

Resumo do livro “Clean Code”

Manutenção Produtiva Total

Foco: manutenção ao invés de produção.

Fundamentos: 5S princípios:

1. Ordenar: saber onde estão as coisas - usar abordagens com nomes adequados.
2. Sistematizar: um pedaço de código deve estar onde você espera encontrá-lo. Caso não esteja, refatore e o coloque lá.
3. Polir: comentários em excesso e sobre linhas de códigos passados são desnecessários, livre-se deles.
4. Padronizar: acordo entre a equipe para manter o trabalho seguindo os mesmos padrões.
5. Autodisciplina: seguir as práticas e estar disposto a mudar quando necessário.

Ideia defendida na página 12:

As pessoas que pensam que o código um dia desaparecerá são como matemáticos que esperam algum dia descobrir uma matemática que não precise ser formal. Elas esperam que um dia descubramos uma forma de criar máquinas que possam fazer o que desejamos em vez do que mandamos. Tais máquinas terão de ser capazes de nos entender tão bem de modo que possam traduzir exigências vagamente especificadas em programas executáveis perfeitos para satisfazer nossas necessidades.

Isso jamais acontecerá. Nem mesmo os seres humanos, com toda sua intuição e criatividade, têm sido capazes de criar sistemas bem-sucedidos a partir das carências confusas de seus clientes. Na verdade, se a matéria sobre especificação de requisitos não nos ensinou nada, é porque os requisitos bem especificados são tão formais quanto os códigos e podem agir como testes executáveis de tais códigos!

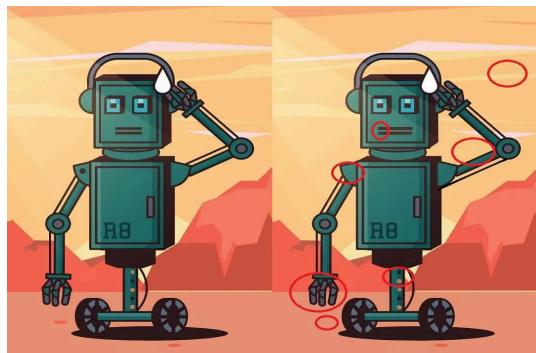
Código Ruim

Inicia o cap. falando sobre uma empresa que, em 1980, criou um aplicativo extraordinário, que se tornou muito popular na época. Contudo, o intervalo entre os lançamentos aumentava cada vez mais, devido a quantidade de *bugs* que necessitavam ser consertados. E, na tentativa de apressar o lançamento do produto, adicionando novos recursos, sem antes terem corrigido as pendências antigas, o código ficou impossível de gerenciar, causando a ruína da empresa.

No decorrer do cap., o autor defende que devemos descartar o pensamento de que “uma bagunça que funciona é melhor do que nada”.

Metáfora com o “Jogo dos 7 Erros”:

O código não precisa apenas funcionar. Assim que você torna o código funcional, você deve problematizá-lo, buscando possíveis erros. É como o “jogo dos 7 erros”, mas com valores desconhecidos de erros. Portanto, a próxima etapa, após criarmos um código funcional, é deixar a arrogância de lado e sempre considerar que haverá ao menos um erro, para, então, procurá-lo.



Consequências de um Código Ruim

Um código confuso faz com que trabalhemos mais lentamente e que a produtividade da equipe diminua. Na tentativa de recuperá-la, a gerência adiciona mais membros ao projeto, contudo, os novos membros não sabem a diferença entre uma mudança que altera o propósito do projeto e aquela que atrapalha, criando, assim, ainda mais confusão. O resultado de tudo isso: o código é criado do zero, com uma equipe mais restrita e especializada, com o objetivo de fazer um novo sistema que faça o mesmo que o antigo (sem os erros antigos).

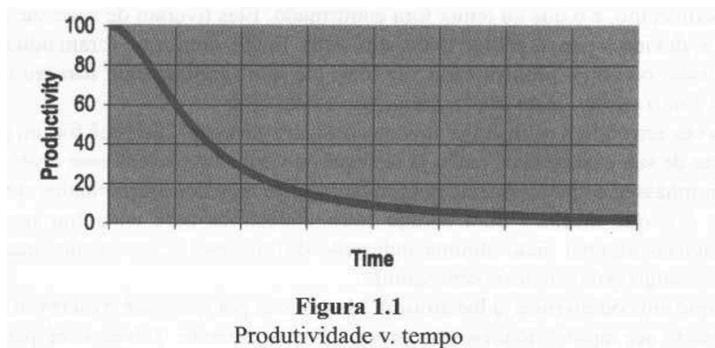


Figura 1.1
Produtividade v. tempo

De quem é a responsabilidade de um código ruim

Isso pode ser algo difícil de engolir. Mas como poderia essa zona ser *nossa* culpa? E os requisitos? E o prazo? E os tolos gerentes e tipos de marketing inúteis? Eles não carregam alguma parcela da culpa?

Não. Os gerentes e marketeiro buscam em nós as informações que precisam para fazer promessas e firmarem compromissos; e mesmo quando não nos procuram, não devemos dar uma de tímidos ao dizer-lhes nossa opinião. Os usuários esperam que validemos as maneiras pelas quais os requisitos se encaixarão no sistema. Os gerentes esperam que os ajudemos a cumprir o prazo. Nossa cumplicidade no planejamento do projeto é tamanha que compartilhamos de uma grande parcela da responsabilidade em caso de falhas; especialmente se estas forem em relação a um código ruim.

Para finalizar essa questão, e se você fosse médico e um paciente exigisse que você parasse com toda aquela lavação das mãos na preparação para a cirurgia só porque isso leva muito tempo?² É óbvio que o chefe neste caso é o paciente; mas, mesmo assim, o médico deverá totalmente se recusar obedecê-lo. Por quê? Porque o médico sabe mais do que o paciente sobre os riscos de doenças e infecções. Não seria profissional (sem mencionar criminoso) que o médico obedecesse ao paciente neste cenário. Da mesma forma que não é profissional que programadores cedam à vontade dos gerentes que não entendem os riscos de se gerar códigos confusos.

O que é um código limpo?

O autor reúne as respostas de alguns programadores que ele considera relevantes:

Bjarne Stroustrup, criador do C++ e autor do livro *A linguagem de programação C++*

Gosto do meu código elegante e eficiente. A lógica deve ser direta para dificultar o encobrimento de bugs, as dependências mínimas para facilitar a manutenção, o tratamento de erro completo de acordo com uma estratégia clara e o desempenho próximo do mais eficiente de modo a não incitar as pessoas a tornarem o código confuso com otimizações sorrteiras. O código limpo faz bem apenas uma coisa.



Baseado nisso, o autor afirma que um código ruim tenta fazer coisas demais e está cheio de propósitos, enquanto um código limpo é *centralizado* (cada função, cada classe, cada módulo expõe uma única tarefa). Além disso, não há duplicidade (mais de um módulo realizando a mesma tarefa).

Grady Booch, autor do livro *Object Oriented Analysis and Design with Applications*

Um código limpo é simples e direto. Ele é tão bem legível quanto uma prosa bem escrita. Ele jamais torna confuso o objetivo do desenvolvedor. Em vez disso, ele está repleto de abstrações claras e linhas de controle objetivas.



O autor reflete sobre a ideia de “abstrações claras”, afirmando que o código deve ser decisivo e sem especulações. Em outras palavras, o código deve parecer abstrato apenas para leigos no assunto.

O “grande” Dave Thomas, fundador da OTI, o pai da estratégia Eclipse

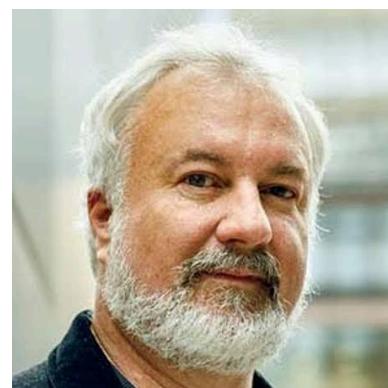
Além de seu criador, um desenvolvedor pode ler e melhorar um código limpo. Ele tem testes de unidade e de aceitação, nomes significativos; ele oferece apenas uma maneira, e não várias, de se fazer uma tarefa; possui poucas dependências, as quais são explicitamente declaradas e oferecem um API mínimo e claro. O código deve ser inteligível já que dependendo da linguagem, nem toda informação necessária pode expressa no código em si.



Referente a isso, o autor afirma que, não importa o qual elegante, legível ou acessível o código esteja, se ele não possuir testes, ele não é limpo.

Michael Feathers, autor de *Working Effectively with Legacy Code*

Eu poderia listar todas as qualidades que vejo em um código limpo, mas há uma predominante que leva a todas as outras. Um código limpo sempre parece que foi escrito por alguém que se importava. Não há nada de óbvio no que se pode fazer para torná-lo melhor. Tudo foi pensado pelo autor do código, e se tentar pensar em algumas melhorias, você voltará ao inicio, ou seja, apreciando o código deixado para você por alguém que se importa bastante com essa tarefa.



O autor parte dessa fala para chegar a conclusão que um bom código é escrito por alguém que se importe em fazer um bom trabalho, que verdadeiramente se dedique à tarefa.

Nomes Significativos

Use nomes que revelem seu propósito

O nome de uma variável, função ou classe deve dizer porque existe, o que faz e como é usado. Se um nome requer um comentário, então ele não revela o seu propósito.

```
int d; // tempo decorrido em dias
```

O nome `d` não revela nada. Ele não indica a ideia de tempo decorrido, nem de dias. Devemos escolher um nome que especifique seu uso para mensuração e a unidade usada.

```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

Exemplo simples com nomenclatura ruim:

```
def f1(a, b):  
    return a * b  
x = 5  
y = 10  
r = f1(x, y)  
print(r)
```

Problemas:

- `f1` não indica o que a função faz.
- `a` e `b` não dizem que representam a base e a altura.
- `x`, `y` e `r` não deixam claro o que armazenam.

Exemplo corrigido para um código limpo:

```
def calcular_area_retangulo(base, altura):  
    return base * altura  
  
largura = 5  
altura = 10  
area = calcular_area_retangulo(largura, altura)  
  
print(f"A área do retângulo é {area}")
```

Melhorias:

- `calcular_area_retangulo` deixa claro o propósito da função.
- `base` e `altura` são descritivos.
- `largura`, `altura` e `area` tornam o código legível.

Funções

Funções devem ser pequenas

Não há uma regra rígida quanto ao número de linhas para uma função, mas muitas convenções sugerem que uma função não deve ter mais do que **20 a 30 linhas**, se possível. Se ela exceder esse tamanho, é uma boa prática verificar se é possível dividi-la em funções menores, mais focadas.

Uma função deve fazer apenas uma coisa (Princípio da Responsabilidade Única)

Cada função deve cumprir **uma tarefa específica**. Isso torna o código mais legível, fácil de entender e de manter. Se uma função começar a realizar **muitas tarefas diferentes** (por exemplo, fazer cálculos, manipular dados e exibir resultados), isso pode indicar que ela está **muito grande** e deve ser dividida em funções menores.

Exemplo de um código que precisa ser dividido:

```
def boas_vindas():
    nome = input("Qual é o seu nome? ")
    print(f"Bem-vindo, {nome}!")

boas_vindas()
```

Problema:

- Este código faz tudo em uma única função, desde ler o nome até imprimir a mensagem.

Código corrigido:

```
def obter_nome():
    return input("Qual é o seu nome? ")

def exibir_boas_vindas(nome):
    print(f"Bem-vindo, {nome}!")

def executar():
    nome = obter_nome()
    exibir_boas_vindas(nome)

executar()
```

Melhorias:

- `obter_nome` pede e retorna o nome da pessoa.
- `exibir_boas_vindas` exibe a mensagem de boas-vindas.
- `executar` chama as funções para realizar a tarefa de pedir o nome e exibir a mensagem.

Vantagens:

- **Modularidade:** Ao dividir o código em funções menores, você pode reutilizar partes do código em diferentes contextos e locais do programa, sem precisar duplicar a lógica. Por exemplo, se você precisar obter o nome em outra parte do seu programa, sem a necessidade de reescrever o código para pedir o nome do usuário, você pode simplesmente usar a função `obter_nome`.
- **Facilidade de manutenção:** Se precisar modificar uma parte específica do programa, você pode fazê-lo em uma função separada, sem afetar outras partes do código. Por exemplo, se você quiser mudar a mensagem de boas-vindas, pode alterar apenas a função `exibir_boas_vindas`, sem precisar mexer em outras partes do código.

Uso de Estrutura de Controle Condisional:

Usar uma estrutura de controle condicional não é um problema. Contudo, usá-la de maneira indiscriminada sim. Ela só deve ser usada em cenários em que a função não irá crescer com o tempo. Imagine que temos um sistema que realiza diferentes operações matemáticas com base na escolha do usuário. Usaremos uma série de `if/else` para demonstrar a ideia.

```
def calcular(operacao, a, b):  
  
    if operacao == "soma":  
        return a + b  
    elif operacao == "subtracao":  
        return a - b  
    elif operacao == "multiplicacao":  
        return a * b  
    elif operacao == "divisao":  
        return a / b  
    else:  
        print("Operação desconhecida!")  
  
print(calcular("soma", 5, 3))  
print(calcular("divisao", 10, 2))
```

Se um novo tipo de operação precisar ser adicionado (como factorial), será necessário voltar à função `calcular` e adicionar outro `elif`. Isso vai tornando a função cada vez maior e mais difícil de manter.

Além disso, quando o código cresce, o código fica **menos modular**, ou seja, uma parte do código tem várias responsabilidades, e você acaba criando redundâncias que tornam o código mais propenso a erros.

Para mitigar os problemas de manutenção e expansão, podemos **modularizar** esse código, separando as operações em funções independentes e utilizando uma estrutura como um **dicionário**.

Por: Alice Dantas

Código melhorado:

```
def soma(a, b):
    return a + b

def subtracao(a, b):
    return a - b

def multiplicacao(a, b):
    return a * b

def divisao(a, b):
    if b == 0:
        raise ValueError("Divisão por zero não permitida!")
    return a / b

def fatorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * fatorial(n - 1)

# Mapeamento das operações para funções
operacoes = {
    "soma": soma,
    "subtracao": subtracao,
    "multiplicacao": multiplicacao,
    "divisao": divisao,
    "fatorial": fatorial
}

def calcular(operacao, a, b=None):
    if operacao not in operacoes:
        print("Operação desconhecida!")
        return None

    if operacao == "fatorial":
        # Fatorial só precisa de um número
        return operacoes[operacao](a)
    else:
        # Outras operações precisam de dois números
        return operacoes[operacao](a, b)

print(calcular("soma", 5, 3))
print(calcular("divisao", 10, 2))
print(calcular("fatorial", 5))
```

Parâmetros:

Parâmetros de Entrada:

Evitar usar parâmetros desnecessariamente é uma boa prática. Isso se aplica especialmente nos casos em que uma função não precisa de uma informação externa¹, quando ela só usa valores fixos², ou quando ela busca os dados por conta própria³.

Exemplos em que os parâmetros são desnecessários:

1.

```
def mensagem_agradecimento():
    print ("Obrigada por ter
acessado nosso site, volte
sempre!")
```

2.

```
def mostrar_pi():
    print(3.14159)
```

3.

```
def ler_arquivo():
    with open('dados.txt') as f:
        print(f.read())
```

Parâmetros de Lógicos:

O autor considera essa uma das piores práticas na hora de definir parâmetros.

Essa prática é problemática pois indica que a função está fazendo mais de uma coisa, diminui a clareza na hora de chamar a função, e dificulta a manutenção e evolução do código.

Exemplo de uso Parâmetro Lógico:

```
def salvar_usuario(usuario, enviar_email=False):
    print(f"Usuário {usuario} salvo no banco de
dados.")

if enviar_email:
    print(f"E-mail enviado para {usuario}.")

salvar_usuario("João", enviar_email=True)
```

Porque esse padrão não é o ideal:

1. Mistura duas responsabilidades diferentes na mesma função:

- Uma função deve ter **uma única responsabilidade** (Princípio da Responsabilidade Única).
- Aqui, ela está: Salvando o usuário; Enviando um e-mail (que é uma outra ação, separada)

2. Diminui a clareza na hora de chamar a função:

- Ao ler `salvar_usuario("João", True)`, não fica óbvio o que o `True` faz.
- Quem lê precisa olhar a função para entender.

3. Dificulta a manutenção e evolução do código:

- Se o envio de e-mail mudar para usar um serviço externo você terá que mexer nessa função que também lida com banco de dados.

Quando usar RETURN:

O uso correto do Return permite evitar efeitos colaterais. Um **efeito colateral** acontece quando uma função **altera algo fora dela**, além de simplesmente **retornar um valor**.

1. Exemplo de uma função sem efeito colateral:

```
def soma(a, b):  
    return a + b
```

Essa função **não altera nada fora dela**, só recebe dois números e retorna a soma.

É previsível: sempre que você passar os mesmos valores, ela vai dar o mesmo resultado.

2. Exemplo de uma função com efeito colateral:

```
saldo = 100
```

```
def sacar(valor):  
    global saldo  
    saldo -= valor
```

A função altera a variável global **saldo**. Esse é um efeito colateral porque algo **fora da função** mudou. Toda vez que **sacar()** é chamada, o estado do programa muda, o que pode gerar erros difíceis de encontrar.

É possível que surja a dúvida “e se eu retirasse o ‘global’?”. Ao retirar o “global”, o código não funcionaria, pois “saldo” seria considerado uma variável local, a qual não foi inicializada dentro da função. É como se o programa pensasse: “Ok, saldo é uma variável local... mas pera aí... eu não sei o valor dela ainda porque você não declarou nada localmente!”. Por isso, o ideal é passar o “saldo” como **argumento e retornar o novo valor**, evitando efeitos colaterais:

```
def sacar(saldo, valor):  
    return saldo - valor
```

Em resumo, sempre que a função **precisar gerar um valor**, use **return**. Use funções sem **return** apenas quando o foco for **executar uma ação** que não precisa entregar um resultado, como imprimir algo na tela.

Tratamento de erros

Tratamento de erro é o conjunto de técnicas usadas para prever e lidar com situações inesperadas que podem acontecer durante a execução de um programa. A ideia é impedir que o programa trave ou quebre quando algo dá errado.

Em vez de deixar o erro explodir e encerrar o programa, você controla o que acontece e decide como o sistema deve reagir. Vamos exemplificar com o uso do try/except, em python.

Blocos try/except:

Os blocos `try/except` são usados em Python para tratar exceções ou erros que podem ocorrer durante a execução do código, permitindo que o código continue sendo executado de maneira controlada.

Como funciona:

try: Você coloca o código que **pode gerar um erro** dentro do bloco `try`.

except: Caso o código dentro do `try` **gere uma exceção (erro)**, o Python **pula** para o bloco `except`.

Exemplo:

`try:`

```
# Código que pode gerar um erro  
resultado = 10 / 0 # Exemplo de erro (divisão por zero)
```

`except ZeroDivisionError:`

```
# Tratamento do erro  
print("Erro: Não é possível dividir por zero.")
```

Exceções:

Os blocos `try/except` permitem que você controle erros de forma elegante e continue a execução do programa, ao invés de ele travar inesperadamente. No entanto, como toda ferramenta, elas devem ser usadas de forma cuidadosa.

Ou seja, use para capturar erros que são **previsíveis** e podem ser tratados com uma mensagem útil, assim, você pode capturar e tratar o erro para que o usuário tenha uma **mensagem amigável**, em vez de uma mensagem técnica de erro do Python.

Jamais o utilize para esconder erros de forma indiscriminada ou quando o erro pode ser evitado de outra forma.

Comentários

O autor defende o uso de comentários em situações extremamente necessárias como comentários para explicar o motivo das decisões, documentar funções/métodos, marcar tarefas pendentes e explicar lógicas complexas.

Ideia defendida na página 60:

O uso adequado de comentários é compensar nosso fracasso em nos expressar no código. Observe que usei a palavra fracasso. E é isso que eu quis dizer. Comentários são sempre fracassos.

Uma das motivações mais comuns para criar comentários é um código ruim. Construímos um módulo e sabemos que está confuso e desorganizado. Estamos cientes da bagunça. Nós mesmos dizemos “Oh, é melhor inserir um comentário!”. Não! É melhor limpá-lo.

Quando o uso de comentários não é uma boa ideia:

1. **Comentários Redundantes:** Comentários que simplesmente repetem o que o código já está fazendo não agregam valor. Ex.:

```
x = 10 # Atribuindo 10 à variável x
```

2. **Comentários Obsoletos:** Comentários desatualizados que não refletem mais o que o código faz podem ser confusos e levar a erros. Isso pode acontecer quando o código é alterado, mas o comentário não é atualizado para refletir as mudanças. Ex.:

```
def calcular_valor_com_desconto(preco):
    # TODO: Implementar o desconto de 10%
    return preco * 0.9
```

3. **Comentários Excessivos:** Embora alguns comentários sejam necessários, um excesso deles pode tornar o código difícil de ler e entender. Isso ocorre quando você tenta "explicar" cada linha de código, tornando-o visualmente desordenado. Ex.:

```
# Primeiro, criamos a variável x
x = 10 # Atribuimos o valor 10 a x
# Agora criamos a variável y
y = 20 # Atribuimos o valor 20 a y
# Agora, somamos x e y
soma = x + y # Somamos os valores de x e y
# Finalmente, imprimimos o resultado da soma
print(soma) # Imprimimos o resultado da soma na tela
```

Formatação

A formatação adequada torna o código mais legível.

Indentação e Espaçamento:

- Use indentação consistente para indicar blocos de código (normalmente 2 ou 4 espaços).
- Separe trechos de código relacionados com linhas em branco para melhorar a leitura.