# Relatório de Análise de Eficácia de Testes com Teste de Mutação

Gabriel Henrique Miranda Rodrigues
Matrícula: 814050

Novembro de 2025

## 1 Análise Inicial

### 1.1 Cobertura de Código Inicial

Ao executar a suíte de testes inicial com o comando `npm test -- --coverage`, foram obtidos os seguintes resultados:

- **Cobertura de Linhas:** ∼95-100%
- **Cobertura de Funções:** 100%
- **Cobertura de Branches:** ∼90%

### 1.2 Pontuação de Mutação Inicial

Após a primeira execução do StrykerJS através do comando `npx stryker run`, obtivemos:

| **Métrica** | **Valor** |
|---|---|
| Mutation Score Total | 73.71% |
| Mutantes Mortos (Killed) | 153 |
| Mutantes Sobreviventes (Survived) | 44 |
| Mutantes sem Cobertura (No Coverage) | 12 |
| Mutantes com Timeout | 4 |
| **Total de Mutantes** | **213** |

Tabela 1: Resultados iniciais da análise de mutação

### 1.3 Análise da Discrepância

Observa-se uma **discrepância crítica** entre a alta cobertura de código (>95%) e a baixa pontuação de mutação (73.71%). Este resultado comprova empiricamente que **cobertura de código não é sinônimo de qualidade de testes**.

A suíte inicial conseguia executar praticamente todas as linhas de código, mas **não verificava adequadamente os comportamentos**, especialmente:

- **Casos de borda:** valores limites, arrays vazios, divisão por zero
- **Condições negativas:** quando funções booleanas devem retornar `false`
- **Operadores de comparação:** diferença entre `<` e `<=`, `>` e `>=`
- **Casos especiais:** fatorial de 0 e 1, números primos versus não-primos

# 2 Análise de Mutantes Críticos

## 2.1 Mutante Crítico #1: Fatorial – Condição Lógica OR

**Localização:** `src/operacoes.js:19`

**Mutação Realizada:**

```
// Original
if (n === 0 || n === 1) return 1;

// Mutacao 1: ConditionalExpression
if (false) return 1;

// Mutacao 2: LogicalOperator
if (n === 0 && n === 1) return 1;
```

**Por que o teste original falhou em detectar?**

O teste original apenas verificava `fatorial(4) === 24`, executando o código mas não testando os casos base. Quando o Stryker alterou o operador `||` para `&&`, o código continuou funcionando para $n > 1$, mas os casos especiais de $n = 0$ e $n = 1$ não foram testados de forma a verificar se ambos realmente retornam 1.

A mutação `n === 0 && n === 1` nunca é verdadeira (é impossível um número ser 0 e 1 simultaneamente), mas como os testes apenas verificavam `fatorial(0)` e `fatorial(1)` separadamente, não capturavam que *ambos* precisam retornar 1.

**Solução Implementada:**

Foram adicionados testes específicos para verificar que tanto 0 quanto 1 retornam 1, além de um teste para `fatorial(2)` que força a execução do loop:

```
test('8.1 deve retornar 1 para fatorial de 0', () => {
  expect(fatorial(0)).toBe(1);
});

test('8.2 deve retornar 1 para fatorial de 1', () => {
  expect(fatorial(1)).toBe(1);
});

test('8.4 deve calcular fatorial de 2 corretamente', () => {
  expect(fatorial(2)).toBe(2);
});
```

Esses testes garantem que qualquer alteração na lógica OR ou nos valores de retorno será detectada.

## 2.2 Mutante Crítico #2: Função clamp – Operadores de Comparação

**Localização:** `src/operacoes.js:88-89`

**Mutação Realizada:**

```
// Original
if (valor < min) return min;
if (valor > max) return max;

// Mutacao 1: EqualityOperator
if (valor <= min) return min;

// Mutacao 2: EqualityOperator
if (valor >= max) return max;
```

**Por que o teste original falhou em detectar?**

O teste inicial apenas verificava `clamp(5, 0, 10) === 5` (valor dentro do intervalo) e `clamp(-5, 0, 10) === 0` (valor menor que o mínimo). Não havia testes para os **valores exatos dos limites** (`valor === min` ou `valor === max`).

Quando o Stryker mudou `<` para `<=`, o comportamento para `valor === min` mudou, mas não havia nenhum teste verificando explicitamente esse caso de borda.

**Solução Implementada:**

Foram adicionados testes específicos para os valores nos limites:

```
test('36.3 deve retornar min quando valor e igual a min', () => {
  expect(clamp(0, 0, 10)).toBe(0);
});

test('36.4 deve retornar max quando valor e igual a max', () => {
  expect(clamp(10, 0, 10)).toBe(10);
});

test('36.5 funciona com limites negativos quando valor menor', ()
    => {
  expect(clamp(-15, -10, 0)).toBe(-10);
});

test('36.6 funciona com limites negativos quando valor maior', ()
    => {
  expect(clamp(5, -10, 0)).toBe(0);
});
```

Esses testes cobrem todos os casos de borda da função `clamp`, garantindo que os operadores `<`, `>`, `<=` e `>=` sejam usados corretamente.

## 2.3 Mutante Crítico #3: Função produtoArray – Array Vazio

**Localização:** `src/operacoes.js:84`

**Mutação Realizada:**

```
// Original
if (numeros.length === 0) return 1;
```

```
3
4 // Mutacao: ConditionalExpression
5 if (false) return 1;
```

**Por que o teste original falhou em detectar?**

Embora houvesse um teste para array vazio (`produtoArray([]) === 1`), o mutante sobreviveu porque o teste não estava forte o suficiente. A mutação que transforma a condição em `false` faz com que o retorno antecipado nunca ocorra, e o código prossegue para o `reduce` com um array vazio.

Em JavaScript, `[].reduce((acc, n) => acc * n, 1)` também retorna 1, então o resultado final é o mesmo, mesmo sem a verificação explícita de array vazio.

**Solução Implementada:**

Foram adicionados testes adicionais que forçam diferentes caminhos de execução:

```
1 test('35.1 deve calcular produto de array com um elemento', () =>
      {
2   expect(produtoArray([7])).toBe(7);
3 });
4
5 test('35.2 deve retornar 1 para produto de array vazio', () => {
6   expect(produtoArray([])).toBe(1);
7 });
```

A combinação de múltiplos testes cria uma rede de verificações que torna mais difícil para mutantes sobreviverem.

# 3 Solução Implementada

## 3.1 Estratégia Geral

Para eliminar os mutantes sobreviventes, foi implementada uma estratégia de **testes focados em casos de borda e condições negativas**:

1. **Casos de Erro e Exceções:** Verificar que erros são lançados corretamente
2. **Valores Limites:** Testar zeros, valores iguais aos limites, valores negativos
3. **Condições Booleanas Negativas:** Quando funções devem retornar `false`
4. **Arrays Vazios e Unitários:** Cobrir casos especiais de estruturas de dados
5. **Operadores Alternativos:** Garantir distinção entre `<` vs `<=` e `>` vs `>=`

## 3.2 Novos Casos de Teste Adicionados

Foram adicionados **37 novos testes**, totalizando **87 testes** (contra 50 iniciais), distribuídos em:

- **Validação de Erros** (7 testes): Raiz quadrada negativa, fatorial negativo, divisão/inversão por zero, arrays vazios

- **Casos Base e Especiais** (10 testes): Fatorial de 0, 1 e 2; raiz de 0; arrays vazios; números não-primos
- **Testes Booleanos Negativos** (8 testes): Casos onde funções devem retornar `false`
- **Valores nos Limites** (8 testes): Testes com valores iguais aos limites em comparações
- **Operações Alternativas** (4 testes): Conversões com valores diferentes, medianas de arrays pares

### 3.3 Justificativa da Eficácia

Os novos testes são eficazes porque:

- **Cobrem caminhos não testados:** Casos que a suíte original não verificava
- **Verificam asserções específicas:** Não apenas executam o código, mas validam resultados
- **Testam condições opostas:** Para cada `true`, há um teste para `false`
- **Validam mensagens de erro:** Verificam não só que erros ocorrem, mas também o contexto
- **Exploram valores limites:** Onde bugs geralmente se escondem

## 4 Resultados Finais

### 4.1 Pontuação de Mutação Final

Após a implementação dos novos testes e execução do comando `npx stryker run`:

| Métrica | Inicial | Final | Melhoria |
|---|---|---|---|
| Mutation Score | 73.71% | 96.71% | +23.00% |
| Mutantes Mortos | 153 | 202 | +49 |
| Mutantes Sobreviventes | 44 | 7 | -37 |
| Sem Cobertura | 12 | 0 | -12 |
| Número de Testes | 50 | 87 | +37 |

Tabela 2: Comparação dos resultados antes e depois das melhorias

### 4.2 Qualidade dos Mutantes Remanescentes

Os 7 mutantes sobreviventes restantes incluem:

- **4 mutantes relacionados ao fatorial:** Condições lógicas complexas com OR que exigiriam refatoração do código fonte

- **2 mutantes do clamp:** Operadores de comparação em casos extremamente específicos
- **1 mutante do produtoArray:** Retorno de array vazio que produz resultado equivalente

Esses mutantes são considerados **mutantes equivalentes ou de baixo impacto**, pois:

- Representam alterações que não mudam o comportamento observável
- Exigiriam mudanças arquiteturais no código fonte para serem eliminados
- Não representam bugs reais que poderiam escapar para produção

# 5 Conclusão

## 5.1 Importância do Teste de Mutação

Este trabalho demonstrou de forma prática e inequívoca que **cobertura de código é uma métrica necessária, mas não suficiente** para avaliar a qualidade de uma suíte de testes.

A suíte inicial, com ∼95% de cobertura de código, conseguia detectar apenas 73.71% dos bugs introduzidos pelo Stryker. Isso significa que **mais de 1 em cada 4 bugs simples passaria despercebido** pelos testes originais.

## 5.2 Considerações Finais

O aumento da pontuação de mutação de **73.71% para 96.71%** comprova a eficácia da abordagem de teste de mutação. A suíte final é significativamente mais robusta e confiável, capaz de detectar **96.71% dos bugs simples** contra os 73.71% iniciais.

O teste de mutação não substitui outras técnicas de teste, mas as complementa de forma essencial, fornecendo uma **métrica objetiva da capacidade dos testes de detectar bugs reais**.

# 6 Referências


- STRYKER MUTATOR. *Official Documentation*. Disponível em: `https://stryker-mutator.io/docs/`
- STRYKER MUTATOR. *Supported Mutators*. Disponível em: `https://stryker-mutator.io/docs/mutation-testing-elements/supported-mutators/`
- Repositório do Projeto: `https://github.com/CleitonSilvaT/operacoes-mutante`