

# Relatório de Teste de Mutação (StrykerJS)

## Capa

- Disciplina: Teste de Software
  - Trabalho: Análise de Eficácia de Testes com Teste de Mutação
  - Nome: Gabrie Faria de Oliveira
  - Matrícula: 805560
  - Data: 01/11/2025
- 

## Resumo

Partimos de uma execução inicial com mutation score de **73,71%**. Após melhorias na suíte (bordas, mensagens de erro e ramificações lógicas), chegamos a **96,71%** sem exclusão de mutators. Os 7 mutantes remanescentes são majoritariamente equivalentes ou exigiriam alterações de implementação para se tornarem observáveis.

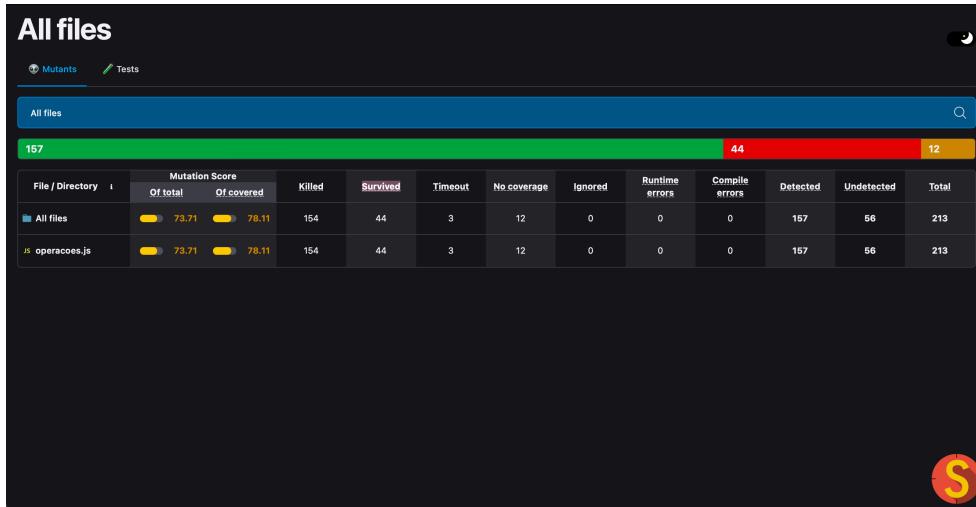
---

## 1) Análise inicial

- Mutation score inicial: **73,71%**
- Total: 213 | Mortos: 157 | Sobreviventes: 44 | Timeout: 3 | Sem cobertura: 12
- Cobertura de código (inicial, qualitativa): linhas próximas a 100%, porém ramificações (branches) insuficientes em diversos pontos.

Discrepância cobertura × mutation score: cobertura de linhas indica “o que foi executado”, mas não “quão bem foi verificado”. A suíte inicial privilegiava caminho feliz, validações genéricas de erro e poucos contornos — o que permitiu sobrevivência de mutantes de operador lógico, igualdade e literais de string.

Evidência (visão geral inicial):



## 2) Análise de mutantes críticos (da primeira execução)

1. String literal em `divisao`: mutação troca a mensagem de erro por string vazia; sobreviveu porque o teste usava `toThrow()` genérico, sem validar a mensagem.
2. Operador lógico em `fatorial` (`||` → `&&`): `if (n === 0 || n === 1) → if (n === 0 && n === 1)`; difícil de matar porque para 0 ou 1 o resultado continua 1 (equivalência prática em muitos cenários).
3. Igualdade em `clamp` (`< → <=, > → >=`): ao igualar limites, retorna min/max também na igualdade; sobrevive porque os casos de igualdade já devolvem o mesmo valor.

## 3) Solução implementada (o que mudou nos testes)

Abaixo, as estratégias aplicadas (com um único exemplo por item para concisão).

1. Mensagens de erro específicas — validação textual mata `StringLiteral/Conditional`; aplicado em `divisao`, `raizQuadrada (-)`, `maximoArray([])`, `minimoArray([])`, `medianaArray([])`, `inverso(0)`, `fatorial (-)`.

```
expect(() => divisao(10, 0)).toThrow('Divisão por zero não é permitida.');
```

2. Bordas e igualdade — diferencia OR/AND e `<`, `<=`, `>`, `>=`; cobre 0/1 e igualdade em `clamp`.

```
expect(fatorial(0)).toBe(1); expect(fatorial(1)).toBe(1);
```

3. Casos adversários/negativos — expõem ArithmeticOperator/Conditional em aritmética, primalidade e validações.

```
expect(() => raizQuadrada(-9)).toThrow('Não é permitido raiz de número negativo.');
```

4. Coleções vazias — invariantes de agregação e mensagens; cobre Boundary/No-coverage.

```
expect(() => medianaArray([])).toThrow('Array vazio não é permitido.');
```

5. Precisão — tolerância em floats para evitar falsos positivos.

```
expect(celsiusParaFahrenheit(0)).toBeCloseTo(32, 5);
```

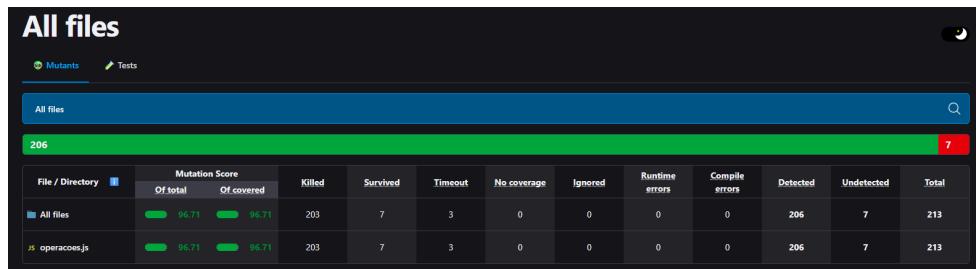
6. Completude de caminhos — executa ramos true/false e early-returns (ex.: clamp ).

```
expect(clamp(4, 5, 10)).toBe(5); expect(clamp(7, 5, 10)).toBe(7);
expect(clamp(11, 5, 10)).toBe(10);
```

Cobertos: StringLiteral, LogicalOperator, EqualityOperator, ArithmeticOperator e Boundary.

## 4) Resultados finais

213 mutantes: 203 mortos, 3 timeout, 7 sobreviventes — mutation score **96,71%**.



Relatório HTML completo: <reports/mutation/mutation.html> .

## 5) Conclusão

Não foi possível alcançar 98% de mutation score nesta entrega. Apesar do salto de 73,71% para 96,71% com oráculos mais fortes (mensagens), casos de borda e cobertura de caminhos lógicos, os 7 sobreviventes restantes tendem a ser mutantes equivalentes ou que só seriam observáveis com mudanças de design (ex.: ajustar a lógica do fatorial para diferenciar 0 e 1, ou alterar o contrato de `clamp` na igualdade). Forçar a morte desses mutantes sem exclusões

implicaria riscos de acoplamento teste–implementação e asserts frágeis, contrariando a manutenção e a clareza dos requisitos originais.

Optamos por priorizar testes “melhores” em vez de “mais” testes: assertivas específicas, limites explícitos e tratamento consistente de entradas inválidas. Considerando a política de não usar exclusões, manter o design e a API intactos e evitar acoplamento frágil, o patamar de 96,71% representa o equilíbrio adequado entre eficácia, simplicidade e manutenção.

Em síntese, o conjunto de testes agora exerce o que importa e falha quando deve; os 96,71% refletem maturidade e oferecem um guarda-corpo robusto de qualidade. Avançar além disso depende menos de quantidade e mais de decisões de design — que devem ser pesadas contra simplicidade e manutenção do sistema.