representam "coisas", então seus nomes devem ser substantivos. Evite misturar com verbos.



Exemplo.class

```
// Correto
class Paciente { }
class Agendamento { }
class TriagemService { }
```

# Capítulo 2: Organização de pacotes - Estrutura por domínio, não por camada

A forma como você organiza os pacotes do seu projeto diz muito sobre sua arquitetura. Muitos projetos Java com Spring Boot seguem a estrutura tradicional por camadas técnicas:

```
src/
└── main/
    └── java/
        └── com/
            └── sistema/
                ├── controller/
                ├── service/
                └── repository/
```

Embora funcione, essa estrutura dificulta a manutenção à medida que o projeto cresce. As classes ficam espalhadas, você precisa pular entre vários pacotes para entender uma única funcionalidade, e o acoplamento entre domínios fica invisível.

O problema da separação por tipo  
Imagine que você precise fazer uma  
mudança na lógica de agendamento de  
consultas.

Com a estrutura tradicional, você teria que  
editar arquivos em:

- controller/ConsultaController.java
- service/ConsultaService.java
- repository/ConsultaRepository.java
- 

Ou seja, o código relacionado à mesma  
funcionalidade está fragmentado.

## Estrutura por domínio

A alternativa mais recomendada é organizar por módulo de negócio, também chamado de estrutura por feature ou por contexto. Cada pacote agrupa tudo que pertence a um domínio específico:

```
src/
└── main/
    └── java/
        └── com/
            └── sistema/
                ├── consultas/
                │   ├── ConsultaController.java
                │   ├── ConsultaService.java
                │   └── ConsultaRepository.java
                └── pacientes/
                    ├── PacienteController.java
                    ├── PacienteService.java
                    └── PacienteRepository.java
```

Dessa forma, tudo o que pertence a “consultas” fica no pacote consultas.

Isso:

- Facilita a leitura
- Reduz o acoplamento entre áreas diferentes
- Permite reusar ou separar domínios facilmente

# Exemplo prático em Spring Boot

```
// src/main/java/com/sistema/pacientes/PacienteController.java

@RestController
@RequestMapping("/pacientes")
public class PacienteController {

    private final PacienteService pacienteService;

    public PacienteController(PacienteService pacienteService) {
        this.pacienteService = pacienteService;
    }

    @GetMapping
    public List<Paciente> listar() {
        return pacienteService.listarTodos();
    }
}
```

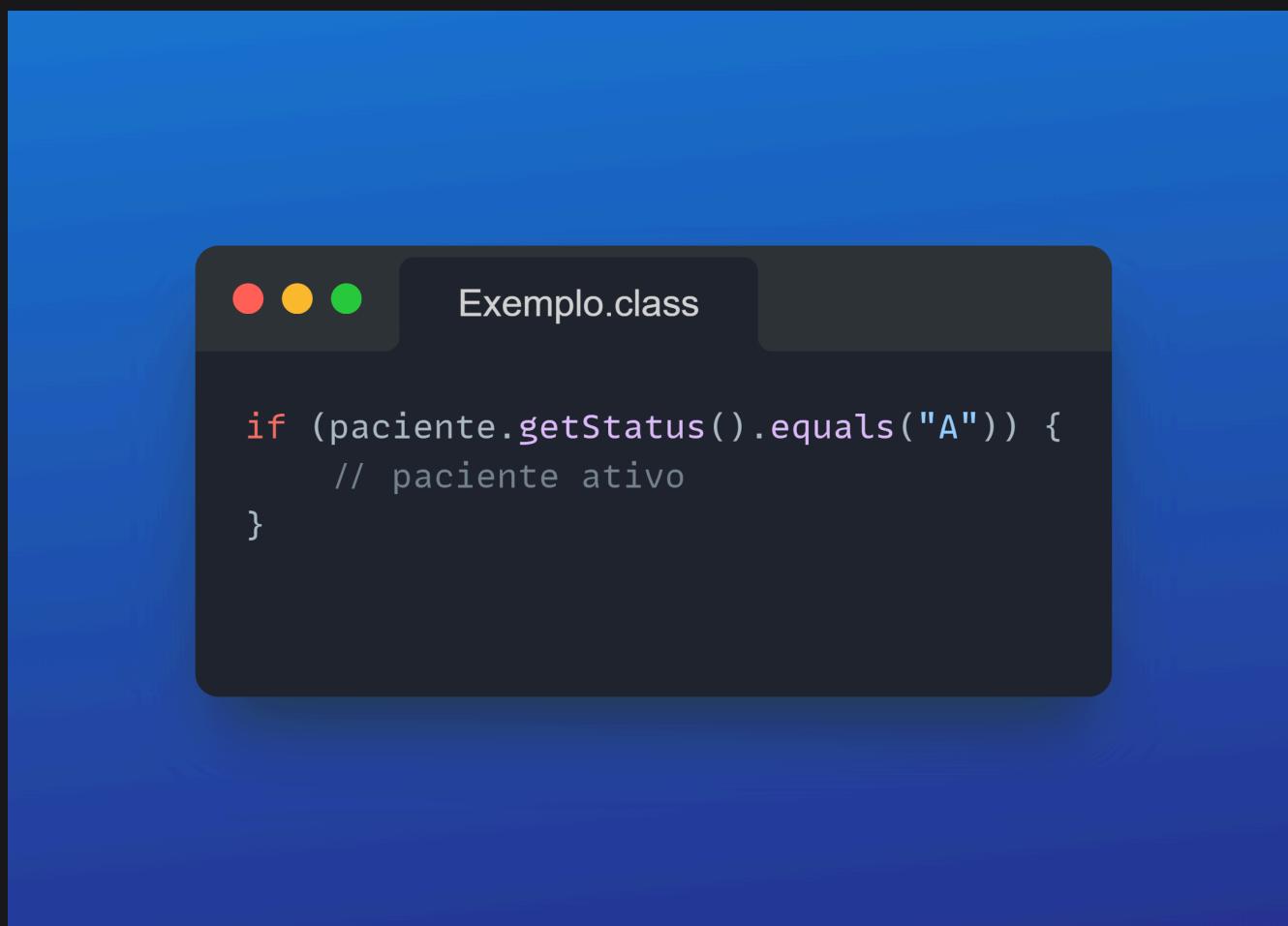
Todos os arquivos relacionados ao domínio pacientes estão no mesmo lugar.

Isso ajuda tanto na manutenção quanto na escalabilidade do projeto.

# Capítulo 3: Evite códigos mágicos - use constantes e enums com intenção

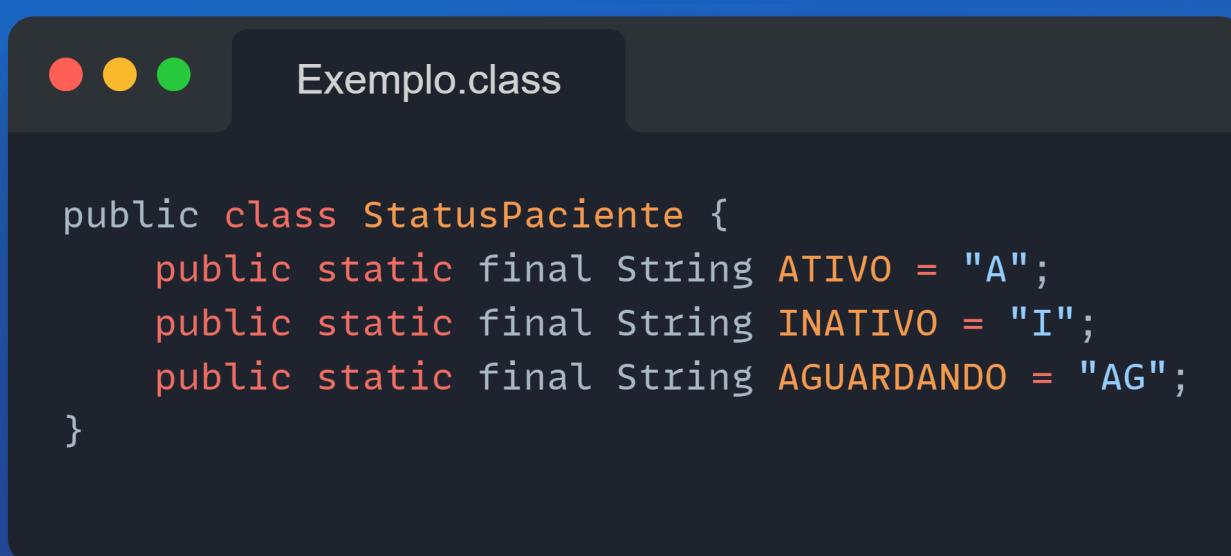
Código mágico é qualquer valor fixo (como números ou strings) que aparece “do nada” no código.

Ele até funciona, mas ninguém sabe de onde veio ou por que existe. Isso dificulta a leitura e aumenta a chance de erro.



O que é "A"? Ativo? Aguardando? Alta?  
Qualquer desenvolvedor que leia esse código  
vai precisar adivinhar - ou sair procurando  
onde isso foi definido.

## Como resolver: use constantes



```
Exemplo.class

public class StatusPaciente {
    public static final String ATIVO = "A";
    public static final String INATIVO = "I";
    public static final String AGUARDANDO = "AG";
}
```



Exemplo.class

```
if (paciente.getStatus().equals(StatusPaciente.ATIVO)) {  
    // agora está claro  
}  
  
public enum StatusPaciente {  
    ATIVO,  
    INATIVO,  
    AGUARDANDO  
}
```

## Quando usar constantes e quando usar enums

- Use constantes para valores simples, especialmente se ainda dependem de bancos legados (como códigos "A", "I", etc.)

- Use enums quando puder tratar os valores como um tipo completo no seu domínio.

Evitar códigos mágicos é uma das mudanças mais simples e poderosas para deixar seu código mais limpo, confiável e fácil de dar manutenção.

# Capítulo 4: SRP - Cada classe com sua responsabilidade

- SRP (Single Responsibility Principle) é o “S” do SOLID.
- 
- Ele diz que uma classe deve ter uma e apenas uma razão para mudar. Ou seja, cada classe deve fazer uma coisa só.
- Na prática, isso significa evitar classes que acumulam múltiplas funções: validação, lógica de negócio, chamadas a banco, envio de e-mail, geração de PDF, tudo no mesmo lugar.

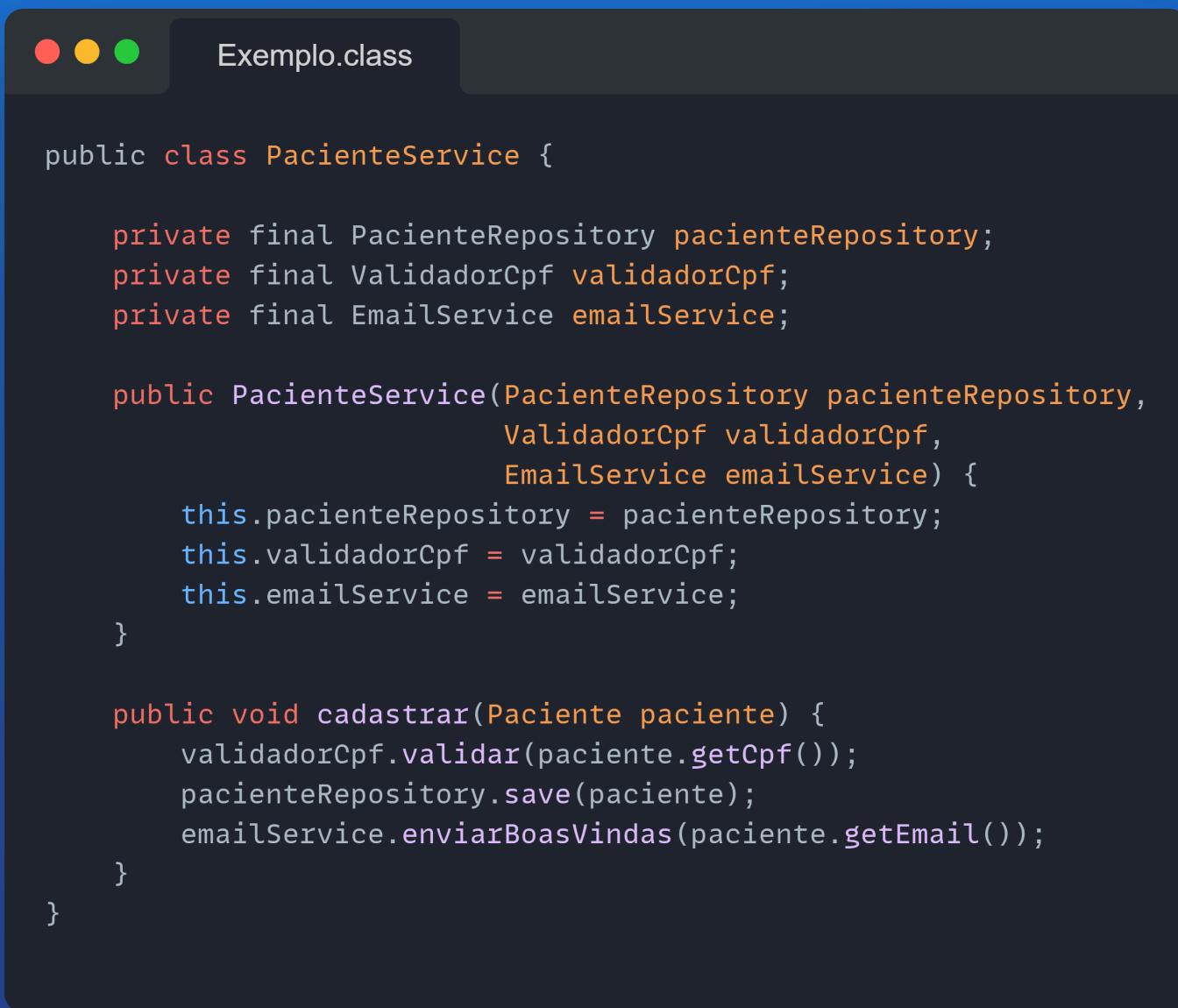
# Exemplo de violação de SRP



The screenshot shows a code editor window titled "Exemplo.class". The code is as follows:

```
public class PacienteService {  
  
    public void cadastrar(Paciente paciente) {  
        validarCpf(paciente.getCpf());  
        pacienteRepository.save(paciente);  
        enviarEmailBoasVindas(paciente.getEmail());  
    }  
  
    private void validarCpf(String cpf) {  
        // lógica de validação  
    }  
  
    private void enviarEmailBoasVindas(String email) {  
        // lógica de envio de e-mail  
    }  
}
```

# Refatorando para respeitar SRP



```
public class PacienteService {

    private final PacienteRepository pacienteRepository;
    private final ValidadorCpf validadorCpf;
    private final EmailService emailService;

    public PacienteService(PacienteRepository pacienteRepository,
                          ValidadorCpf validadorCpf,
                          EmailService emailService) {
        this.pacienteRepository = pacienteRepository;
        this.validadorCpf = validadorCpf;
        this.emailService = emailService;
    }

    public void cadastrar(Paciente paciente) {
        validadorCpf.validar(paciente.getCpf());
        pacienteRepository.save(paciente);
        emailService.enviarBoasVindas(paciente.getEmail());
    }
}
```

Agora temos:

- ValidatorCpf: lida com regras de CPF
- PacienteRepository: lida com o banco
- EmailService: lida com o envio de e-mails

Cada classe tem sua própria razão de existir e pode evoluir de forma isolada.

Benefícios do SRP

- Menos acoplamento
- Classes menores e mais testáveis
- Alterações mais seguras e localizadas
- Melhor reutilização de código

Você não precisa aplicar SRP de forma obsessiva, mas sempre que sentir que sua classe está “inchada” ou que seus métodos estão fazendo demais, é um bom momento para aplicar esse princípio.

# Conclusão

Mandou bem em chegar até aqui! Você já pegou o básico do Clean Code em Java com Spring: aquelas boas práticas que deixam seu código mais claro e fácil de manter e evitou uns erros que podem dar dor de cabeça.

Claro, Clean Code é uma estrada longa, cheia de aprendizados e ajustes.

O importante é ir melhorando um pouco todo dia.

Use o que viu aqui pra deixar seus próximos códigos mais elegantes e sua vida mais tranquila.

Curtiu? Então vamos colocar tudo em prática e continuar evoluindo. Código sujo nunca mais!