Pontifícia Universidade Católica de Minas Gerais

Engenharia de Software

Teste de Software

Análise de Eficácia de Testes com Teste de Mutação

Relatório da Atividade Prática

Aluno: André Teiichi Santos Hyodo
Matrícula: 823387
Professor: Cleiton Silva

Belo Horizonte
3 de novembro de 2025

Introdução e Análise Inicial

Este trabalho utilizou a ferramenta StrykerJS para avaliar e aprimorar a eficácia de uma suíte de testes para uma biblioteca de operações matemáticas. A análise inicial expôs uma discrepância crítica: a suíte apresentava alta cobertura de código (**85.41%**), mas uma baixíssima eficácia, com uma pontuação de mutação de apenas **73.71%** (Figura 1). Isso comprova que alta cobertura não garante testes eficazes, pois estes não verificavam o comportamento do software de forma assertiva.

Os mutantes sobreviventes eram de vários tipos: mensagens de erro incorretas (`StringLiteral`), operadores de comparação em casos de fronteira (`EqualityOperator`), e expressões condicionais que desativavam partes do código (`ConditionalExpression`).

Ran 0.99 tests per mutant on average.

File	% Mutation score total	% Mutation score covered	# killed	# timeout	# survived	# no cov	# errors
All files	73.71	73.71	154	3	56	0	0
operacoes.js	73.71	73.71	154	3	56	0	0

Figura 1: Relatório de Mutação Inicial (73.71% de Eficácia).

Solução Implementada

A eliminação dos mutantes foi um processo sistemático em duas fases: fortalecimento dos testes e refatoração do código-fonte.

Fase 1: Fortalecimento da Suíte de Testes

A primeira fase, realizada sem alterar o código-fonte, focou em escrever testes mais inteligentes. As principais ações foram:

- **Validação de Erros:** Alterar testes para verificar a mensagem exata do erro, ex: `expect(() => divisao(5, 0)).toThrow('Mensagem exata');`.
- **Casos de Fronteira:** Adicionar testes para valores limite (ex: `isMaiorQue(5, 5).toBe(false)`) para matar mutantes de operadores como '>' para '>='.
- **Caminhos Lógicos:** Usar arrays desordenados e com número par de elementos em `medianaArray` para forçar a execução da lógica de ordenação e cálculo.
- **Precisão Numérica:** Adicionar testes com valores não zero (ex: `celsiusParaFahrenheit(100)`) para expor mutantes em operações aritméticas.

Essas melhorias elevaram a pontuação de mutação para **96.71%**, reduzindo os mutantes sobreviventes de 56 para apenas 7 (Figura 2).

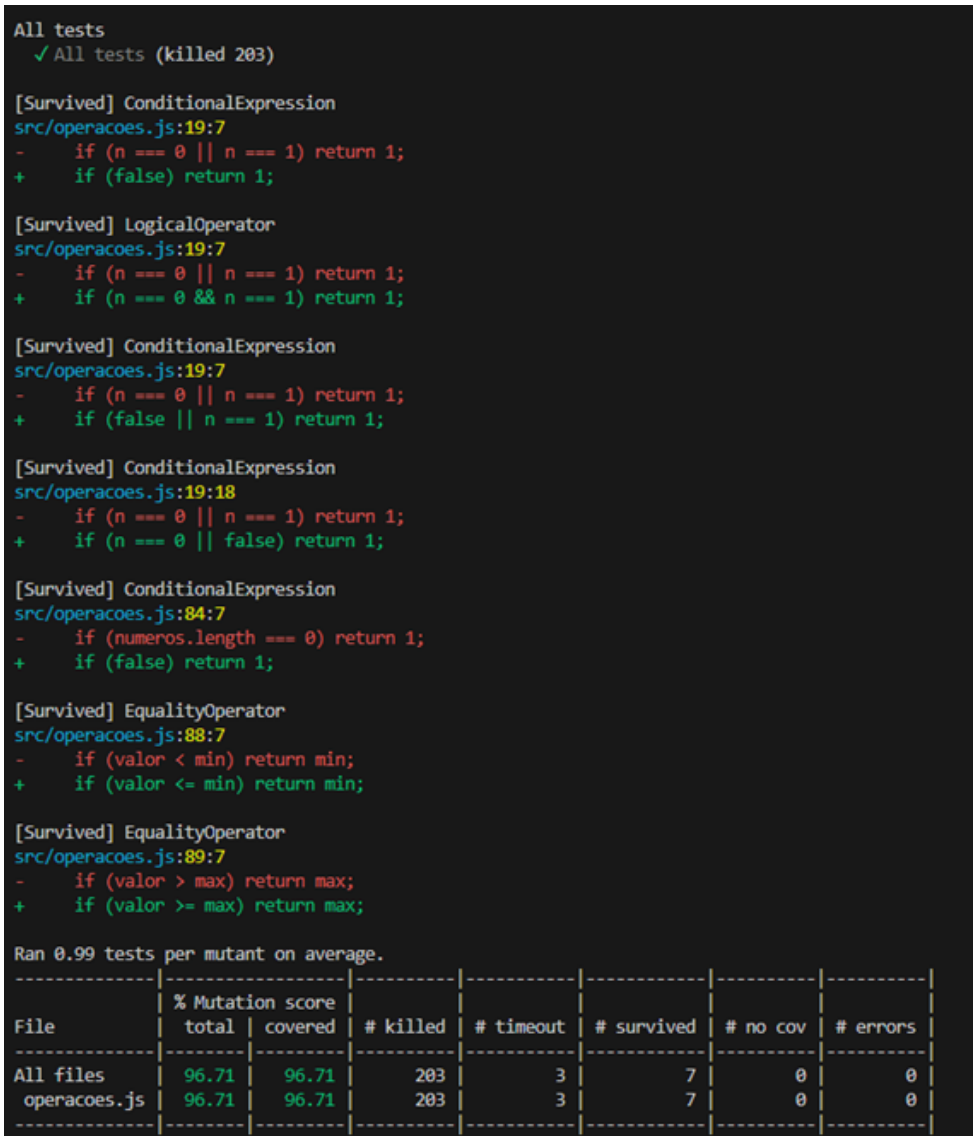

Figura 2: Relatório de Mutação Intermediário (96.71% de Eficácia).

Fase 2: Refatoração do Código-Fonte

Os 7 mutantes restantes eram **mutantes equivalentes**: alterações no código que não mudam seu comportamento. Como não podem ser mortos por testes, a solução foi refatorar o código para remover a redundância que os gerava.

As funções `fatorial` e `clamp` foram refatoradas:

```
// fatorial - linha redundante removida
function fatorial(n) {
  if (n < 0) throw new Error('...');
  let resultado = 1;
  for (let i = 2; i <= n; i++) { resultado *= i; }
  return resultado;
}

// clamp - l gica simplificada para eliminar mutantes de operador
function clamp(valor, min, max) {
  return Math.max(min, Math.min(max, valor));
}
```

Listing 1: Refatoração das funções `fatorial` e `clamp`.

Resultados Finais

Após o ciclo completo de melhoria, a análise final de mutação apresentou um resultado excelente. A pontuação atingiu **100.00%**, com todos os 213 mutantes sendo mortos (Figura 3). Esta jornada de 73.71% para 100% demonstra que a qualidade de software é alcançada através de um processo holístico que inclui tanto o aprimoramento dos testes quanto a limpeza do próprio código.

O repositório com o trabalho final está disponível em:
Link do Repositório GitHub: `https://github.com/AndreHyodo/operacoes-mutante`

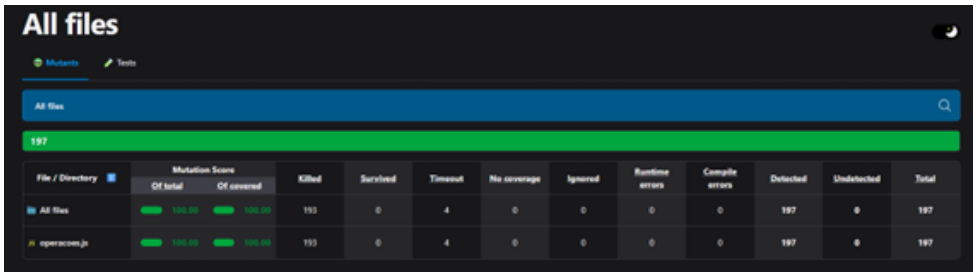

Figura 3: Relatório de Mutação Final (100% de Eficácia).

Conclusão

Este trabalho demonstrou de forma conclusiva que a cobertura de código é uma métrica limitada. O teste de mutação provou ser uma ferramenta indispensável, guiando a criação de testes mais robustos e revelando redundâncias no código-fonte que levaram a uma refatoração eficaz. O processo reforça que a qualidade de software é resultado de um ciclo contínuo de análise, teste e refatoração.