



Qualidade de software com Clean Code e técnicas de usabilidade



Clean Code: a Filosofia do “Código Limpo”

Contexto histórico; código limpo; técnicas de
Clean Code.

Bloco 1

Stella Marys Dornelas Lamounier

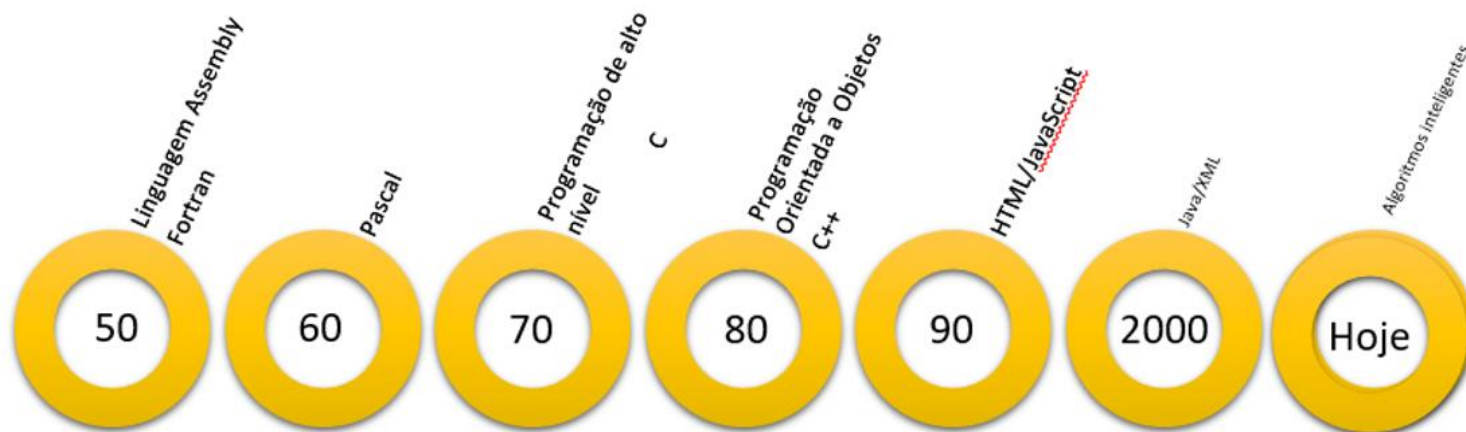


➤ História por trás do código-fonte



➤ História por trás do código-fonte

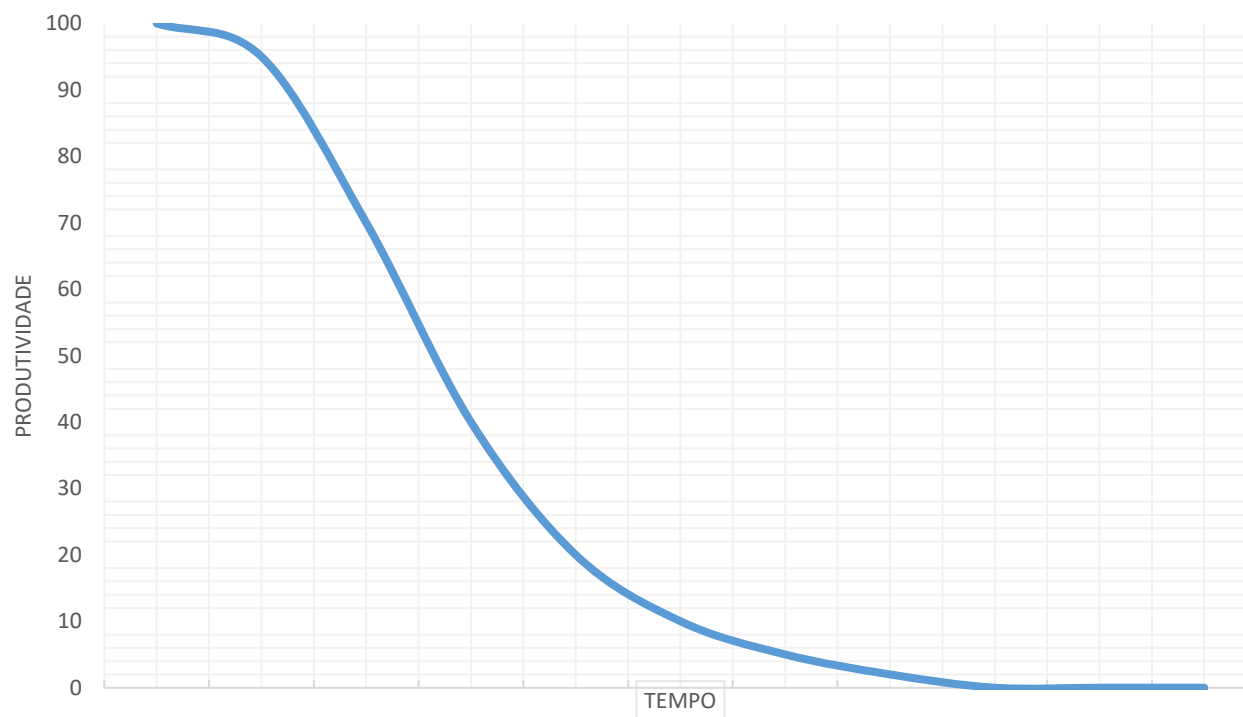
Figura 1 – Linha do tempo – Linguagens de Programação



Fonte: elaborada pela autora.

➤ O código-fonte

Figura 2 – Produtividade *versus* Tempo de desenvolvimento



Fonte: adaptada de Martins (2019, p. 4).



➤ O código-fonte





O código-fonte limpo

- Fácil entendimento.
- Claro.
- Legível.
- Fácil manutenção.
- Livre de *bugs*.
- Simples e direto.



➤ O código-fonte limpo

Figura 3 – Código-fonte com variáveis com nomes não relevantes

```
var n = "Stella";  
var a = 2020;  
var sn = "Dornelas";  
var aux = 1;  
  
console.log(p + " " + js + " está lecionando "  
+ (x + aux));  
// Resultado: Stella Dornelas está lecionando em 2021
```

Fonte: elaborada pela autora.

Figura 4 – Código-fonte com variáveis com nomes relevantes

```
var nome = "Stella";  
var anoAtual = 2020;  
var sobrenome = "Dornelas";  
  
console.log(nome + " " + sobrenome + " está lecionando "  
+ (anoAtual + 1));  
// Resultado: Stella Dornelas está lecionando em 2021
```

Fonte: elaborada pela autora.



O código-fonte limpo

Figura 5 – Código-fonte com variáveis com nomes não relevantes

```
def caminhoR(Vertice v):  
    Vertice w  
    est[v] = 0  
    for(w = 0; w < tamanho(); w++)  
        if(adj(v,w))  
            if(est[w] == -1):  
  
                imprime(w)  
                caminhoR(w)
```

Fonte: Almeida e Miranda (2010, p. 9).

Figura 6 – Código-fonte com variáveis com nomes relevantes

```
def imprimeVerticesDoPasseioComOrigemEm(Vertice origem):  
    Vertice proximo;  
    estado[origem] = JA_VISITADO  
    for(proximo = 0; proximo < numeroDeVertices(); proximo++)  
        if(saoAdjacentes?(origem, proximo))  
            if(estado[proximo] == NAO_VISITADO):  
                imprime(proximo)  
                imprimeVerticesDoPasseioComOrigemEm(proximo)
```

Fonte: Almeida e Miranda (2010, p. 10).

► Variáveis

- Devem ter nomes representativos.
- Devem estar localizadas perto de onde serão utilizadas.
- Evite criá-las uma embaixo da outra.

```
// evite: tht = 0;  
// utilize: totalDeHorasTrabalhadas = 0;  
  
// evite: salario = 3400;  
// utilize: salarioEmReais = 3400;
```

► Importe-se com o seu código-fonte

- Dê nomes claros e objetivos.
- Crie funções específicas e pequenas.
- Não seja redundante.
- Aplique testes.
- Refatore.
- Leia e entenda o que escreveu.

“Um código limpo sempre parece que foi escrito por alguém que se importava”

(Michael C. Feathers)



➤ Evite!

- Nomes muito parecidos e grandes.
- Nomes difíceis de pronunciar.
- Prefixos e caracteres especiais.
- Excesso de comentários.
- Códigos comentados/palavras reservadas.
- Não quebre indentação.

“Um código limpo sempre parece que foi escrito por alguém que se importava”.



Clean Code: a Filosofia do “Código Limpo”

Técnicas de *Clean Code*

Bloco 2

Stella Marys Dornelas Lamounier



➤ Classes

- São representadas por substantivos (Cliente, Aluno, Estoque).
- Devem ser pequenas e com poucas variáveis.
- Devem descrever sua responsabilidade, seguindo a ordem:
 - Públicas (raramente).
 - Estáticas.
 - Constantes.
 - Estáticas privadas.
 - Instância privada.

// Evite: Class DadosAluno.

// Utilize: Class Aluno.

Classes

```
// Evite
public class OrdemPagamento
{
    public void Pagamento(CreditoCartao cartao)
    {
        if(cartao == null)
            // Pagamento via boleto
            // Pagamento via cartão } }
```

```
// Utilize
public class OrdemPagamento {
    public void Pagamento ()      {
        // Pagamento via boleto
    }
    public void Pagamento (CreditoCartao cartao){
        // Pagamento via cartão de crédito
    }
}
```



Funções e métodos

- Menos é sempre mais.
- Nomes significativos.
- Métodos devem ter apenas um *assert* e um conceito.
- Funções exemplo.

// Evite: CHT ().

// CalculeHorasTrabalhadas ().





Funções e métodos

Figura 7 – Código-fonte com variáveis com nomes não relevantes

```
public class AddressTest extends TestCase {  
  
    private Address anAddress;  
  
    protected void setUp() throws Exception {  
        anAddress = new Address("ADDR1$ADDR2$CITY IL 60563$COUNTRY");  
    }  
  
    public void testAddress() throws Exception {  
        assertEquals("ADDR1", anAddress.getAddress());  
    }  
  
    public void testCity() throws Exception {  
        assertEquals("CITY IL 60563", anAddress.getCity());  
    }  
  
    public void testCountry() throws Exception {  
        assertEquals("COUNTRY", anAddress.getCountry());  
    }  
}
```

Fonte: Hora (2019, p. 36).

➤ Indentação

Figura 8 – Código sem indentação

```
public class GerarLista {  
public static void main(String[] args) {  
int i =0;  
int[] lista = new int[5];  
for (int numero : lista){  
lista[i] = i;  
i++;}  
for (int numero : lista){  
System.out.println("Lista: " + numero);}  
}  
}
```

Fonte: Jesus (2021, [s.p.]).

Figura 9 – Código com indentação

```
public class GerarLista {  
    public static void main(String[] args) {  
        int i =0;  
        int[] lista = new int[5];  
        for (int numero : lista){  
            lista[i] = i;  
            i++;}  
        for (int numero : lista){  
            System.out.println("Lista: " + numero);  
        }  
    }  
}
```

Fonte: Jesus (2021, [s.p.]).



Refatoração

Figura 10 – Código sem refatoração

```
public double CalcularDesconto(double valor)
{
    if(Desconto())
    {
        return valor * 0.5;
    }
    return valor;
}
private bool Desconto()
{
    return produto && cliente && semDivida;
}
```

Fonte: Jesus (2021, [s.p.]).

Figura 11 – Código com refatoração

'Código refatorado

```
public double CalcularDesconto(double valor)
{
    if(produto && cliente && semDivida)
    {
        return valor * 0.5;
    }
    return valor;
}
```

Fonte: Jesus (2021, [s.p.]).



Comentários

- O código deve ser autoexplicativo.
- Evite comentários sem contexto.
- Evite comentários longos e com palavras reservadas.
- Códigos sofrem alteração ao longo do tempo, observe os comentários.

```
// Evite  
// Função principal do software - SystemCad  
public void Main() { ... }
```

```
// Evite  
public void CadastrarAluno()  
{  
  // string nome= "Joao";  
  // public void Metodo() {... } }
```

➤ Comentários

- Devem demonstrar sua intenção.

```
// Utilize  
// Retorna a lista de alunos faltosos  
// para o relatório de frequência  
public ListaAluno<Faltas> ObtemAlunosFaltosos() { ... }
```

```
// Utilize public void CancelarCompra()  
{  
// Caso o pedido já tenha sido enviado  
// ele não pode mais ser cancelado.  
if(DataEnvio > DateTime.Now)  
    {  
        AddNotification("Compra realizada com sucesso,  
impossível cancelamento");  
    }  
}
```

Clean Code: a Filosofia do “Código Limpo”

TDD; técnica F.I.R.S.T.

Bloco 3

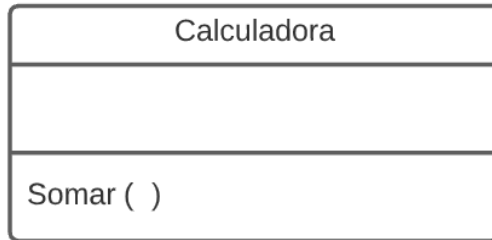
Stella Marys Dornelas Lamounier



➤ Teste Unitário

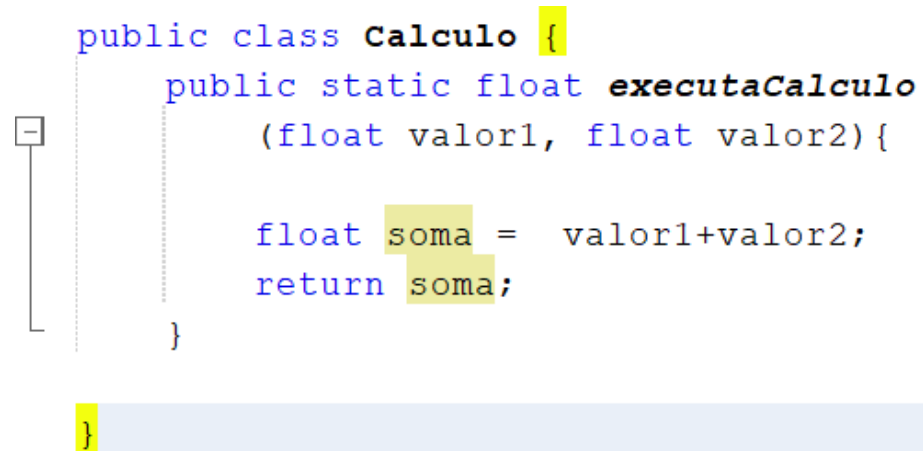
- Responsável por testar as unidades dos sistemas isoladamente.

Figura 12 – Classe Calculadora



Fonte: elaborada pela autora.

Figura 13 – Código-fonte em Java – Classe Calculadora



```
public class Calculo {
    public static float executaCalculo
        (float valor1, float valor2){

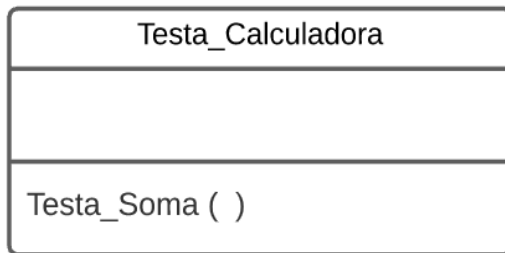
        float soma = valor1+valor2;
        return soma;
    }
}
```

A Java source code snippet for the 'Calculo' class. The code is enclosed in a light blue rectangular box. To the left of the code, there is a small icon of a document with a minus sign, and a vertical line connects it to the start of the code block. The code defines a public class 'Calculo' with a public static method 'executaCalculo' that takes two float parameters and returns a float value representing the sum of the two parameters.

Fonte: elaborada pela autora.

➤ Teste Unitário

**Figura 14 – Classe
Testa_Calculadora**



Fonte: elaborada pela autora.

**Figura 15 – Código-fonte em Java –
Testa_Calculadora**

```
@Test
public void testExecutaCalculo() {
    // System.out.println("executaCalculo");
    float valor1 = 6;
    float valor2 = 3;
    float resultado_esperado = 9;
    float resultado_obtido = Calculo.executaCalculo
(valor1, valor2);
    assertEquals(resultado_esperado,
                resultado_obtido, 0);
}
```

Fonte: elaborada pela autora.



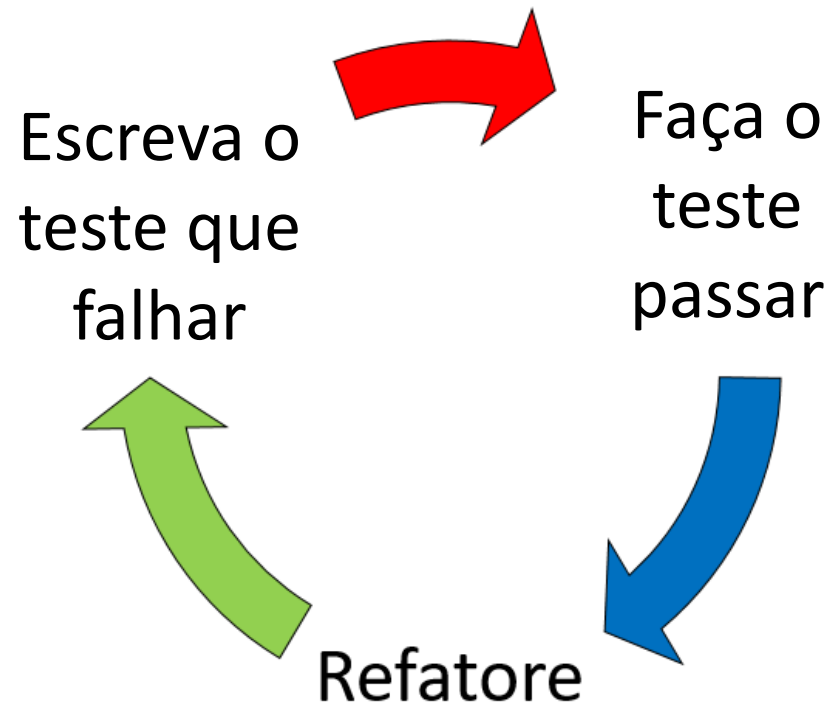
Testes em *Clean Code*

- Código de teste é tão importante quanto de produção.
- Devem ser flexíveis para melhorar a performance.
- Devem ser rápidos.
- Definem o comportamento.
- Legibilidade.
- Simplicidade.



➤ Técnica Orientada a Testes (TDD)

Figura 16 – Ciclo do TDD



Fonte: adaptada de Pinheiro (2015, p. 21).

► Lei do TDD

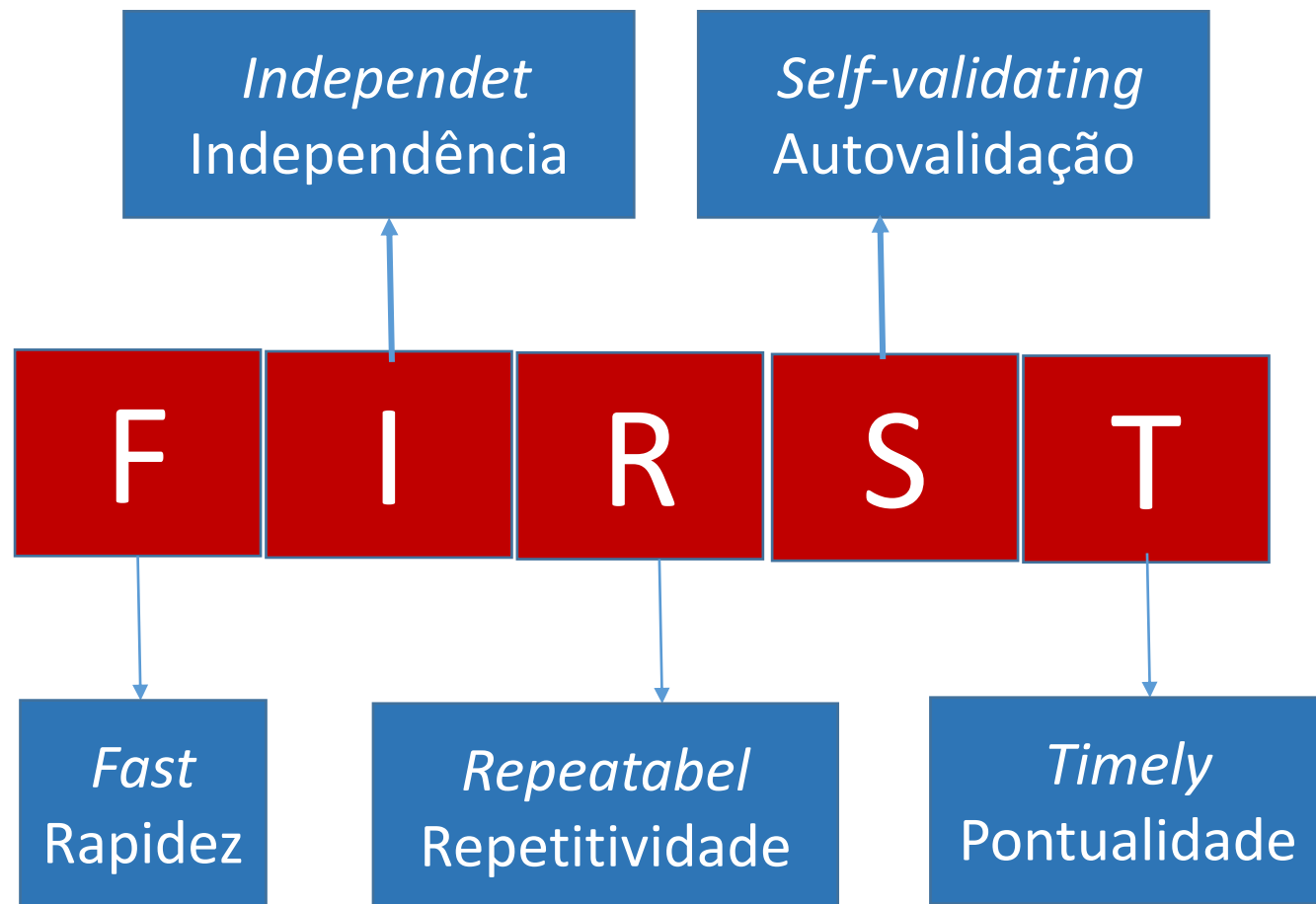
- O código-fonte só deverá ser escrito após a falha do teste unitário.
- Número limitado de testes, ou seja, não é permitido escrever mais testes do que o suficiente para falhar (UM).
- Não se deve escrever mais códigos do que o suficiente para aplicar o teste de falha atual.

(MARTIN, 2019, p. 123)



Como manter os testes limpos?

Figura 17 – Método FIRST



Fonte: Martin (2019, p. 132).

Testes Unitários – *Clean Code*

Figura 18 – Código de testes complexo

```
public void testGetPageHieratchyAsXml() throws Exception
{
    crawler.addPage(root, PathParser.parse("PageOne"));
    crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
    crawler.addPage(root, PathParser.parse("PageTwo"));

    request.setResource("root");
    request.addInput("type", "pages");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
}
```



Fonte: Jesus (2021, [s.p.]).

➤ Testes Unitários – *Clean Code*



Figura 19 – Código de teste limpo

```
public void testGetPageHierarchyAsXml() throws Exception {  
    makePages("PageOne", "PageOne.ChildOne", "PageTwo");  
  
    submitRequest("root", "type:pages");  
  
    assertResponseIsXML();  
    assertResponseContains(  
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"  
    );  
}
```

Fonte: Jesus (2021, [s.p.]).

Teoria em Prática

Bloco 4

Stella Marys Dornelas Lamounier



➤ Reflita sobre a seguinte situação

Reflita sobre a frase:

Sistemas ruins podem funcionar mesmo que elaborados sem nenhuma boa prática na criação de código.

Tente contrariar essa afirmação e convencer uma empresa de desenvolvimento de software a utilizar boas práticas que podem deixar um código-fonte limpo e de fácil entendimento para quem o opera. Quais técnicas poderiam ser utilizadas?



➤ Norte para a resolução

- Nos dias de hoje, os testes de software tanto manuais como automatizados fazem parte do cotidiano das empresas de desenvolvimento, mas, muito mais do que testar, é necessário escrever de maneira limpa os códigos-fonte dos softwares a serem desenvolvidos e os códigos de testes a serem executados, a fim de minimizar erros e garantir a qualidade do sistema a ser criado.





Norte para a resolução

Há inúmeras boas práticas para tornar os códigos limpos, de boa escrita e legibilidade, como:

- Técnica FIRST.
- Utilizar boas práticas para a programação:
 - Nomes com a expressão do que realmente são capazes, que levam seu propósito.
 - Variáveis curtas e padronizadas derivadas de substantivos.
 - Funções derivadas de verbos simples, claras e objetivas, com poucos ou nenhum argumento.
 - Métodos sempre em forma de verbos e com nomes curtos e padronizados.
 - Uso de comentários apenas quando o próprio código não for capaz de expressar o que realmente faz.



Dica do(a) Professor(a)

Bloco 5

Stella Marys Dornelas Lamounier



➤ Dicas

- Aprender a criar códigos limpos é sem dúvida uma tarefa complexa e árdua para a maioria dos profissionais. Requer conhecimento de padrões, boas práticas e muita dedicação.
- Lembre-se da frase: “Um código ruim pode muito bem funcionar, mas, se ele não for limpo, pode acabar com uma empresa de desenvolvimento” (MARTIN, 2019, p. 9).
- Um bom livro poderá auxiliar profissionais a melhorarem o entendimento de código com *Clean Code*.





Dica de livro

Código Limpo – Habilidades Práticas do Agile Software

Autor: Robert C. Martin.

Editora: Alta Books.

Páginas: 456.

Ano: 2020.





Referências

ALMEIDA, L. T.; MIRANDA, J. M. **Código Limpo e seu Mapeamento para Métricas de Código Fonte**. 2010. Disponível em: <https://www.ime.usp.br/~cef/mac499-10/monografias/lucianna-joao/arquivos/monografia.pdf>. Acesso em: 29 out. 2021.

FEATHERS, M. C. **Trabalho Eficaz Com Código Legado**. Porto Alegre: Bookman, 2013.

HORA, A. **Engenharia de Software II: Código Limpo – Parte 3: Código externo e testes de unidade**. 2019. Disponível em: <https://homepages.dcc.ufmg.br/~andrehora/teaching/es2/10-codigo-limpo-parte-3.pdf>. Acesso em: 29 out. 2021.

MARTIN, R. C. **Código limpo: habilidades práticas do Agile software**. Rio de Janeiro: Alta Books, 2019.

PINHEIRO, P. H. R. **Testes na linguagem GO: Como fazer TDD em GOLANG!** 2015. Disponível em: <https://paulohrpinheiro.xyz/texts/go/2018-09-24-testes-na-linguagem-go.html>. Acesso em: 29 out. 2021.

JESUS, L. S. **Clean Code (código limpo). Teste a Velocidade**, 2021. Disponível em: <https://testeavelocidade.com.br/codigo-limpo/>. Acesso em: 10 nov. 2021.

Bons estudos!

