Boas Práticas de Programação e código Limpo

Profº Bruno Baruffi Esteves

Breve Histórico profissional

Profº Bruno Baruffi Esteves

Formado em Sistemas de Informação 15 anos de experiência na área de desenvolvimento.

Atuando em projetos de pequeno a grande porte com foco em tecnologias de ponta, códigos de qualidade e alta maturidade.

Boas Práticas e Código Limpo

É igual, pero no mucho

Boas Práticas

Equivalente a “Regras de Etiqueta“ dentro da programação, ficando em entregar um código bom, funcional e bem escrito.

Código Limpo

Em poucas Palavras, é um código fácil de ler, entender, modificar e testar, tanto para quem desenvolveu como para outros Devs.

Custo do código ruim - Perda de Tempo e Produtividade

Caça aos bugs: Programadores passam horas (ou dias!) decifrando código confuso para encontrar a causa de um único erro.

Dificuldade de implementação: Adicionar novas funcionalidades se torna um pesadelo, cheio de efeitos colaterais inesperados.

Onboarding lento: Novos membros da equipe levam muito mais tempo para entender e contribuir com um código mal escrito.

Números: Programadores gastam, em média, 50% do seu tempo lidando com bugs. Boa parte disso se deve à dificuldade com códigos mal escritos. (Fonte: "Empirical Evaluation of Software Development Tools", University of Cambridge, 2014)

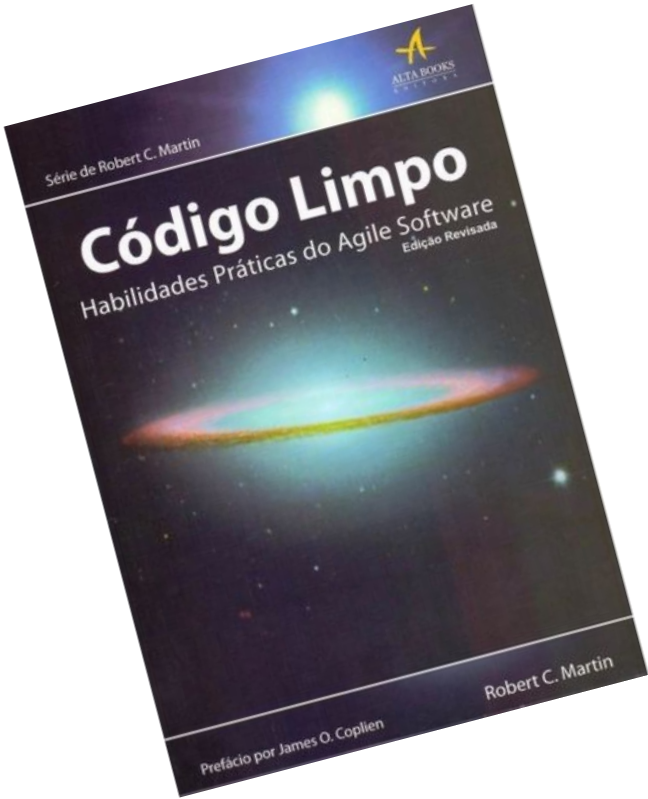
Pilares do código limpo

Legibilidade: Código fácil de entender por qualquer programador, como um livro bem escrito.

Simplicidade : Buscar a solução mais simples e direta, evitando complexidade desnecessária.

Manutenibilidade: Código fácil de modificar e atualizar sem gerar novos problemas ou efeitos colaterais.

Testabilidade: Código escrito de forma que facilite a criação e execução de testes automatizados.



Código limpo - Legibilidade

```java
public class Exemplo {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;
        int c = a + b;
        System.out.println("Resultado: " + c);
    }
}
```

```java
public class Exemplo {
    public static void main(String[] args) {
        int valorInicial = 10;
        int valorASomar = 5;
        int resultadoDaSoma = valorInicial + valorASomar;
        System.out.println("Resultado da Soma: " + resultadoDaSoma);
    }
}
```

Código limpo - Simplicidade

```java
public class VerificadorDeIntervalo {
    public boolean estaNoIntervalo(int numero) {
        if (numero > 10) {
            if (numero < 20) {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }
}
```

```java
public class VerificadorDeIntervalo {

    public boolean estaNoIntervalo(int numero) {
        return numero > 10 && numero < 20;
    }
}
```

Código limpo - Manutenibilidade Ruim

```java
public class Pedido {
    // ... outros atributos ...
    private List<Produto> produtos;

    public double calcularValorTotal() {
        double total = 0;
        for (Produto produto : produtos) {
            total += produto.getPreco() * produto.getQuantidade();
            // Se for um produto eletrônico, adicionar taxa de entrega ad
            if (produto.getCategoria().equals("Eletronicos")) {
                total += 10.0;
            }
            // Se o pedido for maior que R$ 200, aplicar desconto de 10%
            if (total > 200.0) {
                total *= 0.9;
            }
        }
        return total;
    }
}
```

Código limpo - Manutenibilidade Bom

```java
public class Pedido {
    // ... outros atributos ...
    private List<Produto> produtos;

    public double calcularValorTotal() {
        double total = calcularValorBruto();
        total += calcularTaxaEntrega();
        total -= calcularDesconto();
        return total;
    }

    private double calcularValorBruto() {
        double total = 0;
        for (Produto produto : produtos) {
            total += produto.getPreco() * produto.getQuantidade();
        }
        return total;
    }

    private double calcularTaxaEntrega() {
        double taxaEntrega = 0;
        for (Produto produto : produtos) {
            if (produto.getCategoria().equals("Eletronicos")) {
                taxaEntrega += 10.0;
            }
        }
        return taxaEntrega;
    }

    private double calcularDesconto() {
        double valorBruto = calcularValorBruto();
        if (valorBruto > 200.0) {
            return valorBruto * 0.1;
        }
        return 0;
    }
}
```

Código limpo - Testabilidade - Ruim

```
public class UsuarioService {

    public void cadastrarUsuario(String nome, String email) {
        // ... lógica para salvar o usuário no banco de dados ...

        // Envia o email de boas-vindas
        EmailSender emailSender = new EmailSender();
        emailSender.enviar(email, "Bem-vindo!", "Olá " + nome + ", seja bem-vindo
à nossa plataforma!");
    }
}
```

Código limpo - Testabilidade - Bom

```java
public class UsuarioService {

    private EmailSender emailSender;

    // Injeção de dependência via construtor
    public UsuarioService(EmailSender emailSender) {
        this.emailSender = emailSender;
    }

    public void cadastrarUsuario(String nome, String email) {
        // ... lógica para salvar o usuário no banco de dados ...

        // Envia o email de boas-vindas
        emailSender.enviar(email, "Bem-vindo!", "Olá " + nome + ", seja bem-vindo
à nossa plataforma!");
    }
}
```

Código limpo
Testabilidade
Teste

```java
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;

public class UsuarioServiceTest {

    @Test
    public void testCadastrarUsuario() {
        // Cria um mock do EmailSender
        EmailSender emailSenderMock = mock(EmailSender.class);

        // Cria uma instância de UsuarioService com o mock
        UsuarioService usuarioService = new UsuarioService(emailSenderMock);

        // Chama o método que queremos testar
        usuarioService.cadastrarUsuario("João", "joao@email.com");

        // Verifica se o método enviar do mock foi chamado com os parâmetros
corretos
        verify(emailSenderMock, times(1)).enviar("joao@email.com", "Bem-vindo!",
anyString());
    }
}
```

Cuidados a Tomar

Nomenclatura: Nomes claros e descritivos que revelam a intenção do código.
Exemplos: Utilizar nomes descritivos para variáveis, funções, classes, etc. Ex: calcularTotal() no lugar de calc().
Convenções: Seguir as convenções da linguagem (camelCase, snake_case) para padronizar o código.

```
int a = 10;

String b = "texto";

boolean c = false;
```

```
int idadeUsuario = 10;

String nomeProduto = "texto";

boolean emailConfirmado = false;
```

Cuidados a Tomar

Comentários: Explicar o "porquê" do código, não o "o quê". Documentar decisões complexas.
Evitar: Comentários óbvios que apenas repetem o código. Código autoexplicativo sempre.

```
// Soma dois números
int resultado = numero1 + numero2; // Guarda o resultado
```

```
// Calcula o valor total do pedido com base nos itens e descontos.
double valorTotal = calcularValorTotalPedido(itens, descontos);
```

Cuidados a Tomar

Funções e Métodos: Funções curtas e concisas, com um único objetivo.
Coesão: Agrupar código relacionado dentro da mesma função.
Evitar: Evitar muitos parâmetros, ideal até 3. Caso necessite de mais, utilize um objeto.

```
public void processarDados(List<String> dados) {
    // ... 50 linhas de código misturando validações, formatações, cálculos e
persistência ...
}
```

```
public void processarDados(List<String> dados) {
    List<String> dadosValidados = validarDados(dados);
    List<String> dadosFormatados = formatarDados(dadosValidados);
    List<Double> resultadosCalculados = calcularResultados(dadosFormatados);
    persistirResultados(resultadosCalculados);
}

// ... outras funções menores para validação, formatação, cálculo e persistência
```

Cuidados a Tomar

Organização de Código:
Indentação: Utilizar indentação consistente para melhorar a legibilidade.
Espaçamento: Usar espaços em branco para separar blocos de código e operadores.
Estruturação: Organizar o código em pacotes, módulos e classes de forma lógica e coesa.

```
public class MyClass {
    private String nome;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

```
public class MyClass{private String nome;public String getNome(){
return nome;}public void setNome(String n){nome = n;}}
```

Cuidados a Tomar

Tratamento de Erros: Prever e tratar erros de forma eficiente para evitar falhas inesperadas.
Exceções: Utilizar exceções para lidar com situações excepcionais no código.
Mensagens claras: Fornecer mensagens de erro informativas que ajudem na depuração.

```
public void conectarAoBancoDeDados() {
    // ... código que pode gerar exceções
}
```

```
public void conectarAoBancoDeDados() {
    try {
        // ... código que pode gerar exceções ...
    } catch (SQLException ex) {
        // ... tratamento específico para erro de SQL ...
        log.error("Erro ao conectar ao banco de dados", ex);
    }
}
```

Cuidados a Tomar - SOLID

SOLID: É um conjunto de princípios para criar softwares mais robustos, manutenível e escaláveis.

Princípios:

Single Responsibility, **Open/Closed**, **Liskov Substitution**, **Interface Segregation**, **Dependency Inversion**

Cuidados a Tomar - SOLID

Single Responsibility: Cada classe deve ter apenas uma responsabilidade específica e bem definida, evitando que ela se torne complexa e difícil de manter.

Exemplo ruim:

```
class Livro {
    // ... atributos do livro ...

    public void cadastrarLivro(Livro livro) {
        // ... lógica para salvar o livro no banco de dados ...
    }

    public void gerarRelatorioLivrosPorCategoria(String categoria) {
        // ... lógica para gerar um relatório de livros ...
    }
}
```

Cuidados a Tomar - SOLID

Single Responsibility
Exemplo Bom:

```
class Livro {
    // ... atributos do livro ...
}

class LivroService {
    public void cadastrarLivro(Livro livro) {
        // ... lógica para salvar o livro no banco de dados ...
    }
}

class RelatorioService {
    public void gerarRelatorioLivrosPorCategoria(String categoria) {
        // ... lógica para gerar um relatório de livros ...
    }
}
```

Cuidados a Tomar - SOLID

Open/Closed: Devemos poder adicionar novas funcionalidades sem alterar o código existente, o que reduz o risco de introduzir novos bugs.

Exemplo Ruim:

```
class AreaCalculator {
    public double calcularArea(Object forma) {
        if (forma instanceof Retangulo) {
            // ... lógica para calcular a área do retângulo ..
        } else if (forma instanceof Circulo) {
            // ... lógica para calcular a área do círculo ...
        }
        // ... mais condicionais para outras formas ...
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        AreaCalculator calculadora = new AreaCalculator();

        Retangulo retangulo = new Retangulo(5, 10);
        System.out.println("Área: " + calculadora.calcularArea(retangulo));

        Circulo circulo = new Circulo(7);
        System.out.println("Área: " + calculadora.calcularArea(circulo));
    }
}
class Circulo implements Forma {
    private double raio;

    public Circulo(double raio) {
        this.raio = raio;
    }

    @Override
    public double calcularArea() {
        return Math.PI * raio * raio;
    }
}

class AreaCalculator {
    public double calcularArea(Forma forma) {
        return forma.calcularArea();
    }
}
```

```java
interface Forma {
    double calcularArea();
}

class Retangulo implements Forma {
    private double base;
    private double altura;

    public Retangulo(double base, double altura) {
        this.base = base;
        this.altura = altura;
    }

    @Override
    public double calcularArea() {
        return base * altura;
    }
}
```

Open/Closed
Exemplo Bom:

Cuidados a Tomar - SOLID

Liskov Substitution: As subclasses devem poder ser usadas no lugar de suas classes pai sem causar problemas ou comportamentos inesperados.

```
class Passaro {
    public void voar() {
        // ... lógica para voar ...
    }
}

class Pinguim extends Passaro {
    @Override
    public void voar() {
        throw new RuntimeException("Pinguins não voam!");
    }
}
```

```
interface Ave {
    void mover();
}

class Passaro implements Ave {
    @Override
    public void mover() {
        // ... lógica para voar ...
    }
}

class Pinguim implements Ave {
    @Override
    public void mover() {
        // ... lógica para nadar ...
    }
}
```

Cuidados a Tomar - SOLID

Interface Segregation:É melhor ter várias interfaces específicas do que uma única interface geral, evitando que as classes dependam de métodos que não utilizam.

Exemplo Ruim:

```
interface Trabalhador {
    void trabalhar();
    void irAoEscritorio();
}

class TrabalhadorRemoto implements Trabalhador {
    @Override
    public void trabalhar() {
        // ... lógica para trabalhar remotamente ...
    }

    @Override
    public void irAoEscritorio() {
        throw new UnsupportedOperationException("Trabalhadores
 remotos não vão ao escritório!");
    }
```

Cuidados a Tomar - SOLID

Interface Segregation
Exemplo Bom:

```
interface Trabalhador {
    void trabalhar();
}

interface Presencial {
    void irAoEscritorio();
}

class TrabalhadorPresencial implements Trabalhador, Presencial {
    // ... implementações ...
}

class TrabalhadorRemoto implements Trabalhador {
    // ... implementações ...
}
```

Cuidados a Tomar - SOLID

Dependency Inversion:Classes devem depender de interfaces ou classes abstratas em vez de classes concretas, promovendo baixo acoplamento e maior flexibilidade.

Exemplo Ruim:

```
class Notificador {
    private SmsSender smsSender;

    public Notificador() {
        this.smsSender = new SmsSender();
    }

    public void enviarNotificacao(String mensagem) {
        smsSender.enviarSms(mensagem);
    }
}
```

Cuidados a Tomar - SOLID

Dependency Inversion
Exemplo Bom:

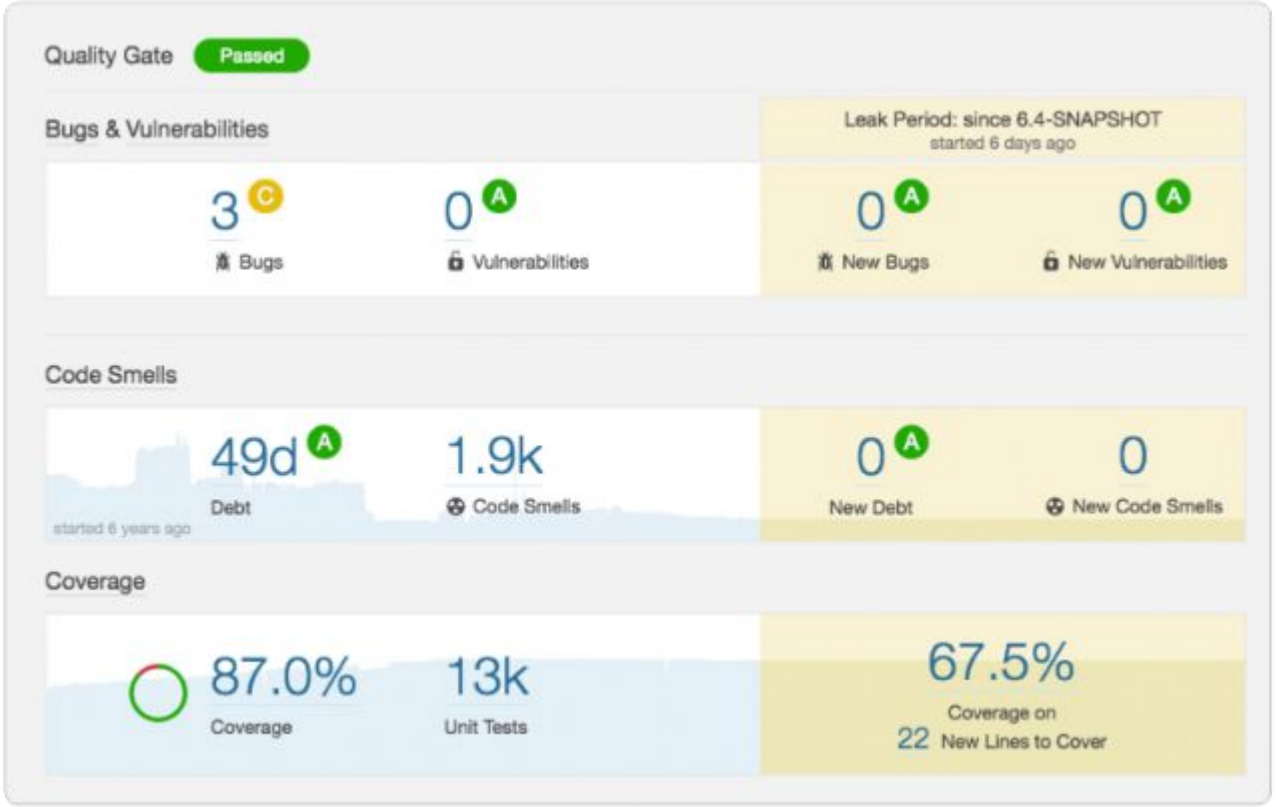
```java
interface Notificador {
    void enviarNotificacao(String mensagem);
}

class SmsNotificador implements Notificador {
    private SmsSender smsSender;

    public SmsNotificador(SmsSender smsSender) {
        this.smsSender = smsSender;
    }

    @Override
    public void enviarNotificacao(String mensagem) {
        smsSender.enviarSms(mensagem);
    }
}

class EmailNotificador implements Notificador {
    // ... implementação para enviar email ...
}
```

Ferramental

Ferramentas de Análise Estática:

Identificar automaticamente problemas de código como código duplicado, complexidade excessiva, más práticas, etc.

Exemplos: SonarQube, PMD, FindBugs, ESLint, Pylint.

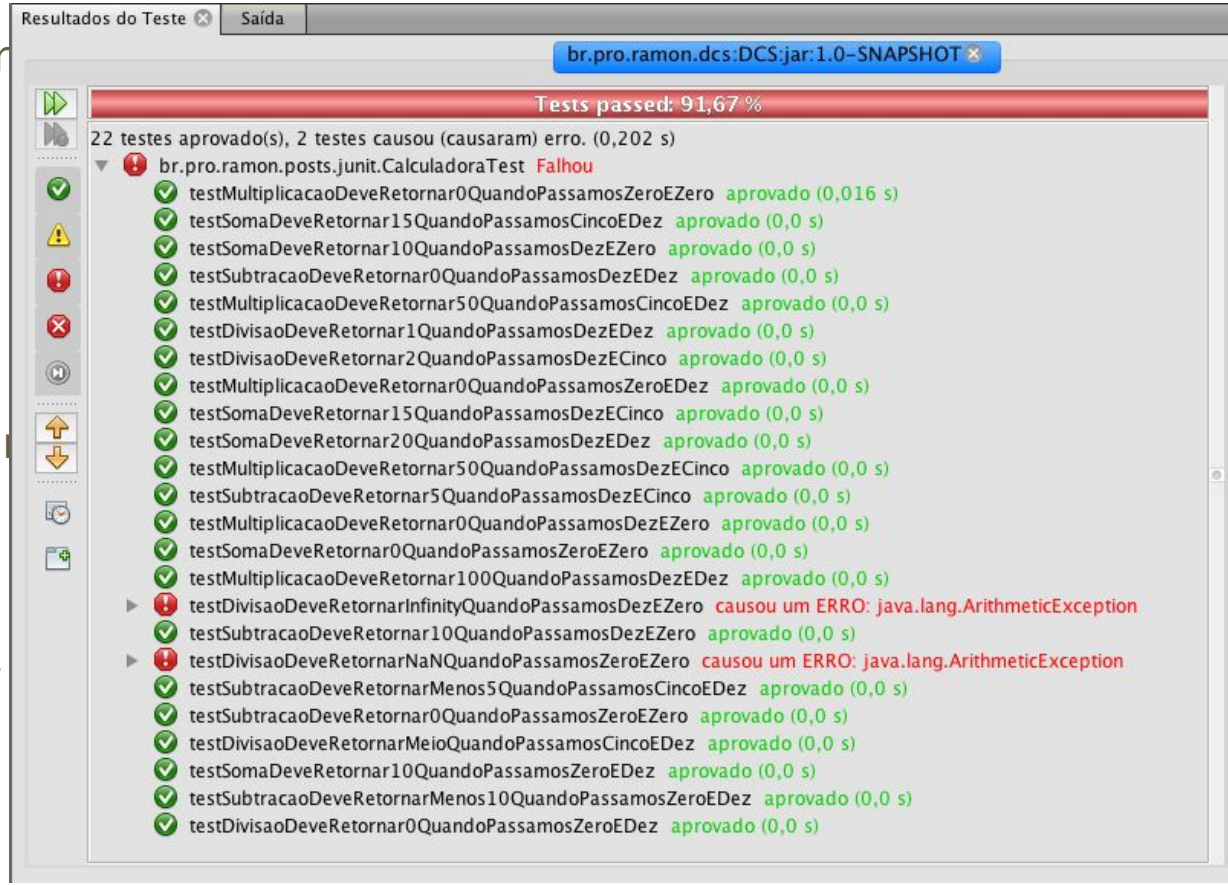


Ferramental

Testes Automatizados: Garantir a qualidade do código e evitar regressões (bugs que voltam)

Tipos de testes: Unidade (testar funções/métodos individualmente), Integração (testar a interação entre componentes), Fim-a-fim (simular o comportamento do usuário final).

Frameworks de teste: JUnit, pytest, Mocha, Jasmine. (Mostrar um exemplo simples de teste automatizado).



Ferramental

Refatoração: Melhorar a estrutura interna do código sem alterar seu comportamento externo.

Técnicas: Renomear variáveis/métodos para torná-los mais claros, dividir funções longas em partes menores, eliminar código duplicado, etc.

Ferramental - Refatorar

```java
public class CalculadoraPreco {

    public static void main(String[] args) {
        double preco1 = 100.0;
        int qtd1 = 2;
        double desc1 = 0.1;
        double precoFinal1 = preco1 * qtd1 * (1 - desc1);
        System.out.println("Preço final do produto 1: " + precoFinal1);

        double preco2 = 50.0;
        int qtd2 = 5;
        double precoFinal2 = preco2 * qtd2;
        System.out.println("Preço final do produto 2: " + precoFinal2);
    }
}
```

Ferramental - Refatorado

```java
public class CalculadoraPreco {

    public static void main(String[] args) {
        Produto produto1 = new Produto("Produto 1", 100.0);
        Produto produto2 = new Produto("Produto 2", 50.0);

        double precoFinal1 = calcularPrecoFinal(produto1, 2, 0.1);
        System.out.println("Preço final do " + produto1.getNome() + ": " +
precoFinal1);

        double precoFinal2 = calcularPrecoFinal(produto2, 5, 0.0); // Sem des
        System.out.println("Preço final do " + produto2.getNome() + ": " +
precoFinal2);
    }

    public static double calcularPrecoFinal(Produto produto, int quantidade,
double desconto) {
        double valorTotal = produto.getPrecoBase() * quantidade;
        return valorTotal * (1 - desconto);
    }
}
```

```java
class Produto {
    private String nome;
    private double precoBase;

    public Produto(String nome, double precoBase) {
        this.nome = nome;
        this.precoBase = precoBase;
    }

    // Getters
    public String getNome() {
        return nome;
    }

    public double getPrecoBase() {
        return precoBase;
    }
}
```

Ferramental

Code Review: Pedir a outro desenvolvedor para validar seu código no momento em que você deseja subir para algum ambiente.

Benefícios: Melhoria da qualidade do código, compartilhamento de informação entre o time, aprendizado do time, etc..

Pair Programming: Programar em conjunto, estar em chamada com outro desenvolvedor, discutindo e compartilhando informações sobre o código.

Benefícios: Compartilhamento de conhecimento, velocidade de desenvolvimento, menor número de bugs, etc..

Contatos

Linkedin:
www.linkedin.com/in/bruno-baruffi-esteves-78a88b36

GITHUB: https://github.com/brunobaruffi

Baixe a apresentação:
https://github.com/brunobaruffi/apresentacoesCompartilhadas

Baixe apresentação