Bumper-Sticker Teoria da computação

Coluna 6 do livro *More Programming Pearls* (Jon Bentley)

Disciplina: Teoria da Computação (CCSMB) — UVV 2026

Alunos: André Luiz, Thomas Krieger

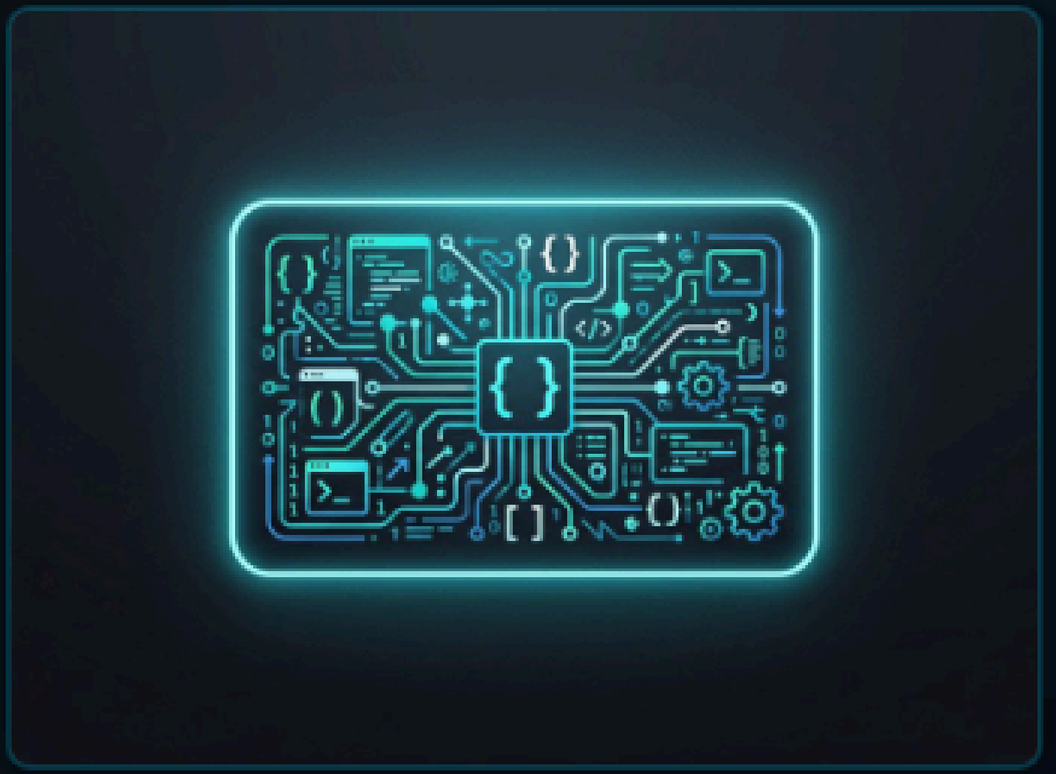
O que é Bumper-Sticker CS?

Apresentado por Jon Bentley no livro *More Programming Pearls*.

Reúne **frases curtas** com sabedoria prática sobre programação e desenvolvimento de software.

Inspirado nos clássicos *bumper stickers* (adesivos de para-choque de carro).

O objetivo central é **condensar décadas de experiências acumuladas** por programadores em regras fáceis de lembrar.



Origem das Frases

A maioria foi enviada por leitores de publicações da **ACM** (Association for Computing Machinery). Elas não são teorias acadêmicas, mas lições de trincheira que representam:

Experiências Reais

Histórias e vivências reais de desenvolvimento de software em produção.

Erros Comuns

Padrões de falha recorrentes e armadilhas (pitfalls) a serem evitadas.

Boas Práticas

Regras de ouro e heurísticas para escrever código limpo e mantenível.

O Valor das Estimativas Rápidas

Programadores frequentemente precisam fazer cálculos "de guardanapo" para validar ideias:

O que estimar?

- Tempo de execução de rotinas
- Capacidade de processamento/memória
- Limites de escalabilidade (Big O)

Qual a utilidade?

- **Planejar** projetos com viabilidade
- **Avaliar** gargalos de desempenho
- **Evitar** soluções arquiteturais inviáveis

Exemplo de Estimativa de Padaria

Pergunta: Quanto tempo levaria para processar **1.000.000 de registros**?

1. Premissa

100 reg/s

2. Divisão Bruta

1M ÷ 100 =

10.000 s

3. Conversão

10k s ≈

3 horas

Mensagem de bolso: Uma conta simples de 5 segundos fornece intuição imediata sobre o desempenho esperado.

A Regra do Nanocentury

10M segundos ≈ **4** meses

Um **nanoséculo** (*nanocentury*) é uma aproximação comum em computação.

Ele ajuda programadores a fazer **estimativas rápidas de tempo** em sistemas de processamento de dados massivos.

> *"The sooner you start to code, the longer the program will take."*

Começar a programar sem planejamento gera **retrabalho**.
Entender a fundo o problema antes de abrir o editor
reduz erros estruturais.

Se Não Dá Para Explicar, Não Dá Para Codar

"If you can't write it down in English, you can't code it."

Se você não consegue explicar claramente o problema, **você ainda não o compreendeu** o suficiente para resolvê-lo em código.

Especificações claras facilitam:

1. Desenvolvimento
2. Manutenção
3. Testes

Os Detalhes Importam

"Details count."

Na prática da engenharia de software:

Pequenos erros podem gerar **grandes problemas** em escala.

Um detalhe incorreto pode causar **bugs difíceis de identificar**.

Código e Comentários

"If the code and the comments disagree, both are probably wrong."

Problemas Comuns

- Comentários desatualizados
- Documentação incorreta

Boas Práticas

- Escrever código legível
- Manter docs atualizadas

Estruturas de Dados Primeiro

"Get your data structures correct first."

A estrutura de dados correta resolve grande parte do problema algorítmico, definindo a verdadeira **eficiência do software**.

- **Mais rápido**
- **Mais simples**
- **Fácil de manter**

Prática: Busca em Lista vs Set

Problema: verificar se o usuário "João" existe na estrutura de dados.

Usando Lista (Array)

```
usuarios = ["Ana", "Carlos", "João"]
if "João" in usuarios:
print("Encontrado")
```

Desempenho O(n): Precisa percorrer os elementos até encontrar. Custa mais tempo se a lista for longa.

VS

Usando Conjunto (Set)

```
usuarios = {"Ana", "Carlos", "João"}
if "João" in usuarios:
print("Encontrado")
```

Desempenho O(1): Busca por hash é muito mais rápida, ilustrando a força da estrutura correta.

O Princípio do Menor Espanto

Comportamento Previsível

Consistência

O usuário **não deve** precisar adivinhar.

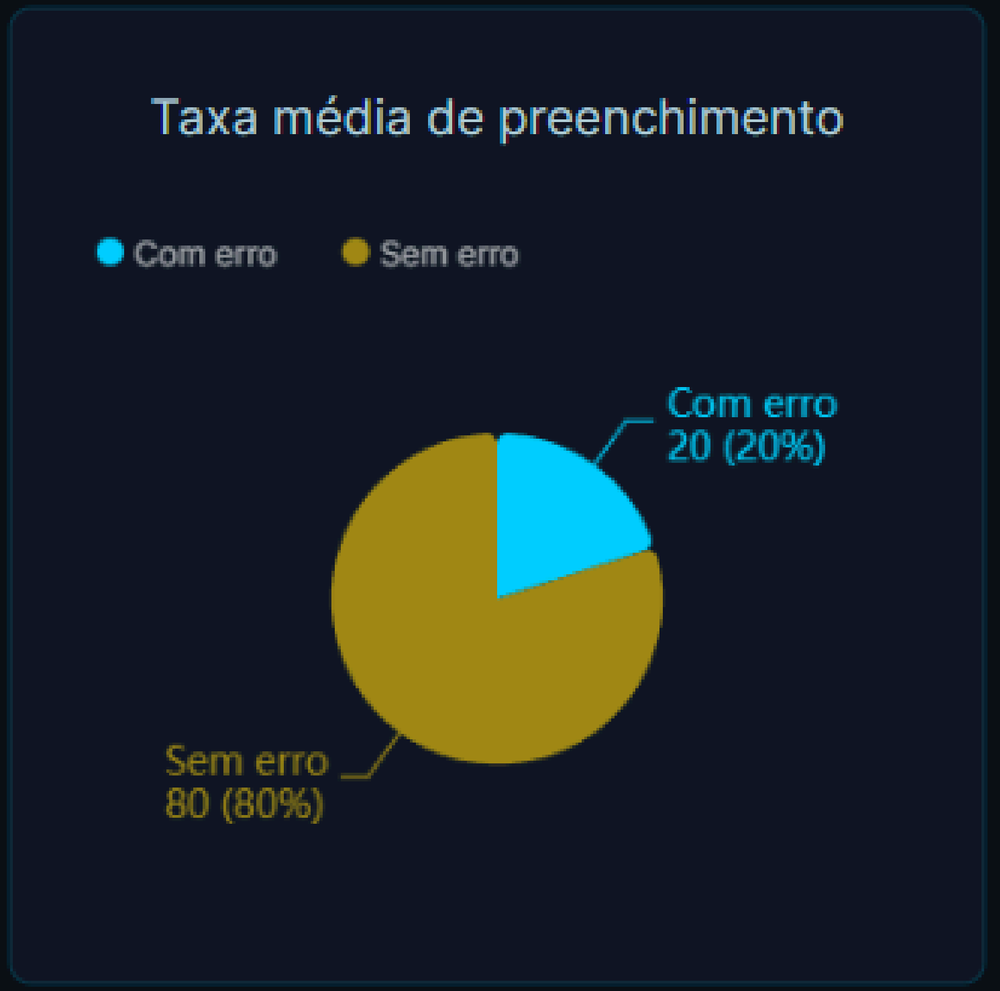
Facilidade de Uso

Não Confie na Entrada do Usuário

Um problema comum em sistemas é confiar demais nos dados enviados.

Por isso, é essencial sempre:

- **Validar** todos os dados recebidos
- **Tratar erros** graciosamente
- **Evitar falhas** no sistema (crashes)



"Testing can show the presence of bugs, but not their absence."

— **Edsger W. Dijkstra**

Testes comprovam que existem erros, mas **nunca garantem** que um software está 100% livre deles.

O Processo de Debugging

"The first step in fixing a bug is making it reproducible."

Recriar o ambiente e ações que disparam o erro consistemente.

Analisar o contexto, logs e momento exato em que ocorre.

Localizar o código causador e preparar a correção.

Otimização

Regras Fundamentais

1. Não otimizar **prematuramente**.
2. **Medir** antes de otimizar.

Efeito Pareto no Código

< 4% do código consome **> 50%** do tempo

Em muitos programas, o "gargalo" é extremamente pontual. Identifique os pontos críticos antes de tentar melhorar o código todo.

Achar o **gargalo certo** é mais útil do que micro-otimizações.

Conclusão

A **Coluna 6** evidencia que décadas de "dor de cabeça" em programação podem ser destiladas em regras práticas e simples.

Evitar Erros

Comuns do Dia a Dia

Decidir Melhor

Com Mais Clareza

Software Eficiente

Arquitetura Sólida

"Pequenas frases podem transmitir grandes lições da ciência da computação."

Obrigado!

Alguma pergunta?