Seminário: Testes, Depuração e Andaimes de Software

Baseado em *More Programming Pearls*, Coluna 3 — **Confessions of a Coder**

Guilherme Altoé Tomazini Kaiky Viglioni Tavares Moura
Marco Antônio Faustini Pessoa

1 Introdução: O Papel do *Scaffolding*

Testar e depurar são atividades que consomem grande parte do tempo de um programador, mas raramente recebem atenção adequada na literatura. A Coluna 3 de *More Programming Pearls*, de Jon Bentley, aborda justamente esse aspecto, com ênfase em uma ferramenta simples e poderosa: o ***scaffolding*** (andaime de software).

A metáfora é direta: assim como andaimes de construção civil dão acesso a partes de um edifício que os trabalhadores não alcançariam de outra forma, o *scaffolding* de software consiste em **programas e dados temporários** que permitem ao programador isolar, observar e exercitar componentes do sistema de forma independente. Esse código auxiliar nunca é entregue ao cliente, mas é indispensável durante o desenvolvimento.

A coluna estrutura-se em torno de duas "confissões" do próprio autor — falhas reais causadas pela ausência de testes adequados — e depois apresenta dois estudos de caso detalhados que ilustram como o *scaffolding* teria evitado, ou de fato revelou, os bugs em questão.

2 As Confissões: Motivação Narrativa

Confissão 1. Em um programa de 500 linhas, Bentley copiou uma sub-rotina de seleção de 20 linhas de um livro reconhecido de algoritmos. O programa funcionava na maioria dos casos, mas falhava esporadicamente. Dois dias de depuração foram necessários para rastrear o problema até a sub-rotina — onde um `<` no livro deveria ser um `<=`. A conclusão é dura: quinze minutos de *scaffolding* em torno da rotina teriam exibido o bug imediatamente.

Confissão 2. Ao escrever seu próprio livro, Bentley derivou uma rotina de seleção por meio de técnicas de verificação formal e ficou tentado a não testá-la. Após hesitar, decidiu testar — e encontrou um bug que o próprio raciocínio formal não havia capturado. A lição é clara: *verificação formal e testes experimentais são complementares, não substitutos.*

3 Estudo de Caso: Busca Binária

O primeiro estudo de caso é uma busca binária implementada em Awk. O autor adiciona uma única instrução de impressão como *scaffolding* para tornar visível o estreitamento do intervalo `[l, u]` a cada iteração:

```
function bsearch(t, l, u, m) {
    l = 1; u = n
    while (l <= u) {
        m = int((l+u)/2)
        print " ", l, m, u   # scaffolding: imprime l, m, u
        if (x[m] < t) l = m
        else if (x[m] > t) u = m
        else return m
    }
    return 0
```

```
}
```

Ao buscar o elemento 50 em um array de cinco elementos, a saída de diagnóstico revelou um loop infinito:

```
search 50
1 3 5
3 4 5
4 4 5    <-- l == m: o intervalo para de estreitar
4 4 5
4 4 5    (continua indefinidamente)
```

Quando `l == m`, a atribuição `l = m` não avança o limite inferior e o intervalo nunca estreita. A correção é substituir por `l = m + 1` (e `u = m - 1` no caso simétrico), excluindo `m` do novo intervalo e garantindo a terminação. O custo do *scaffolding* foi **uma única linha de impressão**; sem ela, esse bug poderia exigir horas de depuração ao emergir no interior de um sistema maior.

4 Estudo de Caso: Seleção do k-ésimo Menor Elemento

O segundo estudo de caso é uma rotina de seleção baseada no algoritmo de Hoare, que rearranja `x[1..n]` de modo que `x[1..k-1]` $\leq$ `x[k]` $\leq$ `x[k+1..n]`. A versão com recursão de cauda foi reescrita como um loop `while` e, conforme a segunda confissão, Bentley hesitou em testá-la.

O *scaffolding* adicionado imprimia os limites `l` e `u` a cada iteração e também verificava a corretude do resultado final, inspecionando todos os elementos à esquerda e à direita de `x[k]`:

```
# Verificacao pos-execucao (scaffolding de caixa-preta):
for (i = 1;   i < k;  i++) if (x[i] > x[k]) print i
for (i = k+1; i <= n; i++) if (x[i] < x[k]) print i
```

O bug permanecia oculto na maioria dos casos porque o *swap* aleatório do pivô frequentemente o contornava. Somente ao testar um array degenerado — todos os elementos iguais — o loop infinito foi exposto:

```
fill 2
x 1 5
x 2 5     # array: [5, 5]
sel 1     # busca o 1o menor elemento
1 2
1 2       # l e u nao convergem: loop infinito
1 2
```

Esse caso ilustra um ponto importante: **a aleatoriedade pode mascarar bugs**. Casos degenerados — todos os elementos iguais, array de um elemento, array já ordenado — são essenciais para expor falhas que entradas aleatórias escondem.

5 A Biblioteca de Sub-rotinas e os Testes de Caixa-Preta

Motivado pelas colunas anteriores, Bentley compilou uma biblioteca de algoritmos clássicos em Awk — linguagem escolhida pela brevidade e clareza. A biblioteca inclui busca binária, seleção, ordenação, heaps e estruturas associadas.

O aspecto mais revelador é a proporção entre código de produto e *scaffolding*: as sub-rotinas representam *menos da metade* do texto total do programa; o restante são testes de corretude que constroem entradas, chamam as rotinas e verificam as saídas automaticamente. Fred Brooks (*The Mythical Man Month*) estima que o *scaffolding* pode representar metade do código entregável; Knuth eleva essa estimativa a uma proporção de 1:1.

Durante os testes, o *scaffolding* revelou não apenas bugs no código do autor, mas também **dois bugs no próprio interpretador Awk**: um relacionado ao `return` dentro de loops em funções, e outro de comparação incorreta entre strings numéricas (`"10"` antes de `"5"` na ordenação lexicográfica). Em ambos os casos, Bentley reproduziu o comportamento em programas de 6 e 15 linhas antes de reportar os problemas — prática que Brian Kernighan corrigiu em minutos. A regra geral derivada desse episódio é direta: *reporte bugs com o menor caso de teste possível.*

6 Conclusão e Princípios

Os exemplos e confissões da coluna convergem para um conjunto coeso de princípios práticos:

1. **Construa *scaffolding* cedo:** poucos minutos de instrumentação em uma sub-rotina complexa podem economizar dias de depuração após a integração ao sistema.
2. **Combine abordagens de teste:** a visão de caixa-branca confirma que o código se comporta como esperado durante o desenvolvimento; a caixa-preta aumenta a confiança na corretude final; a verificação formal ajuda a derivar o código corretamente desde o início.
3. **Teste casos degenerados:** entradas aleatórias não são suficientes. Arrays com elementos iguais, tamanho um, e limites extremos são essenciais para expor bugs que a aleatoriedade mascara.
4. **Use linguagens de suporte:** Awk, Python e similares são ideais para prototipar algoritmos sutis rapidamente. O custo de reimplementar em uma linguagem expressiva é amplamente compensado pela facilidade de teste e observação.
5. **Reporte bugs com exemplos mínimos:** um caso de teste de 10 linhas que reproduz o problema é infinitamente mais útil do que um relatório com 25.000 linhas de programa.

Em última análise, a mensagem da coluna é modesta e honesta: mesmo programadores experientes introduzem bugs em rotinas que parecem triviais ou formalmente corretas. A diferença entre desperdiçar dois dias e gastar quinze minutos está, quase sempre, na disposição de construir um andaime antes de subir.