Código Limpo: Princípios e Práticas

Boas Práticas de Programação - BPP 2025.2

Prof. Fernando Marques Filho

29 de agosto de 2025

Universidade Federal do Rio Grande do Norte

Agenda

Introdução ao Código Limpo

Nomes Significativos

Funções

Comentários

Conclusão

Introdução ao Código Limpo

O que é Código Limpo?

Código Limpo
É um código
fácil de entender e **fácil de alterar**

"Qualquer tolo consegue escrever código que um computador entende.
Bons programadores escrevem código que humanos podem entender."

— Martin Fowler

Origem e Motivação

- **Criado por:** Robert C. Martin (Uncle Bob)
- **Motivação:** Combater o débito técnico
- **Objetivo:** Software sustentável e de qualidade

Débito Técnico

Custo implícito de uma implementação pensada apenas no agora, em vez de usar uma abordagem de melhor qualidade.

Nomes Significativos

Nomes Significativos

✗ Ruim

```
// Nomes que não revelam propósito
int d; // dias decorridos
String nm = "João";
List<String> lst = new ArrayList<>();

// Difícil de entender o contexto
double x = calculateSalary(h, r);
```

✓ Bom

```
// Nomes que revelam propósito claramente
int daysSinceLastLogin;
String customerName = "João";
List<String> approvedUsers = new
    ArrayList<>();

// Fica claro o propósito de cada
    variável
// Fácil de entender e manter
double monthlySalary =
    calculateMonthlySalary(hoursWorked,
    hourlyRate);
```

Regra de Ouro

O nome deve responder: **Por que existe? O que faz? Como é usado?**

Nomes Pronunciáveis e Buscáveis

✗ Evite

```
// Nomes difíceis de pronunciar e buscar
String xlzqp = "data";
int grmblwskx = 42;
List<User> usrLstMngr = new ArrayList<>()
    ;

class XyzManager {
    // Difícil de encontrar no código
    // Difícil de pronunciar
    void prcssXyzData() { }
}
```

✓ Prefira

```
// Nomes pronunciáveis e buscáveis
String applicationData = "data";
int maxRetryAttempts = 42;
List<User> userAccountManager = new
    ArrayList<>();

class AccountManager {
    // Fácil de encontrar com Ctrl+F
    // Fácil de pronunciar e discutir
    sobre
    void processAccountData() { }
}
```

- **Pronunciáveis:** Facilita comunicação entre equipe
- **Buscáveis:** Permite encontrar rapidamente no código
- **Sem prefixos:** Evite notações húngaras (strNome, intIdade)

Classes e Métodos: Nomenclatura

✗ Classes Genéricas

```
// Nomes genéricos que não revelam
    propósito
class Manager { }
class Data { }
class Info { }
class Processor { }
class Handler { }
```

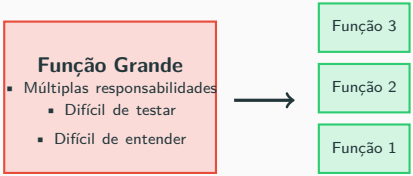
✓ Classes Específicas

```
// Nomes específicos que revelam
    propósito
class UserAccountManager { }
class CustomerData { }
class OrderInformation { }
class PaymentProcessor { }
class EmailNotificationHandler { }
```

- **Classes:** Substantivos ou frases nominais
- **Métodos:** Verbos ou frases verbais
- **Evite:** Manager, Processor, Data, Info

Funções

Funções: Pequenas e Focadas



Regra

Uma função deve fazer uma coisa, fazê-la bem e fazer apenas ela.

Exemplo: Refatoração de Função

✗ Função com Múltiplas Responsabilidades

```
# Função com múltiplas responsabilidades
def processar_tarefas(tarefas):
    for tarefa in tarefas:
        if tarefa.status == "pendente":
            print("Descrição:", tarefa.descricao)
            enviar_email(tarefa.usuario.email)
```

✓ Funções Especializadas

```
# Funções especializadas e focadas

def filtrar_tarefas_pendentes(tarefas):
    return [t for t in tarefas if t.status == "pendente"]

def notificar_usuario(tarefa):
    enviar_email(tarefa.usuario.email)

def exibir_tarefa(tarefa):
```

Argumentos de Função

✔ Objeto de Parâmetro

```
// Objeto de parâmetro - mais claro e
    flexível
public class DadosUsuario {
    public String nome;
    public String email;
    public String telefone;
    public String endereco;
    public int idade;
    public boolean ativo;
    public String departamento;
}

```

✖ Muitos Parâmetros

```
// Muitos parâmetros - difícil de usar
public void criarUsuario(String nome,
    String email,
                         String telefone,
    String endereco,
                         int idade,
    boolean ativo,
                         String
```

Efeitos Colaterais em Funções

Problema
Funções que fazem mais do que prometem em seu nome causam efeitos colaterais inesperados.

✗ Com Efeito Colateral

```
1 // Função com efeito colateral inesperado
2 public boolean validarSenha(String senha)
      {
3     if (senha.length() > 8) {
4         // Efeito colateral: inicia
      sessão
5         iniciarSessao();
6         return true;
7     }
8     return false;
9 }
```

✓ Sem Efeito Colateral

```
1  // Separação clara de responsabilidades
2  public boolean validarSenha(String senha)
       {
3      return senha.length() > 8;
4  }
5
6  public void fazerLogin(String senha) {
7      if (validarSenha(senha)) {
8          iniciarSessao();
9      }
10 }
```

Comentários

Comentários: Quando e Como

> *"A necessidade de comentários muitas vezes indica*
> *que o código não está claro o suficiente"*
> **— Uncle Bob**

✔ Bons Comentários

- Explicação de intenções
- Esclarecimentos
- Avisos de consequências
- TODOs

✘ Maus Comentários

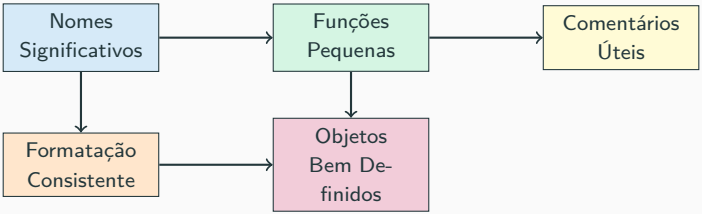
- Murmúrios
- Redundantes
- Enganosos
- Código comentado

ℹ Regra

Escreva código autoexplicativo primeiro. Use comentários apenas quando necessário.

Conclusão

Resumo dos Princípios



Lembre-se

Código limpo não é escrito de uma vez. É refinado continuamente.

Próximos Passos

1. **Pratique:** Aplique esses princípios no seu código diário
2. **Refatore:** Melhore código existente gradualmente
3. **Code Review:** Use esses critérios para avaliar código
4. **Ferramentas:** Utilize analisadores estáticos (próxima aula)

Dúvidas?

Referências

- Martin, Robert C. **Clean Code: A Handbook of Agile Software Craftsmanship**. Prentice Hall, 2008.
- Fowler, Martin. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley, 2019.
- Catálogo de Code Smells: `https://luzkan.github.io/smells/`
- Repositório do curso: `https://github.com/fmarquesfilho/bpp-2025-2`

Obrigado pela atenção!

`fernando@dimap.ufrn.br`