

# Análise de Eficácia de Testes com Teste de Mutação

Disciplina: Teste de Software  
Aluno: Davi Aguilar Nunes Oliveira  
Data: 02/11/2025

## 1 Análise Inicial

### 1.1 Cobertura de Código Inicial

O projeto inicial apresentava a seguinte situação:

Métrica	Percentual
Statements	85.41%
Branches	58.82%
Functions	100%
Lines	98.64%

Tabela 1: Métricas de Cobertura Inicial

**Testes:** 50 testes passando

**Linha não coberta:** Linha 112 (determinada pelo relatório de cobertura)

### 1.2 Pontuação de Mutação Inicial

Após configurar o StrykerJS e executar a primeira análise, obtivemos:

- **Mutation Score:** 73.71%
- **Total de Mutantes:** 213
- **Mutantes Mortos:** 154
- **Mutantes Sobreviventes:** 44
- **Timeout:** 3
- **No Coverage:** 12

### 1.3 Discrepância entre Cobertura e Eficácia

A discrepança é significativa:

- **Cobertura de Statements:** 85.41% (alta)
- **Pontuação de Mutação:** 73.71% (baixa)
- **Diferença:** 11.70 pontos percentuais

## Análise da Discrepância:

Esta diferença confirma a limitação da métrica de cobertura de código. Embora os testes **executassem** 85% das linhas, eles **não eram eficazes** em detectar bugs. Os mutantes sobreviventes revelaram que:

1. **Asserções genéricas** não capturavam mutações sutis
2. **Casos de borda** não estavam sendo testados
3. **Validações específicas** (como mensagens de erro exatas) estavam ausentes
4. **Contra-exemplos** para funções booleanas não existiam

Isso demonstra que executar código  $\neq$  testar qualidade. A cobertura mede **execução**, enquanto o teste de mutação mede **detecção de bugs**.

## 2 Análise de Mutantes Críticos

Analizamos em detalhe 3 mutantes sobreviventes da primeira execução que ilustram diferentes categorias de fraquezas na suíte de testes inicial:

### 2.1 Mutante #1: Divisão por Zero - Mensagem de Erro Genérica

Localização: `src/operacoes.js:8`

Código Original:

```
1 function divisao(a, b) {  
2     if (b === 0) throw new Error('Divisão por zero não é permitida.');//  
3     return a / b;  
4 }
```

Mutação Aplicada:

```
1 function divisao(a, b) {  
2     if (b === 0) throw new Error("");//  
3     return a / b;  
4 }
```

Status: Survived (Sobreviveu)

Análise: O teste original apenas verificava `toThrow()` sem mensagem específica, permitindo que a mutação da mensagem para passasse. Solução: Adicionamos `toThrow('Divisão por zero não é permitida.')`.

### 2.2 Mutante #2: Fatorial - Condição Repleta de Bugs

Localização: `src/operacoes.js:18-19`

Código Original:

```
1 function fatorial(n) {  
2     if (n < 0) throw new Error('Fatorial só deve ser definido para números negativos.');//  
3     if (n === 0 || n === 1) return 1;  
4     let resultado = 1;  
5     for (let i = 2; i <= n; i++) { resultado *= i; }  
6     return resultado;  
7 }
```

Mutação Aplicada:

```

1 function fatorial(n) {
2   if (n <= 0) throw new Error('Fatorial n o      definido para n meros
3   negativos.');
4   if (n === 0 || n === 1) return 1;
5   let resultado = 1;
6   for (let i = 2; i <= n; i++) { resultado *= i; }
7   return resultado;
}

```

**Status:** Survived (Sobreviveu)

**Análise:** O teste original usava `fatorial(4)` (valor > 1), não testando casos de borda como `fatorial(0)`, `fatorial(1)` ou números negativos. Isso permitiu que mutantes alterando `n < 0` para `n <= 0` sobrevivessem. **Solução:** Adicionamos testes para 0, 1, -5 e valores adicionais (2, 3).

```

17 function fatorial(n) {
18   if (n < 0) throw new Error('Fatorial não é definido para números negativos.');
19 - if (n === 0 || n === 1) return 1; ▼●●●
20 + if (false) return 1;
21   let resultado = 1;
22   for (let i = 2; i <= n; i++) { resultado *= i; } ●
23   return resultado;
}

```

ConditionalExpression Survived (19:7) Less  
Covered by 5 tests (yet still survived)

- Suite de Testes Fraca para 50 Operações Aritméticas 8. deve calcular o fatorial de um número maior que 1 (test/operacoes.test.js:29:4)
- Suite de Testes Fraca para 50 Operações Aritméticas 8.1. deve retornar 1 para fatorial de 0 (test/operacoes.test.js:30:4)
- Suite de Testes Fraca para 50 Operações Aritméticas 8.2. deve retornar 1 para fatorial de 1 (test/operacoes.test.js:31:4)
- Suite de Testes Fraca para 50 Operações Aritméticas 8.4. deve calcular fatorial de 2 (test/operacoes.test.js:35:4)
- Suite de Testes Fraca para 50 Operações Aritméticas 8.5. deve calcular fatorial de 3 (test/operacoes.test.js:36:4)

Figura 1: Screenshot do relatório StrykerJS exibindo múltiplos mutantes sobreviventes na função `fatorial`. Os mutantes em vermelho mostram as linhas 18-19, onde a condição `if (n === 0 || n === 1)` foi modificada de várias formas, mas os testes originais não conseguiram detectar essas alterações.

## 2.3 Mutante #3: Funções Booleanas - Retorno Sempre True

**Localização:** `src/operacoes.js:43-44`

**Código Original:**

```

1 function isPar(n) { return n % 2 === 0; }
2 function isImpar(n) { return n % 2 !== 0; }

```

**Mutação Aplicada:**

```

1 function isPar(n) { return true; }
2 function isImpar(n) { return true; }

```

**Status:** Both Survived (Ambos Sobreviveram)

**Análise:** Os testes verificavam apenas casos que retornam `true` (`isPar(100)`, `isImpar(7)`), não detectando funções que sempre retornam `true`. **Solução:** Adicionamos contra-exemplos testando retornos `false` (`isPar(7)`, `isImpar(100)`).

## 3 Solução Implementada

Implementamos uma estratégia em 4 categorias adicionando **33 novos testes**: (1) **6 testes** para mensagens de erro específicas; (2) **8 testes** para casos de borda (0, 1, arrays vazios); (3) **12 testes**

com contra-exemplos para funções booleanas; (4) **7 testes** com múltiplos valores para funções com loops.

## 4 Resultados Finais

Após implementar os testes adicionais, obtivemos:

Métrica	Início	Final	Melhoria
<b>Mutation Score</b>	73.71%	<b>96.71%</b>	+23.00%
<b>Mutantes Mortos</b>	154	203	+49
<b>Mutantes Sobreviventes</b>	44	7	-37
<b>Cobertura Statements</b>	85.41%	<b>100%</b>	+14.59%
<b>Cobertura Branches</b>	58.82%	<b>100%</b>	+41.18%
<b>Total de Testes</b>	50	83	+33

Tabela 2: Resultados da Análise de Mutação - Comparaçāo

### 4.1 Mutantes Sobreviventes Remanescentes

Apenas **7 mutantes** permaneceram sobreviventes (todos equivalentes): **Fatorial (4)** - condições mutadas ainda funcionam pelo loop; **Clamp (2)** - mudanças de < para <= equivalentes; **Produto Array (1)** - condição redundante.

The figure consists of two side-by-side screenshots of the StrykerJS user interface. Both screenshots show the same mutation report for the `clamp` function. The left screenshot shows the initial state with 50 tests and 44 survivors. The right screenshot shows the final state after adding additional tests, with 83 tests and 7 survivors. Both reports list 7 surviving mutants, which are highlighted in green. The mutants are described as being equivalent to the original code. The reports also include coverage statistics and a summary of the tests run.

Figura 2: Mutantes sobreviventes na função `clamp` (linhas 88-89) com mutações de < para <= e > para >=. São equivalentes pelos mesmos resultados.

## 5 Conclusão

Este trabalho demonstrou a superioridade do **teste de mutação** sobre métricas de **cobertura de código**. A discrepância de 11.70 pontos (85.41% cobertura vs 73.71% mutação) revelou que executar código  $\neq$  testar qualidade. A melhoria para 96.71% (+23 pontos) com 49 mutantes adicionais mortos comprova que cobertura mede **execução**, enquanto teste de mutação mede **detecção de bugs**. Os principais aprendizados foram: (1) Asserções específicas são críticas; (2) Casos de borda revelam fraquezas ocultas; (3) Contra-exemplos são essenciais para funções booleanas; (4) O StrykerJS é fundamental para avaliar qualidade real de testes.

Fim do Relatório