2025-2 / Lorenzon : OO – Clean Code

Clean Code (Código Limpo)

Definição: Clean Code é um conjunto de princípios e valores que orientam a escrita de código claro, legível e de fácil manutenção. Foi popularizado por **Robert C. Martin (Uncle Bob)** em seu livro *Clean Code: A Handbook of Agile Software Craftsmanship* (2008).

Origem e História: Surge como parte do movimento de **Software Craftsmanship**, que defende a ideia de programadores como artesãos do código, preocupados não apenas em "fazer funcionar", mas em **fazer bem feito**.

Objetivo: Escrever código que qualquer desenvolvedor — inclusive você no futuro — consiga **ler, entender e evoluir** sem esforço, reduzindo riscos de erros e custos de manutenção.

Características de um código limpo

- **Legível:** fácil de entender sem precisar de explicações adicionais;
- **Manutenível:** pode ser modificado ou ampliado sem quebrar outras partes do sistema;
- **Simples e direto:** evita soluções complexas quando uma abordagem clara resolve o problema.

Metáfora didática: Um código limpo é como um bom texto: frases bem estruturadas, vocabulário claro e ausência de ruídos. Assim como um leitor compreende o texto com facilidade, qualquer programador deve compreender o código.

Benefícios:

- Comunicação clara entre desenvolvedores;
- Redução de erros e ambiguidades;
- Maior produtividade a longo prazo;
- Sistemas mais fáceis de manter e evoluir.

Desvantagens:

- Requer disciplina e tempo inicial para aplicar boas práticas;
- Pode parecer "mais trabalhoso" no começo;
- Exige aprendizado e adaptação da equipe.

Relações com padrões

- **Design Patterns**: oferecem soluções recorrentes. O Clean Code foca em **escrever o código que implementa esses padrões de forma clara**;

- **DDD**: traz a **linguagem ubíqua**; Clean Code reforça que nomes de classes e métodos devem refletir o domínio, não jargões técnicos obscuros;

- **Object Calisthenics**: são quase "exercícios de Clean Code" para treinar disciplina;

- **SOLID**: princípios de design que sustentam código limpo e sustentável.

Princípios práticos de Clean Code para complementar

1. **Nomes significativos**
    - calcularTotalPedido() é melhor que calcTot();
    - Relaciona-se com *Object Calisthenics – Não abrevie.*

2. **Funções pequenas**
    - Cada função deve fazer só uma coisa;
    - Cada classe deve ter uma responsabilidade;
    - Alinhado ao *SRP (Single Responsibility Principle).*

3. **Evite comentários desnecessários**
    - Código deve se explicar pelos nomes e pela clareza;
    - Reduz poluição no código. Pilar do Manifesto Ágil;
    - Comentário só para explicar “por que”, nunca “o que”.

4. **Formatação clara**
    - Indentação correta;
    - Object Calisthenics para indentação de um nível, menos elses;

5. **Evitar duplicação de código (DRY – Don’t Repeat Yourself)**
    - Código repetido = chance duplicada de erro;
    - Se algo repete é motivo pra classe ou função;
    - Se algo reusa é motivo pra generalização.

6. **Tratamento de erros com clareza**
    - Usar exceções no lugar de códigos mágicos (-1, null);
        - Throw new, throws;
    - Sempre trate erros de forma explícita.
    - Fail fast e guard Clauses;
    - Earley return, happy path (Object Calisthenics);

7. **Código expressivo**
    - Ao ler, deve parecer quase uma narrativa em linguagem natural (exemplo de DDD).

8. **Testabilidade**
    - Código limpo é código fácil de testar.
    - Funções pequenas e responsabilidades únicas ajudam a escrever testes automáticos.

“Maus Cheiros” a evitar

- Métodos longos demais.
- Classes “Deus” (fazem de tudo).
- Nomes curtos e sem significado (x1, data2).
- Comentários que tentam explicar código confuso.
- Duplicação de lógica.

Por que é importante Clean Code?

- **Tempo**: boa parte da vida de um dev é **ler código existente**, não escrever novo;
- **Equipe**: o código é uma linguagem de comunicação entre desenvolvedores;
- **Qualidade**: sistemas mal escritos se tornam difíceis de manter, acumulando o famoso *código legado*.

Código sujo x limpo

```
public class Pedido {
    public int st; // 0 = aberto, 1 = pago, 2 = cancelado
    public double v;

    public Pedido(int s, double val) {
        this.st = s;
        this.v = val;
    }

    public void proc() {
        if (st == 1) {
            if (v > 0) {
                System.out.println("OK");
            } else {
                System.out.println("Valor inválido");
            }
        } else {
            System.out.println("Pedido não processado");
        }
    }
}
```

Problemas:

Variáveis com nomes ruins (st, v, proc).

Números mágicos (0, 1, 2 para status).

Método com múltiplas responsabilidades.

Mensagens pouco claras.

Estrutura de controle confusa.

Melhorias a aplicar:

Nomes significativos (status, valor, processar).

Enum substitui números mágicos (mais expressivo e seguro).

Método extraído para clareza (podeSerProcessado).

Fail Fast: se não pode ser processado, retorna cedo.

Mensagens claras para o usuário/leitor.

```
public class Pedido {
    private StatusPedido status;
    private double valor;

    public Pedido(StatusPedido status, double valor) {
        this.status = status;
        this.valor = valor;
    }

    public void processar() {
        if (!podeSerProcessado()) {
            System.out.println("Pedido não pode ser processado.");
            return;
        }
        System.out.println("Processando pedido de valor R$ " + valor);
    }

    private boolean podeSerProcessado() {
        return status.equals(StatusPedido.PAGO) && valor > 0;
    }
}

enum StatusPedido {
    ABERTO, PAGO, CANCELADO
}
```