

Relatório de Teste de Mutação com StrykerJS

Disciplina: Teste de Software

Trabalho: Teste de Mutação

Nome: Matheus Hoske Aguiar

Matrícula: 728000

1. Análise Inicial

1.1 Cobertura de Código Inicial

Ao executar a análise de cobertura de código inicial com `npm test -- --coverage`, obtivemos os seguintes resultados:

File	% Stmts	% Branch	% Funcs	% Lines
All files	100	100	100	100
operacoes.js	100	100	100	100

A cobertura de código estava em **100%** em todas as métricas (Statements, Branches, Functions e Lines). Isso indica que todo o código fonte foi executado durante os testes.

1.2 Pontuação de Mutação Inicial

Após configurar e executar o StrykerJS pela primeira vez, a pontuação de mutação inicial foi de **73.71%**, com os seguintes resultados:

- **Total de mutantes:** 213
- **Mutantes mortos:** 154
- **Mutantes sobreviventes:** 44
- **Mutantes com timeout:** 3
- **Mutantes sem cobertura:** 12

1.3 Discrepância entre Cobertura e Eficácia

A discrepância entre a cobertura de código (100%) e a pontuação de mutação (73.71%) evidencia um problema fundamental: **cobertura de código não garante eficácia de testes.**

A cobertura de código mede apenas se o código foi executado, mas não verifica se os testes são capazes de detectar erros sutis. O teste de mutação revela que, apesar de toda a linha de código ter sido executada, quase 27% das mutações introduzidas não foram detectadas pelos testes, indicando que:

1. Os testes validavam apenas casos "felizes" (happy paths)
2. Faltavam testes para casos de borda (valores zero, negativos, arrays vazios)
3. Funções booleanas eram testadas apenas para um dos valores possíveis (true ou false)
4. Mensagens de erro não eram verificadas explicitamente
5. Operadores de comparação não eram testados adequadamente

2. Análise de Mutantes Críticos

Foram identificados 44 mutantes sobreviventes na primeira execução. A seguir, apresentamos três mutantes críticos que exemplificam fraquezas comuns na suíte de testes inicial.

2.1 Mutante 1: Função `isPar` - Retorno Sempre Verdadeiro

Localização: `src/operacoes.js:43`

Mutação realizada:

```
// Código Original function isPar(n) { return n % 2 === 0; } //  
Código Mutado function isPar(n) { return true; }
```

Análise:

O mutante substituiu toda a lógica da função por um retorno constante `true`. O teste original era:

```
test('15. deve retornar true para um número par', () => {  
  expect(isPar(100)).toBe(true); });
```

Por que o mutante sobreviveu:

O teste original verificava apenas se a função retornava `true` para um número par (100). Quando a função foi mutada para sempre retornar `true`, o teste continuou passando, pois o valor esperado (`true`) continuou sendo retornado, mesmo que a lógica estivesse completamente errada. O teste não verificava se a função retornaria `false` para números ímpares, permitindo que a mutação passasse despercebida.

2.2 Mutante 2: Função `clamp` - Operador de Comparação

Localização: `src/operacoes.js:88`

Mutação realizada:

```
// Código Original if (valor < min) return min; // Código Mutado if (valor <= min) return min;
```

Análise:

O mutante alterou o operador de comparação estrita (`<`) para comparação inclusiva (`<=`). O teste original era:

```
test('36. deve manter um valor dentro de um intervalo (clamp)', () => { expect(clamp(5, 0, 10)).toBe(5); });
```

Por que o mutante sobreviveu:

O teste original verificava apenas um caso onde o valor (5) estava estritamente entre min (0) e max (10), sem ser igual a nenhum deles. Com a mutação `<=`, quando o valor é menor que min, ambos os operadores se comportam igualmente. O teste não verificava o comportamento quando o valor é exatamente igual ao `min` ou ao `max`, permitindo que a mutação passasse. Além disso, faltava um teste que garantisse que valores menores que `min` são corretamente limitados.

2.3 Mutante 3: Função `isPrimo` - Condição do Loop

Localização: `src/operacoes.js:74`

Mutação realizada:

```
// Código Original for (let i = 2; i < n; i++) { if (n % i === 0) return false; } // Código Mutado for (let i = 2; false; i++) { if (n % i === 0) return false; }
```

Análise:

O mutante substituiu a condição do loop `i < n` por `false`, fazendo com que o loop nunca seja executado. O teste original era:

```
test('33. deve verificar que um número é primo', () => { expect(isPrimo(7)).toBe(true); });
```

Por que o mutante sobreviveu:

O teste verificava apenas se o número 7 (que é primo) retornava `true`. Quando a condição do loop é mutada para `false`, o loop não executa, mas a função `isPrimo` ainda retorna `true` para 7, pois o número passa pela verificação inicial `if (n <= 1) return false;` (`7 > 1`) e, como o loop não executa, a função retorna `true` no final. O teste não verificava números não primos ou casos de borda (como 0, 1, números negativos), permitindo que essa mutação crítica passasse.

3. Solução Implementada

Para eliminar os mutantes sobreviventes e aumentar a pontuação de mutação, foram implementados **38 novos testes**, elevando o total de testes de 50 para 88. As melhorias foram organizadas nas seguintes categorias:

3.1 Testes para Casos de Borda

Foram adicionados testes para valores zero, negativos e arrays vazios:

Exemplos:

- `test('6b. deve lançar erro para raiz quadrada de número negativo')` - Verifica se números negativos lançam erro com a mensagem correta
- `test('6c. deve calcular raiz quadrada de zero')` - Testa o caso limite zero
- `test('8b. deve calcular fatorial de zero')` - Testa `fatorial(0) = 1`
- `test('8c. deve calcular fatorial de um')` - Testa `fatorial(1) = 1`
- `test('9b. deve retornar zero para média de array vazio')` - Testa comportamento com array vazio

Eficácia: Esses testes mataram mutantes que alteravam condições de validação ou retornavam valores incorretos para casos de borda.

3.2 Testes para Retornos Booleanos (True e False)

Funções booleanas agora são testadas para ambos os valores possíveis:

Exemplos:

- `test('15b. deve retornar false para um número ímpar')` - Verifica que `isPar(7)` retorna `false`

- `test('16b. deve retornar false para um número par') -`
Verifica que `isImpar(100)` retorna `false`
- `test('33b. deve retornar false para número não primo') -`
Verifica que `isPrimo(4)` retorna `false`
- `test('37b. deve retornar false quando não é divisível') -`
Verifica casos negativos de divisibilidade

Eficácia: Esses testes mataram mutantes que substituíam a lógica por retornos constantes (`return true` ou `return false`).

3.3 Testes para Operadores de Comparação

Foram adicionados testes que verificam explicitamente o comportamento com operadores `>=`, `<=` e comparações de igualdade:

Exemplos:

- `test('44c. deve retornar false quando são iguais') -`
Verifica que `isMaiorQue(5, 5)` retorna `false`
- `test('45c. deve retornar false quando são iguais') -`
Verifica que `isMenorQue(5, 5)` retorna `false`
- `test('46b. deve retornar false quando não são iguais') -`
Verifica que `isEqual(7, 8)` retorna `false`

Eficácia: Esses testes mataram mutantes que alteravam operadores de comparação (`>` para `>=`, `<` para `<=`).

3.4 Testes para Mensagens de Erro Específicas

Os testes agora verificam não apenas se um erro é lançado, mas também a mensagem específica:

Exemplos:

- `test('4. deve dividir e lançar erro para divisão por zero') - Agora verifica: expect(() => divisao(5, 0)).toThrow('Divisão por zero não é permitida.');`

- `test('11b. deve lançar erro para máximo de array vazio')` -
Verifica a mensagem específica do erro
- `test('40b. deve lançar erro para inverso de zero')` -
Verifica a mensagem de erro específica

Eficácia: Esses testes mataram mutantes que alteravam as mensagens de erro para strings vazias.

3.5 Testes para Casos Especiais

Foram adicionados testes para cenários mais complexos:

Exemplos:

- `test('47b. deve calcular mediana de array par')` - Testa mediana com array de tamanho par (retorna média dos dois elementos centrais)
- `test('47d. deve calcular mediana de array não ordenado')` -
Verifica que a função ordena corretamente antes de calcular
- `test('36g. deve garantir que valor menor que min retorna min')` - Testa múltiplos valores menores que min

Eficácia: Esses testes mataram mutantes relacionados à lógica de ordenação, cálculo de mediana e tratamento de arrays.

4. Resultados Finais

4.1 Pontuação de Mutação Final

Após implementar os novos testes, a pontuação de mutação aumentou significativamente:

Métrica	Inicial	Final	Melhoria
Pontuação de Mutação	73.71%	96.71%	+23.00%
Mutantes Mortos	154	203	+49
Mutantes Sobreviventes	44	7	-37
Total de Testes	50	88	+38

4.2 Análise dos Mutantes Restantes

Os 7 mutantes sobreviventes restantes são considerados **mutantes equivalentes** - mutações que alteram o código mas produzem comportamento observável idêntico ao código original. Exemplos:

1. **Fatorial (`n === 0 || n === 1`)**: Mutantes que alteram a condição para `false || n === 1` ou `n === 0 || false` ainda produzem o resultado correto porque o comportamento final é equivalente para os valores testados.
2. **produtoArray**: O mutante `if (false)` faz com que arrays vazios passem pelo `reduce`, mas o `reduce` com valor inicial `1` ainda retorna `1`, produzindo o mesmo resultado.
3. **clamp**: Os mutantes que trocam `<` por `<=` e `>` por `>=` são equivalentes quando o valor está estritamente entre min e max, que é o caso testado.

Esses mutantes não podem ser eliminados sem alterar o comportamento observável do código ou adicionar testes que verificariam aspectos não relacionados à

funcionalidade.

4.3 Comparação: Cobertura vs. Eficácia

A tabela abaixo resume a diferença entre cobertura de código e eficácia de testes:

Métrica	Valor	Interpretação
Cobertura de Código	100%	Todo o código foi executado
Pontuação de Mutação Inicial	73.71%	Apenas 73.71% dos erros seriam detectados
Pontuação de Mutação Final	96.71%	96.71% dos erros seriam detectados

Esta comparação demonstra claramente que **cobertura de código é uma métrica enganosa** quando considerada isoladamente. Uma suíte com 100% de cobertura pode ter apenas 73% de eficácia na detecção de bugs.

5. Conclusão

O teste de mutação revelou-se uma ferramenta fundamental para avaliar a real qualidade de uma suíte de testes. Este trabalho demonstrou que:

1. **Cobertura de código é insuficiente:** Apesar de 100% de cobertura, apenas 73.71% dos mutantes foram detectados inicialmente, revelando testes que executavam código mas não validavam comportamentos críticos.
2. **Teste de mutação identifica fraquezas específicas:** A análise dos mutantes sobreviventes permitiu identificar exatamente quais casos de teste estavam faltando, como validações de casos de borda, retornos booleanos negativos e mensagens de erro específicas.
3. **Melhoria direcionada é eficaz:** Ao adicionar 38 testes específicos baseados na análise de mutantes, conseguimos aumentar a pontuação de mutação de 73.71% para 96.71%, uma melhoria de 23 pontos percentuais.
4. **Mutantes equivalentes são inevitáveis:** Alguns mutantes sobreviventes (7 no total) são equivalentes, ou seja, produzem comportamento observável idêntico. Estes não podem ser eliminados sem testes que verificariam implementações específicas ao invés de comportamentos.
5. **Teste de mutação complementa cobertura:** Enquanto cobertura de código responde "o código foi executado?", teste de mutação responde "os testes detectariam bugs?". Ambas as métricas são importantes, mas eficácia é mais valiosa que cobertura.

O teste de mutação é, portanto, uma técnica essencial para garantir que uma suíte de testes não apenas cubra o código, mas seja verdadeiramente eficaz na detecção de erros, contribuindo significativamente para a qualidade e confiabilidade do software.

Data: 02/11/2025

Repositório GitHub: <https://github.com/matheushoske/operacoes-mutante>