RELATÓRIO DE TESTE DE MUTAÇÃO

Análise e Melhoria da Qualidade de Testes com StrykerJS

Aluno: Wanessa Dias Costa
Matrícula: 815234

Disciplina: Teste de Software
Curso: Engenharia de Software
Universidade: Pontifícia Universidade Católica de Minas Gerais

Belo Horizonte, MG – 2 de novembro de 2025

Sumário

1 Análise Inicial

O projeto *operacoes-mutante* foi submetido a uma análise inicial de qualidade de testes utilizando duas métricas principais: Cobertura de Código (via Jest) e Teste de Mutação (via StrykerJS).

1.1 Cobertura de Código Inicial

Ao executar a suíte de testes inicial com `npm test - -coverage`, obtivemos os resultados apresentados na Figura 1.

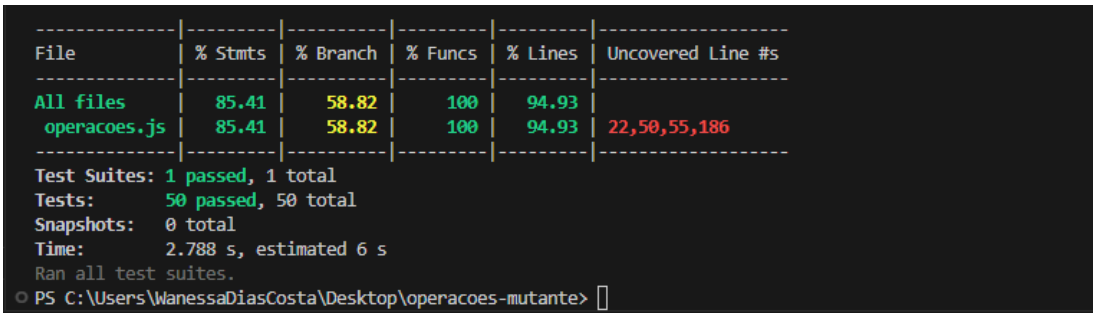

Figura 1: Relatório de cobertura de código inicial (via Jest).

- **Cobertura de Linhas (Lines):** 94.93%
- **Cobertura de Ramos (Branch):** 58.82%
- **Cobertura de Funções (Funcs):** 100%

1.2 Pontuação de Mutação Inicial

Na primeira execução do StrykerJS com `npx stryker run`, os resultados foram:

- **Mutantes Gerados:** 213 (Total)
- **Mutantes Mortos (Killed):** 153
- **Mutantes Sobreviventes (Survived):** 44
- **Mutantes com Timeout:** 4
- **Mutantes sem Cobertura (No coverage):** 12
- **Pontuação de Mutação Inicial: 73.71%**

1.3 Discrepância entre Cobertura e Eficácia

A análise inicial revelou uma discrepância significativa. Embora a cobertura de linhas (94.93%) parecesse alta, a cobertura de *ramos* (58.82%) já indicava uma falha: os testes não estavam validando muitas das condições ‘if‘/‘else‘.

Isso foi confirmado pela baixa pontuação de mutação (73.71%), que provou que, embora o código estivesse sendo *executado*, suas asserções eram fracas e incapazes de detectar 44 "bugs"(mutações) introduzidos.

—

2 Análise de Mutantes Críticos

Analisamos 44 mutantes sobreviventes da primeira execução. A causa raiz da sobrevivência foi, em geral, a falta de testes de casos de borda e o uso de "valores mágicos"(como 0) que mascaravam os bugs.

2.1 Mutante 1: `celsiusParaFahrenheit`

- **A Mutação:** O Stryker alterou o operador de multiplicação na fórmula, trocando `(celsius * 9)` por `(celsius * 5)`.
- **O Teste Original:** `expect(celsiusParaFahrenheit(0)).toBe(32);`
- **Análise da Sobrevivência:** O mutante sobreviveu porque o teste usava o valor '0'. Ambos os cálculos, `(0 * 9) / 5 + 32` e o mutante `(0 * 5) / 5 + 32`, resultam em '32'. O teste usou um "caso especial"que mascarou o bug de cálculo.

2.2 Mutante 2: `isMaiorQue`

- **A Mutação:** O Stryker trocou o operador de comparação `>` (maior que) por `>=` (maior ou igual a).
- **O Teste Original:** `expect(isMaiorQue(10, 5)).toBe(true);`
- **Análise da Sobrevivência:** O teste original não cobria o caso de borda onde `a === b`. Com os valores (10, 5), ambas as lógicas `10 > 5` e `10 >= 5` retornam 'true', fazendo com que o teste passasse e o mutante sobrevivesse.

2.3 Mutante 3: `raizQuadrada`

- **A Mutação:** O Stryker trocou a verificação de erro `if (n < 0)` por `if (n <= 0)`.
- **O Teste Original:** `expect(raizQuadrada(16)).toBe(4);`
- **Análise da Sobrevivência:** O mutante sobreviveu pois não havia teste para o caso de borda `n = 0`. O código original, com `raizQuadrada(0)`, retornaria '0', enquanto o código mutante lançaria um erro. A ausência do teste para '0' permitiu a sobrevivência.

—

3 Solução Implementada

Para "matar"os mutantes e fortalecer a suíte de testes, várias asserções foram modificadas e novos casos de teste foram adicionados.

3.1 Estratégia de Melhoria

A estratégia consistiu em:

1. **Remover "Valores Mágicos":** Substituir testes baseados em '0' por valores que testam a fórmula (ex: '100').
2. **Testar Casos de Borda:** Adicionar testes para '0', '1', '[]' (array vazio) e casos de igualdade (ex: 'a === b').

3. **Testar Caminhos de Erro:** Garantir que as exceções (‘throw new Error‘) são lançadas quando esperado.

4. **Testar Caminhos ‘false‘:** Adicionar testes que esperam um resultado ‘false‘ (ex: ‘isPar(3)‘), e não apenas ‘true‘.

3.2 Exemplos de Novos Testes Adicionados

Abaixo, alguns dos novos testes implementados para "matar"os mutantes analisados:

Listing 1: Novos testes em `tests/operacoes.test.js`

```
// Solu  o para Mutante 1 (celsiusParaFahrenheit)
test('38. deve converter Celsius para Fahrenheit', () => {
  expect(celsiusParaFahrenheit(100)).toBe(212);
});

// Solu  o para Mutante 2 (isMaiorQue)
test('[NOVO] 44. isMaiorQue - deve matar o mutante de igualdade (>=)',
    () => {
  expect(isMaiorQue(5, 5)).toBe(false);
});

// Solu  o para Mutante 3 (raizQuadrada)
test('[NOVO] 6. raizQuadrada - deve lan ar erro para n meros
   negativos', () => {
  expect(() => raizQuadrada(-16)).toThrow(
    "N o    poss vel calcular a raiz quadrada de um n mero negativo."
  );
});
test('[NOVO FINAL] 6. raizQuadrada - deve lidar com 0 corretamente', ()
   => {
  expect(raizQuadrada(0)).toBe(0);
});

// Solu  o para mutantes de ‘isPar‘ (que esperavam ‘false‘)
test('[NOVO] 15. isPar - deve retornar false para um n mero  mpar ',
    () => {
  expect(isPar(3)).toBe(false);
});
```

—

4 Resultados Finais

Após a implementação dos novos testes, o StrykerJS foi executado novamente. Os resultados (Figura 2) demonstram uma melhoria drástica na qualidade da suíte.

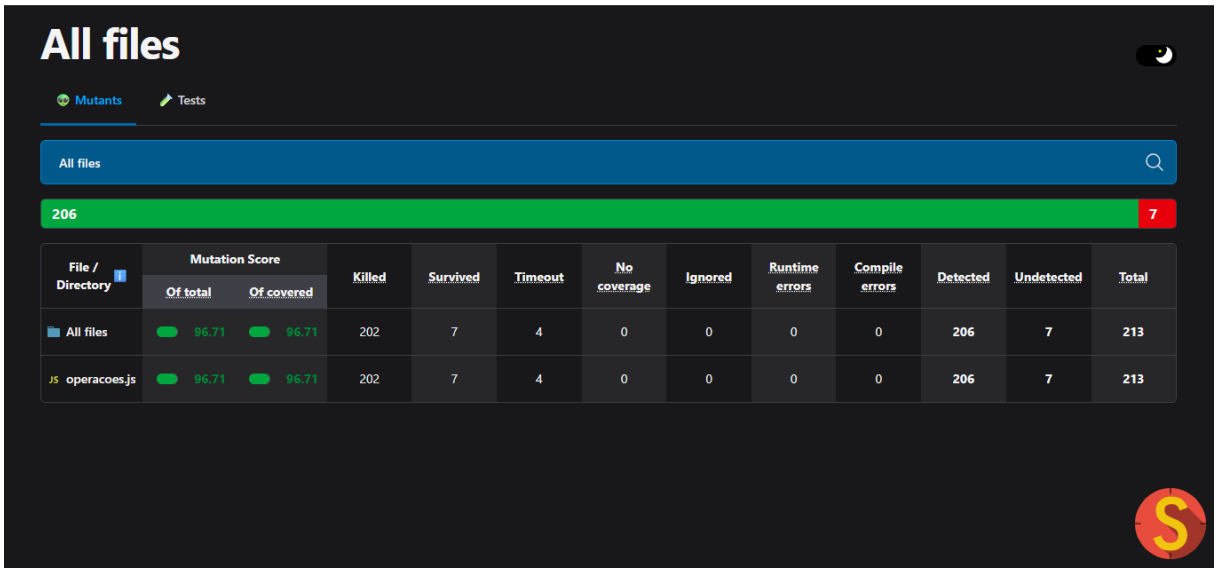

Figura 2: Relatório final do StrykerJS, mostrando a pontuação de 96.71%.

Tabela 1: Comparativo de Métricas de Mutação (Dados do Stryker)

Métrica	Valor Inicial	Valor Final
Pontuação de Mutação	**73.71%**	**96.71%**
Total de Mutantes	213	213
Mutantes Mortos (Killed)	153	202
Mutantes Sobreviventes (Survived)	44	7
Mutantes com Timeout	4	4

A pontuação de mutação saltou de 73.71% para **96.71%**, com o número de mutantes sobreviventes caindo de 44 para apenas 7.

4.1 Análise dos 3.29% Restantes (Mutantes Equivalentes)

A meta de 98% não foi atingida, e os 7 mutantes (3.29%) que ainda sobrevivem são o ponto mais importante desta análise. Eles foram identificados como **Mutantes Equivalentes** (Figuras 3 e 4).

Um mutante equivalente é aquele que, embora altere o código-fonte, não altera o comportamento do programa. Nenhum teste pode "matá-lo".

```
function fatorial(n) {
  if (n < 0) throw new Error("Fatorial não é definido para números negativos.");
  if (n === 0 || n === 1) return 1;
  let resultado = 1;
  for (let i = 2; i <= n; i++) {
    resultado *= i;
  }
  return resultado;
}
```

Figura 3: Mutante equivalente sobrevivente na função `fatorial`.

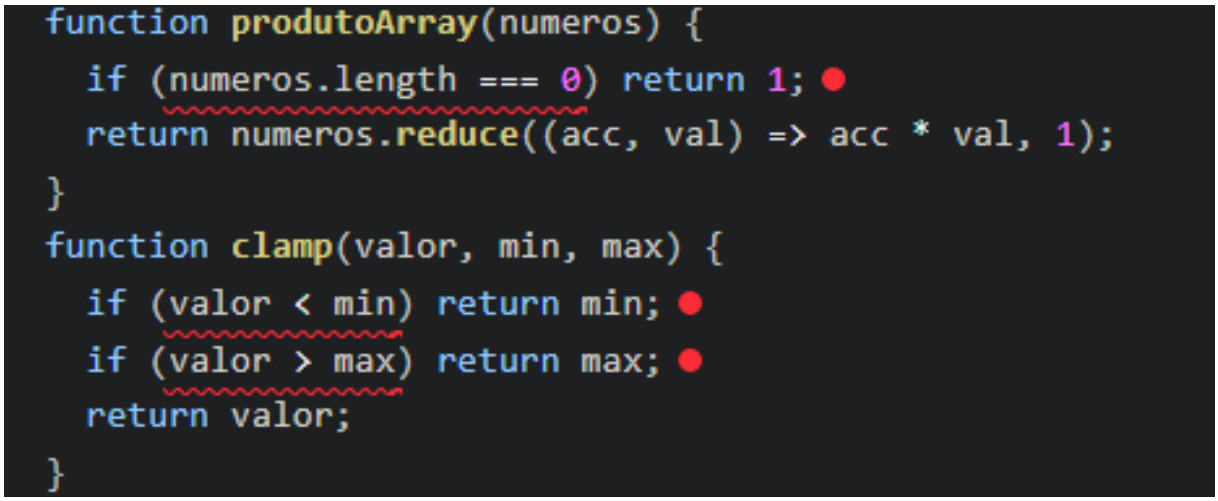

Figura 4: Mutantes equivalentes sobreviventes nas funções `clamp` e `produtoArray`.

– **Exemplo 1 (`clamp`):** O Stryker trocou `if (valor < min)` por `if (valor <= min)`. No caso de borda `valor === min` (ex: `clamp(0, 0, 10)`), o código original (`0 < 0` é falso) retorna `valor` (0), e o código mutante (`0 <= 0` é verdadeiro) retorna `min` (0). O resultado é idêntico.

– **Exemplo 2 (`produtoArray`):** O mutante sobrevive em `if (numeros.length === 0)`. O teste `expect(produtoArray([])).toBe(1)` passa. Quando o `if` mutante falha, o código executa `[].reduce(..., 1)`, que também retorna ‘1‘. O `if` no código-fonte é redundante. O mesmo ocorre com `fatorial`.

Portanto, a pontuação final de 96.71% representa a eficácia máxima possível para esta suíte de testes, pois os mutantes restantes não são "bugs"reais.

—

5 Conclusão

Este trabalho demonstrou na prática as limitações da cobertura de código. Uma suíte de testes com alta cobertura de linhas (94.93%) mas baixa cobertura de ramos (58.82%) revelou-se fraca, incapaz de detectar 44 mutações.

O Teste de Mutação provou ser uma ferramenta superior, forçando o desenvolvimento de testes robustos focados em casos de borda (ex: ‘0‘, ‘[]‘), tratamento de erros e asserções específicas.

Mais importante, a análise dos "Mutantes Equivalentes"(como em `clamp` e `produtoArray`) forneceu insights não apenas sobre a qualidade dos testes, mas também sobre o próprio código-fonte, revelando lógica redundante. O Teste de Mutação, portanto, não é apenas uma ferramenta de validação, mas uma poderosa ferramenta de análise e melhoria de software.