# BOAS PRÁTICAS DE PROGRAMAÇÃO

SENAI <LAB365>

# SUMÁRIO

# O QUE É *CLEAN CODE*?

*Clean Code* ou Código Limpo é uma filosofia de desenvolvimento de software, que tem como principal objetivo aplicar boas práticas de programação para simplificar a leitura e escrita de códigos, deixando-os simples e facilmente entendíveis.


## Por que usar *Clean Code*?

Códigos precisam de manutenção e adição de novas funcionalidades, então em algum momento ele precisará ser lido e entendido por um desenvolvedor, por este motivo é tão importante escrever um código que seja fácil de ler e compreender o que ele faz.

Códigos Limpos estão altamente ligados à qualidade do produto desenvolvido e, consequentemente, à facilidade de dar manutenção e implementação de novos recursos.

> **Qualquer um consegue escrever código que um computador entende.**
> **Bons programadores escrevem código que humanos entendem.**
> Martin Fowler

A seguir, confira alguns exemplos de boas práticas.

# BOAS PRÁTICAS DE *CLEAN CODE*

Boas práticas de *Clean Code* são essenciais para uma boa produtividade no trabalho, para que diversos colaboradores possam trabalhar em um mesmo código e entender tudo que está acontecendo, confira a seguir alguns exemplos de boas práticas importantes.

## Nomes significativos

No processo de desenvolvimento de software, praticamente tudo deve ser nomeado, por exemplo, variáveis, funções, classes, métodos...

Para criar bons nomes podem ser seguidas algumas regras simples:

Devem ser escolhidos nomes que expressam o seu propósito de forma clara e objetiva, mesmo que isso gere um nome extenso.

Se uma função, variável ou qualquer outra coisa necessitar de comentários e não estiverem claras o suficiente, elas precisarão ser repensadas.

Para auxiliar na escolha de um bom nome, ele deve ser capaz de responder por que existe, o que faz e como é usado. Analise o código a seguir:

```
// Ruim: nome não intuitivo e necessita de comentário
String nc; // nome completo

// Bom: nome claro e objetivo
String nomeCompletoAluno;
```

A variável “nc” não indica que armazena um nome e nem que esse nome é completo. Já com a variável “nomeCompletoAluno” é possível entender o que ela armazena de forma clara e objetiva.

Para aplicar essa boa prática sempre deve ser escolhido um nome que ao olhar para o código já seja possível identificar o que ele representa sem a necessidade de um comentário.

Usar nomes pronunciáveis, pois palavras são feitas para serem pronunciadas e seria uma grande perda não tirar proveito disso.

Se o nome não for facilmente pronunciável irá dificultar as discussões e a leitura rápida do código. Inventar palavras não é uma boa prática.

Confira mais um exemplo a seguir:

```
// Ruim: não é possível entender o que a variável armazena
private Date genymdhms;

// Bom: é possível identificar o que a variável armazena
private Date generationTimestamp;
```

Usar nomes passíveis de busca, pois facilitam encontrar os códigos.

Nomes de uma só letra ou números possuem um problema por não serem fáceis de localizar ao longo de um texto grande, por exemplo, utilizar um nome apenas como “a” ao filtrar aparecerão milhões de resultados já que “a” é uma letra muito comum na maioria das palavras. Outra dica é utilizar constantes ao invés de utilizar números mágicos espalhados pelo código, pois é mais fácil de entender o que ela significa e também de pesquisar pela constante do que pelo número, confira a seguir:

```
//Ruim: o número 5 não deixa claro sobre o que se trata e é difícil de ser buscado
for (int i=0; i < 5; i++) {

}

//Bom: a constante deixa claro sobre o que se trata e é fácil de ser buscada
const int DIAS_TRABALHADOS_NA_SEMANA = 5;
for (int i=0; i < DIAS_TRABALHADOS_NA_SEMANA; i++) {

}
```

Nomes de classes e objetos devem evitar palavras conflitantes com a linguagem e preferencialmente devem ser utilizados substantivos em vez de verbos.

Entenda mais com um exemplo:

```
// Ruim: utiliza palavra reservada da linguagem Java
public class Final {

}

// Bom: utiliza substantivo
public class Aluno {

}
```

Nomes de métodos devem conter verbos e deixar claro para que servem.

Confira como realizar essa boa prática:

```
// Ruim: não deixa claro o que faz
String descricao() {

}

// Bom: deixa claro o que faz
String obterDescricao() {

}
```

A seguir, aprenda mais sobre a boa prática com o uso de condicionais.

## Condicionais

O uso de condicionais é parte essencial de uma aplicação e existem algumas regras que podem auxiliar na utilização delas para que fiquem fáceis de ler e de dar manutenção.

Muitas vezes, a lógica booleana pode ser complexa de entender até mesmo fora do contexto de operações condicionais e uma boa prática é extrair a lógica para funções que expliquem o propósito da estrutura condicional, com isso, além de estimular o reaproveitamento, irá facilitar a manutenção do código, pois se for necessário alterar a lógica, a alteração é realizada apenas em um único lugar, confira:

```
// Ruim: Não deixa claro sobre o que se trata a condicional
if (!timer.expirado() && !timer.recorrente()) {

}

// Bom: deixa claro sobre o que se trata a condicional
if (deveSerExcluido(timer)) {

}
```

Outra boa prática é evitar o uso de condicionais negativas, pois elas são mais complexas de serem entendidas do que condicionais positivas. Então, sempre que possível deve ser dada a preferência ao uso de condicionais positivas por serem mais simples de entender e consequentemente de dar manutenção ao código, entenda com o exemplo:

```
// Ruim: fica mais difícil de entender
if (!naoDeveSerAdicionado(aluno)) {

}

// Bom: fica fácil de entender
if (deveSerAdicionado(aluno)) {

}
```

Em seguida entenda mais sobre funções e elementos de boas práticas.

# Funções

Funções são essenciais para o desenvolvimento de sistemas, e também a primeira linha de organização em qualquer programa. Existem algumas regras que podem ser seguidas para deixar as funções simples e legíveis, confira alguns exemplos:

- Deve ser pequena, pois funções muito grandes dificultam a leitura e consequentemente o entendimento do que ela faz, com isso se tornam praticamente impossíveis de serem testadas de forma simples.

- Blocos e indentação devem possuir poucas linhas de código, por exemplo, dentro e um “if” deve ter o mínimo de linhas possível uma chamada para uma função. Outro ponto é o nível de indentação que deve ser de no máximo um ou dois níveis. Uma função com muitos “if” e “else” encadeados não é uma boa prática, pois dificultam o entendimento.

- A função deve fazer apenas uma única coisa, uma dica é seguir o conselho: “Funções devem fazer uma coisa apenas. Fazê-la bem. Fazer somente ela“. Uncle Bob. Muitas vezes é difícil de identificar o que é uma coisa apenas, para auxiliar nesse processo de identificação pode ser analisado o nome da função e a implementação dentro dela, se algum trecho dentro desta função não corresponde à ideia que está sendo passada pelo nome, ela está fazendo mais de uma coisa, e na maioria dos casos a lógica deve ser repensada, criando novas funções assim separando suas responsabilidades.

• Parâmetros de funções: a quantidade ideal de parâmetros para uma função é zero, um ou dois e sempre que possível deve ser evitado três ou mais parâmetros. Essa regra foi criada, pois é uma tarefa muito difícil complementar o nome de função com os nomes dos parâmetros e ainda assim garantir que ela faça apenas uma coisa e seja pequena.
A utilização de muitos parâmetros e principalmente se eles forem condicionais dificultam o teste da função, pois seria necessário criar cenários que garantam que todas as várias combinações de parâmetros funcionem adequadamente.

• Parâmetros lógicos, também conhecidos como flags, são valores booleanos passados a uma função. Essa é uma péssima prática que deve ser evitada, pois viola a regra de que a função deve ter apenas uma responsabilidade, já que se for verdadeira irá fazer uma coisa e se for falsa irá fazer outra.

• Evitar duplicação, sempre deve ser realizada uma análise das funcionalidades implementadas e caso seja percebida alguma semelhança entre as funções, elas podem ser transformadas em uma só e utilizadas em ambos os casos. Códigos duplicados dificultam a manutenção, pois se um comportamento mudar precisará ser replicado em vários locais, o que pode ocasionar esquecimentos e assim gerar erros na aplicação.

Códigos centralizados são mais simples de serem testados e também de serem mantidos. A seguir, são apresentados dois exemplos de funções, uma com boas práticas e outra sem a aplicação delas:

```
// Ruim: pois a função está bem grande e faz várias coisas
public static Funcionario criarFuncionario(Funcionario funcionario) {
  Object descontoSalario;
  if (funcionario.tipo == TIPO.CLT) {
     if (funcionario.salario <= FAIXA_SALARIAL_1)
        descontoSalario = {
           inss: 7.5,
           irrf: 0,
           fgts: 8
        };
     else if (funcionario.salario <= FAIXA_SALARIAL_2) {
       ....
     }
     ....
  } else if (funcionario.tipo == TIPO.ESTAGIARIO) {
    ....
  }
  funcionario.descontoSalario = descontoSalario;
  ...
  FuncionarioRespository = new FuncionarioRespository();
  FuncionarioRespository.save(funcionario);
  return funcionario;
}

// Bom: pois a função foi separada em outras menores, assim ficando pequena
e sendo responsável por fazer apenas uma única coisa.
public static Funcionario obterDescontoSalario(String tipo, float salario) {
   if (tipo == TIPO_CLT) {
      return obterDescontoSalarioCLT(salario)
   }
   if (tipo == ESTAGIARIO) {
      return obterDescontoSalarioEstagiario(salario)
   }
}

public static Funcionario atualizarDescontoSalario(Funcionario funcionario) {
   Object descontoSalario = obterDescontoSalario(funcionario.tipo, funcionario.sa-
lario);
   funcionario.atualizarDescontoSalario(descontoSalario)
}

public static Funcionario criarFuncionario(Funcionario funcionario) {
   FuncionarioRespository = new FuncionarioRespository();
   FuncionarioRespository.save(funcionario);
   return funcionario;
}
```

Os comentários também podem acontecer a partir de algumas dicas de boas práticas, confira a seguir.

## Comentários

Adicionar comentários em códigos é uma prática bastante comum, porém devem ser utilizados com moderação.

> **Não insira comentários num código ruim, reescreva-o.**
> Brian W. Kernighan e P. J. Plaugher.

Uma das maiores motivações para criar comentários é um código ruim, então para tentar deixar esse código compreensível adiciona-se comentários ao invés de limpar o código e reestruturá-lo de forma simples e intuitiva.

Códigos simples e expressivos com poucos comentários são muito mais vantajosos do que códigos complexos e com muitos comentários.

Comentários na maioria das vezes não são vantajosos, pois o código pode mudar, até ser removido e os comentários não serem alterados, fazendo com que o comentário expresse uma coisa, porém o código faça outra.

É sempre preferível criar funções com nomes intuitivos ou declarar constantes ao invés de adicionar comentários ao código, olhando o exemplo abaixo, é visível que a segunda opção onde é criado uma função com nome intuitivo é melhor que a primeira onde foi adicionado o comentário, confira um exemplo bom e um ruim a respeito dos comentários.

```
// Ruim: pois se o código mudar o comentário pode ficar obsoleto
// verifica se o funcionário é elegível para todos os benefícios
if ((funcionario.flags & POR_HORA_FLAG) & funcionario.idade > 65) {

}

// Bom: pois o nome é auto explicativo dispensando comentários
if (funcionario.ehElegivelParaTodosBeneficios()) {

}
```

Algumas boas práticas podem ser levadas em consideração para auxiliar na criação apenas de comentários relevantes:

- Não comentar o óbvio, comentários irrelevantes só servem para poluir o código.
- Não devem explicar o código, pois um código bem escrito dispensa comentários e deve ser autoexplicativo.
- Não deve ser usado para versionar o código, por exemplo, quando algo mudar, comentar o código antigo e escrever o novo sem removê-lo, para isso existem ferramentas de versionamento de código.
- Evitar comentários redundantes e desatualizados.
- Comentar apenas o que é relevante, como por exemplo, utilizar TODO para comentar algo que precisa ser implementado ou removido futuramente é um bom exemplo.

Aprenda mais adiante sobre boas práticas a respeito da formatação.

# Formatação

Ao olhar para um código, é esperado enxergar organização, consistência e atenção aos detalhes, pois isso facilita muito a leitura do código.

O principal objetivo da formatação de um código é servir como uma comunicação e deve ser tratado como a primeira regra de negócio de um desenvolvedor profissional, pode ser tratado como o cartão de visitas de um código.

Quando falamos em formatação alguns tópicos podem ser abordados:

A indentação é responsável por tornar os níveis visíveis, permitir visualizar facilmente escopos de “ifs”, “whiles”, “fors”, pois sem indentação códigos se tornam ilegíveis para humanos.

Formatação vertical trata-se do quão grande um arquivo pode ser, e isso vai depender muito, porém arquivos pequenos tendem a ser mais fáceis de serem lidos e mantidos.

Para auxiliar na formatação vertical pode ser utilizada a metáfora do jornal, em que o código seja como o artigo publicado.

- O nome deve ser simples e explicativo, ou seja, deve ser capaz de dizer se estamos no módulo certo ou não.
- No topo devem estar os conceitos e as funções de alto nível.
- Os detalhes e as funções de baixo nível devem ser adicionados mais abaixo.

Um jornal é composto de vários artigos e a maioria é bastante pequena, traduzindo isso para o código, podemos ter vários arquivos, classes e funções, porém eles devem ser pequenos.

Separação de conceitos, trata-se de utilizar linhas em branco para separar os conceitos da aplicação.

Levando em consideração que quase todo código é lido da esquerda para a direita e de cima para baixo, cada linha representa uma expressão e cada grupo de linhas representa um pensamento completo. Esses pensamentos devem ser separados por uma linha em branco, assim facilitando a identificação dos contextos, deixando o código muito mais legível.

A seguir, duas versões do mesmo código serão apresentadas, uma aplicando as boas práticas de formatação e a outra não. É claramente visível que a versão formatada deixa a leitura e entendimento do código muito mais simples, pois conseguimos identificar a separação dos contextos de forma clara, além de entender o que está dentro de cada bloco de código.

```
// Ruim: pois não aplica as boas práticas de formatação, o que deixa o código muito mais complexo e difícil de ler
import br.com.model.Aluno;
import br.com.model.dto.aluno.AlunoDTO;
import javax.servlet.http.HttpServletResponse;
public class AlunoController {
@Autowired
private AlunoService alunoService;
@GetMapping
public List < AlunoDTO > listAll() {
List < Aluno > alunos = alunoService.findAll();
return alunos.stream().map(a - > new
AlunoDTO(a)).collect(Collectors.toList());
   }
}
```

```
// Bom: pois aplica as boas práticas de formatação, o que deixa o código muito mais simples e fácil de ler
import br.com.model.Aluno;
import br.com.model.dto.aluno.AlunoDTO;

import javax.servlet.http.HttpServletResponse;

public class AlunoController {

    @Autowired
    private AlunoService alunoService;

    @GetMapping
    public List < AlunoDTO > listAll() {
        List < Aluno > alunos = alunoService.findAll();

        return alunos.stream().map(a - > new AlunoDTO(a)).collect(Collectors.toList());
    }
}
```

Gostou de aprender mais sobre algumas boas práticas? Esses elementos são todos muito importantes para o seu dia a dia em uma empresa. Eles te permitem trabalhar em equipe com mais sincronicidade e ajudam cada vez mais refinar a escrita de seus códigos. Práticas como essa, prepara você para o mercado de trabalho assim como aprender o código em si. Portanto, continue sempre a aprofundar seus conhecimentos, além de manter-se sempre atualizado em relação aos assuntos estudados. Lembre-se que a tecnologia está em constante evolução e manter-se a par de tudo é essencial.

# REFERÊNCIAS

MARTIN, R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Boston. Tearson Educaction Inc. 2009.

**Thais Cristina Bertoldo**

Experiência em desenvolvimento de software utilizando metodologias ágeis. No momento focada em desenvolvimento front-end utilizando React e arquitetura de micro front-end.
Atua como desenvolvedora front-end React na Gupy e Mentora educacional no LAB365 do SENAI/SC.

SENAI <LAB365>