PUC Minas

TESTE DE SOFTWARE

Análise de Eficácia de Testes com Teste de Mutação

Larissa Pedrosa Silva

801161

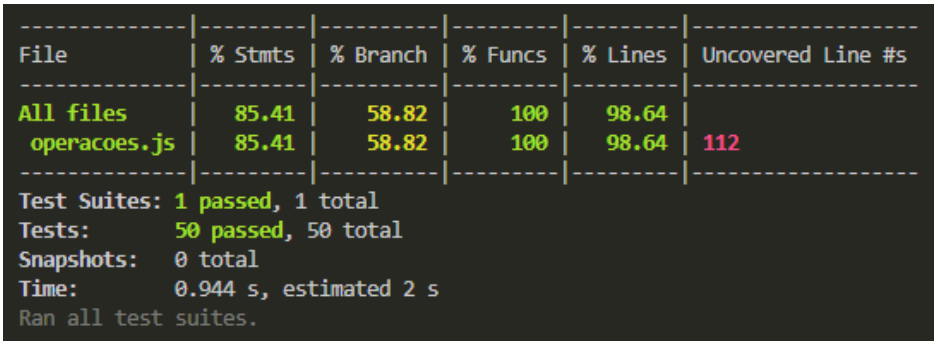
Belo Horizonte

2025

1. Análise Inicial

Cobertura de código inicial: Execução com o comando `npm test -- --coverage`, cobertura:

A cobertura de código mostra um resultado aparentemente bom, principalmente na cobertura de linhas e funções. No entanto, a cobertura de branches indica que nem todos os caminhos de execução estão sendo testados.



```
-------------|---------|----------|---------|---------|-------------------
File         | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------|---------|----------|---------|---------|-------------------
All files    |   85.41 |    58.82 |     100 |   98.64 |
 operacoes.js |   85.41 |    58.82 |     100 |   98.64 | 112
-------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       50 passed, 50 total
Snapshots:   0 total
Time:        0.944 s, estimated 2 s
Ran all test suites.
```

Pontuação de mutação: Execução com o comando `npx stryker run`:

- **Mutation Score:** 73.71%
- **Mutantes Mortos:** 154
- **Mutantes Sobreviventes:** 44
- **Timeouts:** 3
- **Total de Mutantes:** 213

Discrepância entre os Valores

A discrepância entre a cobertura de código (98.64% de linhas) e a pontuação de mutação (73.71%) é significativa . Esta diferença mostra a limitação da métrica de cobertura de código:

A cobertura de código mede apenas se o código foi executado, mas não mede se o comportamento foi corretamente verificado.

Exemplos da falha:

1. **Testes que executam código mas não verificam comportamento:** Um teste pode chamar uma função e obter um resultado, mas se não verificar o valor esperado corretamente, mutações sutis podem passar despercebidas.
2. **Falta de testes para casos de borda:** Muitas funções foram testadas apenas com casos "felizes", mas não testavam cenários de erro, valores limites ou comportamentos específicos.
3. **Asserções fracas:** Testes que apenas verificam que uma função não lançou exceção, mas não verificam o resultado correto.

Esta discrepância de aproximadamente 25 pontos percentuais (98.64% vs 73.71%) evidencia que a cobertura de código, sozinha, não é suficiente para garantir a qualidade de uma suíte de testes.

2. Análise de Mutantes Críticos

Foram selecionados 3 mutantes sobreviventes da primeira execução que mostram diferentes tipos de fraquezas nos testes originais:

a) Mutante 1: Função `isPar`-

Teste Original:

```
test('15. deve retornar true para um número par', () => {
  expect(isPar(100)).toBe(true);
});
```

O teste original apenas verificava que isPar(100) retorna true, o que é correto para o número 100. No entanto, quando a função é mutada para sempre retornar true, o teste continua passando porque:

O valor de entrada (100) é par, o valor esperado é true e o valor retornado pela função mutada também é true

O teste não verificava se a função poderia retornar false quando recebe um número ímpar, tornando impossível detectar essa mutação crítica.

Solução Implementada:

```
1  test('15a. deve retornar false para um número ímpar', () => { expect(isPar(7)).toBe(false); });
2    test('15b. deve retornar true para zero', () => { expect(isPar(0)).toBe(true); });
3    test('16a. deve retornar false para um número par', () => { expect(isImpar(100)).toBe(false); });
4    test('16b. deve retornar false para zero', () => { expect(isImpar(0)).toBe(false); });
```

b) Mutante 2: Função `Fatorial`

Teste Original:

```
test('8. deve calcular o fatorial de um número maior que 1', () => {
  expect(fatorial(4)).toBe(24);
});
```

O teste original testava apenas o fatorial de 4, que não ativa a condição `n === 0 || n === 1`. Quando a mutação muda || para &&, a condição n === 0 && n === 1 sempre será falsa (um número não pode ser 0 E 1 ao mesmo tempo), mas o teste ainda passa porque:

A entrada é 4, que não ativa a condição em ambos os casos; o código segue para o loop e calcula corretamente; o resultado esperado (24) é o mesmo

Os testes originais não verificavam explicitamente os casos de borda (n = 0 e n = 1), que são justamente os casos afetados por essa mutação.

Solução Implementada:

```
test('8a. deve calcular o fatorial de 0', () => { expect(fatorial(0)).toBe(1); });
 test('8b. deve calcular o fatorial de 1', () => { expect(fatorial(1)).toBe(1); });
 test('8d. deve garantir que fatorial de 2 retorna 2 (não 1)', () => { expect(fatorial(2)).toBe(2); });
 test('8c. deve lançar erro para número negativo no fatorial', () => {
   expect(() => fatorial(-1)).toThrow('Fatorial não é definido para números negativos.');
 });
```

c) Mutante 3: Função `clamp`

Teste Original:

```
test('36. deve manter um valor dentro de um intervalo (clamp)', () => {
  expect(clamp(5, 0, 10)).toBe(5);
});
```

O teste original usava valores onde valor (5) estava entre min (0) e max (10), não testando o caso onde valor é exatamente igual a min. Quando a mutação muda < para <=:

Para `valor = 5, min = 0`: `5 <= 0` é falso em ambos os casos, então o comportamento é o mesmo.

Para `valor = 0, min = 0`: `0 < 0` é falso, mas `0 <= 0` é verdadeiro - este caso não foi testado.

O teste não cobria o cenário crítico onde valor === min, permitindo que a mutação passe despercebida.

Solução Implementada:

```
test('deve retornar min quando valor é menor que min', () => { expect(clamp(-1, 0, 10)).toBe(0); });
  test('deve retornar max quando valor é maior que max', () => { expect(clamp(15, 0, 10)).toBe(10); });
  test('deve retornar o valor quando está no limite inferior', () => { expect(clamp(0, 0, 10)).toBe(0); });
  test('deve retornar o valor quando está no limite superior', () => { expect(clamp(10, 0, 10)).toBe(10); });
  test('deve garantir que clamp(1, 0, 10) retorna 1 e não 0', () => { expect(clamp(1, 0, 10)).toBe(1); });
  test('deve garantir que clamp(9, 0, 10) retorna 9 e não 10', () => { expect(clamp(9, 0, 10)).toBe(9); });
```

3. Resultados Finais

Após a implementação dos novos testes, a segunda execução do StrykerJS apresentou resultados significativamente melhores. A pontuação de mutação aumentou de **73.71%** para **96.71%**, uma melhoria de **23 pontos percentuais**. O trabalho realizado demonstrou a importância do teste de mutação como ferramenta complementar à análise de cobertura de código. A discrepância inicial entre a cobertura de código (98.64% de linhas) e a pontuação de mutação (73.71%) evidenciou que executar código não é suficiente - é necessário verificar se o comportamento está correto.

File / Directory	Mutation Score		Killed	Survived	Timeout	No coverage
	Of total	Of covered				
All files	96.71	96.71	203	7	3	0
operacoes.js	96.71	96.71	203	7	3	0