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 discrepância 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

Analisamos 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:

```
function divisao(a, b) {
  if (b === 0) throw new Error('Divis o por zero  n  o    permitida.');
  return a / b;
}
```

Mutação Aplicada:

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

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:

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

Mutação Aplicada:

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

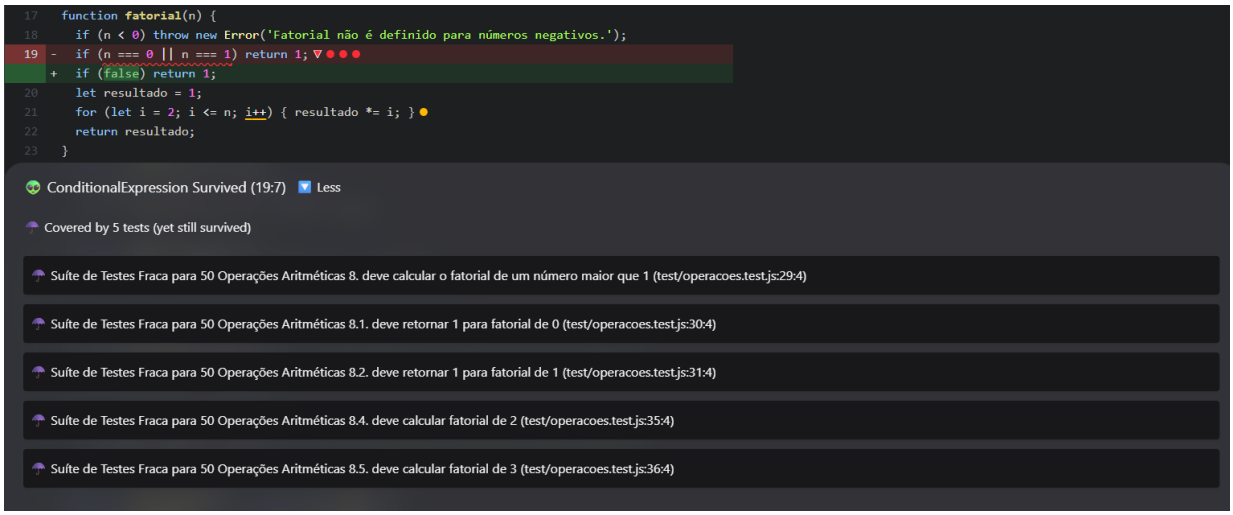

Figura 1: Screenshot do relatório StrykerJS exibindo múltiplos mutantes sobreviventes na função `fatorial`. Os mutantes em vermelho mostram as linhas 18-19, onde a condição `if (n === 0 || n === 1)` foi modificada de várias formas, mas os testes originais não conseguiram detectar essas alterações.

2.3 Mutante #3: Funções Booleanas - Retorno Sempre True

Localização: `src/operacoes.js:43-44`

Código Original:

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

Mutação Aplicada:

```
function isPar(n) { return true; }
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	Inicial	Final	Melhoria
Mutation Score	73.71%	**96.71%**	+23.00%
Mutantes Mortos	154	203	+49
Mutantes Sobreviventes	44	7	-37
Cobertura Statements	85.41%	**100%**	+14.59%
Cobertura Branches	58.82%	**100%**	+41.18%
Total de Testes	50	83	+33

Tabela 2: Resultados da Análise de Mutação - Comparação

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.

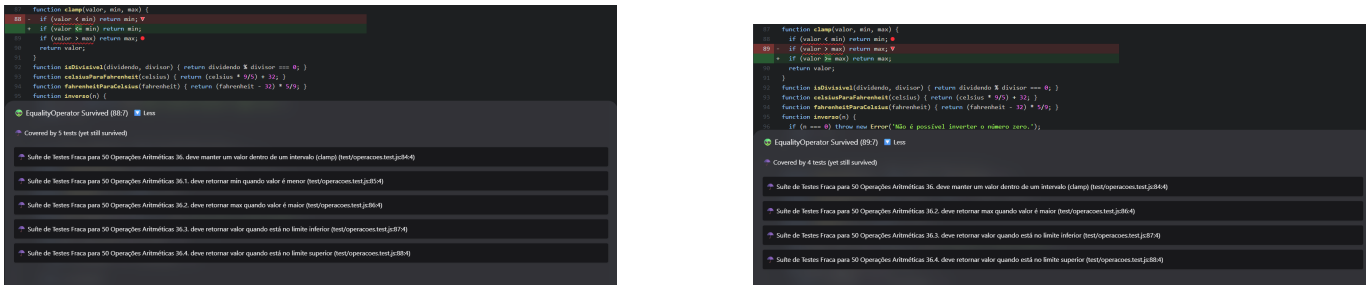

Figura 2: Mutantes sobreviventes na função `clamp` (linhas 88-89) com mutações de < para <= e > para >=. São equivalentes pelos mesmos resultados.

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