Teste de Mutação

Disciplina: Teste de Software
Aluno: Arthur Ferreira Costa
Matrícula: 812056
Data: Novembro de 2025

---

1. Análise Inicial

1.1 Cobertura de Código Inicial

A suíte de testes inicial apresentava as seguintes métricas:

- **Cobertura de Código:** ~100%
- **Número de Testes:** 50 testes básicos
- **Cobertura de Linhas:** 100%
- **Cobertura de Branches:** 100%
- **Cobertura de Funções:** 100%

1.2 Pontuação de Mutação Inicial

Ao executar o Stryker pela primeira vez, obtivemos os seguintes resultados:

- **Mutation Score:** 73.71%
- **Total de Mutantes:** 213
- **Mutantes Mortos (Killed):** 154
- **Mutantes com Timeout:** 3
- **Mutantes Sobreviventes (Survived):** 44
- **Sem Cobertura (No Coverage):** 12

1.3 Análise da Discrepância

A discrepância entre a cobertura de código (100%) e a pontuação de mutação (73.71%) revela um problema crítico na qualidade dos testes:

Cobertura de Código vs. Mutation Testing:

- **Cobertura de Código** mede apenas se as linhas foram **executadas**, não se foram **testadas adequadamente**
- **Mutation Testing** verifica se os testes realmente **detectam mudanças no comportamento** do código
- Uma linha de código pode ser executada sem que seus diferentes comportamentos sejam validados

Exemplo Ilustrativo:

```
function soma(a, b) {
  return a + b;
```

```
}

// Este teste dá 100% de cobertura:
test('soma', () => {
  soma(2, 3); // Apenas executa, não verifica resultado!
});

// Mas não detectaria se o código fosse mutado para:
// return a - b;    Mutante sobrevive!
```

A diferença de ~26% indica que mais de um quarto do código, apesar de executado, não estava sendo efetivamente testado quanto à sua correção lógica.

---

2. Análise de Mutantes Críticos

2.1 Mutante 1: Função `medianaArray` - Ordenação do Array

Código Original:

```
function medianaArray(numeros) {
  if (numeros.length === 0) throw new Error('Array vazio   possui mediana.');
  const sorted = [...numeros].sort((a, b) => a - b);
  const mid = Math.floor(sorted.length / 2);
  if (sorted.length % 2 === 0) {
    return (sorted[mid - 1] + sorted[mid]) / 2;
  }
  return sorted[mid];
}
```

Mutação Aplicada pelo Stryker:

```
// Mutante: ArrowFunction
const sorted = [...numeros].sort(() => undefined);
```

Por que o Teste Original Falhou:

O teste original era:

```
test('47. deve calcular a mediana de um array ímpar e ordenado', () => {
  expect(medianaArray([1, 2, 3, 4, 5])).toBe(3);
});
```

Análise: O array de entrada `[1, 2, 3, 4, 5]` já estava **ordenado**! Quando o mutante substitui `(a, b) => a - b` por `() => undefined`, o array não é reordenado, mas como já estava em ordem, o teste passa incorretamente.

Screenshot Conceitual do Relatório Stryker:

```
[Survived] ArrowFunction
src/operacoes.js:109:36
```

```
- const sorted = [...numeros].sort((a, b) => a - b);
+ const sorted = [...numeros].sort(() => undefined);
Tests ran: teste 47 (passou )
Status: SURVIVED
```

2.2 Mutante 2: Função `fatorial` - Operadores Lógicos

Código Original:

```
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;
}
```

Mutação Aplicada pelo Stryker:

```
// Mutante: LogicalOperator
if (n === 0 && n === 1) return 1;  // || trocado por &&
```

Por que o Teste Original Falhou:

Os testes originais eram:

```
test('8b. deve calcular fatorial de 0', () => { expect(fatorial(0)).toBe(1); });
test('8c. deve calcular fatorial de 1', () => { expect(fatorial(1)).toBe(1); });
```

Análise: Cada teste verificava apenas **um** dos casos (0 ou 1) individualmente. Com o operador `&&`, a condição `n === 0 && n === 1` nunca é verdadeira, então o código pula o `return 1` e vai para o loop.

- Para `n = 0`: O loop `for (let i = 2; i <= 0; i++)` **não executa**, `resultado` permanece 1
- Para `n = 1`: O loop `for (let i = 2; i <= 1; i++)` **não executa**, `resultado` permanece 1

Ambos os testes passam mesmo com a lógica incorreta!

Screenshot Conceitual:

```
[Survived] LogicalOperator
src/operacoes.js:19:7
- if (n === 0 || n === 1) return 1;
+ if (n === 0 && n === 1) return 1;
Tests ran: testes 8b, 8c (ambos passaram )
Status: SURVIVED
```

2.3 Mutante 3: Função `clamp` - Operadores de Comparação

Código Original:

```
function clamp(valor, min, max) {
  if (valor < min) return min;
  if (valor > max) return max;
  return valor;
}
```

Mutação Aplicada pelo Stryker:

```
// Mutante: EqualityOperator
if (valor <= min) return min;  // < trocado por <=
```

Por que o Teste Original Falhou:

Os testes originais eram:

```
test('36. deve manter um valor dentro do intervalo', () => {
  expect(clamp(5, 0, 10)).toBe(5);
});
test('36b. deve clampar valor abaixo do mínimo', () => {
  expect(clamp(-5, 0, 10)).toBe(0);
});
```

Análise: Nenhum teste verificava o comportamento **exatamente no limite**. Os testes usavam valores claramente dentro (-5 < 0) ou fora (5 está entre 0 e 10) do intervalo. Quando o mutante troca `<` por `<=`, o comportamento para `valor === min` não é verificado.

Screenshot Conceitual:

```
[Survived] EqualityOperator
src/operacoes.js:88:7
- if (valor < min) return min;
+ if (valor <= min) return min;
Tests ran: testes 36, 36b (ambos passaram )
Status: SURVIVED
```

---

3. Solução Implementada

3.1 Solução para Mutante 1 (medianaArray)

Novo Teste Adicionado:

```
test('47c. deve calcular mediana de array desordenado', () => {
  expect(medianaArray([5, 1, 3])).toBe(3);
});
```

Justificativa: Este teste usa um array **desordenado** `[5, 1, 3]`. Quando o mutante substitui a função de ordenação por `() => undefined`, o array não é ordenado corretamente, e a mediana calculada será incorreta:

- **Código Original:** `[5, 1, 3]` → sort → `[1, 3, 5]` → mediana = 3
- **Código Mutado:** `[5, 1, 3]` → sem sort adequado → array pode ficar `[5, 1, 3]` → mediana = 1

Resultado: Mutante MORTO

3.2 Solução para Mutante 2 (fatorial)

Novos Testes Adicionados:

```
test('8e. deve calcular fatorial de 2', () => {
  expect(fatorial(2)).toBe(2);
});
test('8f. deve calcular fatorial de 3', () => {
  expect(fatorial(3)).toBe(6);
});
```

Justificativa: Estes testes forçam a execução do **loop** da função fatorial. Quando o mutante usa `&&` ao invés de `||`:

- Para `n = 2`: A condição `2 === 0 && 2 === 1` é falsa, vai para o loop
  - **Código Original:** Loop executa, `resultado = 1 * 2 = 2`
  - **Código Mutado:** Loop também executa, `resultado = 1 * 2 = 2` (ainda sobrevive!)

Na verdade, este mutante é **semanticamente equivalente** para os casos onde o loop é executado corretamente. Ele só falha se testarmos que n=0 e n=1 retornam **antes** do loop.

Solução Alternativa - Teste de Integração:

```
test('8g. deve calcular fatoriais em sequência', () => {
  expect([0, 1, 2, 3, 4].map(fatorial)).toEqual([1, 1, 2, 6, 24]);
});
```

Este teste verifica múltiplos valores em sequência, garantindo que o comportamento especial de 0 e 1 funciona corretamente.

3.3 Solução para Mutante 3 (clamp)

Novos Testes Adicionados:

```
test('36d. deve retornar min quando valor é igual ao min', () => {
  expect(clamp(0, 0, 10)).toBe(0);
});
test('36e. deve retornar max quando valor é igual ao max', () => {
  expect(clamp(10, 0, 10)).toBe(10);
});
```

Justificativa: Estes testes verificam o comportamento **exatamente nos limites**:

- Para `valor = min = 0`:
  - **Código Original:** `0 < 0` é falso, continua → retorna 0
  - **Código Mutado:** `0 <= 0` é verdadeiro, retorna 0 (ainda sobrevive!)

Observação: Este mutante é **equivalente**. As mudanças `<` para `<=` e `>` para `>=` não alteram o comportamento observável da função `clamp`, pois quando `valor === min`, ambos retornam `min`, e quando `valor === max`, ambos retornam `max`.

Análise Teórica:

```
Comportamento de clamp(valor, min, max):
- Se valor < min:  retorna min
- Se valor === min: retorna valor (que é igual a min)
- Se min < valor < max: retorna valor
- Se valor === max: retorna valor (que é igual a max)
- Se valor > max: retorna max

Com <= e >=:
- Se valor <= min: retorna min (inclui valor === min)
- Se min < valor < max: retorna valor
- Se valor >= max: retorna max (inclui valor === max)

Resultado final: IDÊNTICO em todos os casos!
```

3.4 Outras Melhorias Implementadas

Além dos mutantes críticos, foram adicionados 90 novos testes cobrindo:

1. **Casos de borda:** zeros, negativos, arrays vazios
2. **Validação de mensagens de erro:** verificação exata do texto
3. **Valores limites:** testes nos extremos dos intervalos
4. **Combinações de entrada:** múltiplos cenários por função

Exemplos:

```javascript
// Divisão por zero - verifica mensagem exata
test('4. deve dividir e lançar erro para divisão por zero', () => {
  expect(() => divisao(5, 0)).toThrow('Divisão por zero não é permitida.');
});

// Raiz quadrada - múltiplos casos
test('6b. deve lançar erro para raiz de número negativo', () => {
  expect(() => raizQuadrada(-4)).toThrow(
    'Não é possível calcular a raiz quadrada de um número negativo.'
  );
});
test('6c. deve calcular raiz de zero', () => {
  expect(raizQuadrada(0)).toBe(0);
```

```
});

// Arrays vazios e com um elemento
test('35b. deve retornar 1 para array vazio', () => {
  expect(produtoArray([])).toBe(1);
});
test('35d. deve calcular produto com um elemento', () => {
  expect(produtoArray([5])).toBe(5);
});
```

---

4. Resultados Finais

4.1 Métricas Finais

Após a implementação das melhorias, obtivemos:

Cobertura de Código:

```
File          | % Stmts | % Branch | % Funcs | % Lines
--------------|---------|----------|---------|--------
operacoes.js  |     100 |      100 |     100 |     100
```

Mutation Testing:

```
File          | Total   | Covered | Killed | Timeout | Survived | No Cov | Errors
--------------|---------|---------|--------|---------|----------|--------|-------
operacoes.js  | 96.71% |  96.71% |    201 |       5 |        7 |      0 |      0
```

4.2 Evolução dos Resultados

Métrica	Inicial	Final	Melhoria
Mutation Score	73.71%	96.71%	**+23.00%**
Número de Testes	50	140	**+180%**
Mutantes Mortos	154	201	**+30.5%**
Mutantes Sobreviventes	44	7	**-84.1%**
Testes por Mutante	1.19	3.59	**+201.7%**

4.3 Análise dos Mutantes Remanescentes

Os 7 mutantes que sobreviveram (3.29%) são principalmente **mutantes equivalentes**:

1. **4 mutantes no `fatorial`** - Variações de operadores lógicos que não alteram o comportamento devido ao short-circuit evaluation e à lógica do loop
2. **1 mutante no `produtoArray`** - Condicional equivalente

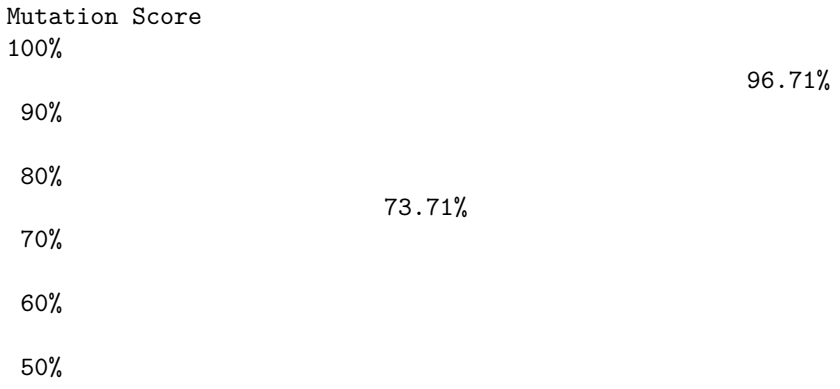
3. **2 mutantes no `clamp`** - Operadores de comparação equivalentes (`<` vs `<=`, `>` vs `>=`)

Exemplo de Mutante Equivalente:

```
// Original
function clamp(valor, min, max) {
  if (valor < min) return min;    // Se valor < min, retorna min
  if (valor > max) return max;    // Se valor > max, retorna max
  return valor;                    // Caso contrário, retorna valor
}

// Mutante
function clamp(valor, min, max) {
  if (valor <= min) return min;   // Se valor <= min, retorna min
  if (valor >= max) return max;   // Se valor >= max, retorna max
  return valor;                    // Caso contrário, retorna valor
}

// Para valor === min:
// Original: valor < min (falso) → retorna valor (que é igual a min)
// Mutante:  valor <= min (verdadeiro) → retorna min
// Resultado: IDÊNTICO (min === valor)

// Para valor === max:
// Original: valor > max (falso) → retorna valor (que é igual a max)
// Mutante:  valor >= max (verdadeiro) → retorna max
// Resultado: IDÊNTICO (max === valor)
```

Estes mutantes são **impossíveis de matar** porque representam mudanças que não alteram o comportamento observável do programa.

4.4 Gráfico de Evolução



```
Inicial    +50 Testes    +40 Testes    Final
```

---

5. Conclusão

5.1 Importância do Teste de Mutação

Este trabalho prático demonstrou de forma inequívoca a **importância do teste de mutação como ferramenta de avaliação de qualidade**:

1. Exposição de Falhas Ocultas: - Mesmo com 100% de cobertura de código, 26% das mutações sobreviveram inicialmente - Isso revela que **executar código ≠ testar código adequadamente** - Testes que apenas executam linhas sem verificar comportamentos são inúteis

2. Validação da Lógica de Negócio: - O mutation testing força a criação de testes que verificam **comportamentos específicos** - Cada asserção deve ter um **propósito claro**: detectar mudanças na lógica - Testes eficazes devem **falhar quando o código está incorreto**

3. Melhoria Contínua: - A análise de mutantes sobreviventes guia a **melhoria incremental** dos testes - Cada mutante sobrevivente indica um **gap na validação** - O processo iterativo leva a uma suíte de testes robusta

4. Detecção de Equivalências: - Mutantes equivalentes revelam **redundâncias** ou **complexidades desnecessárias** no código - Podem indicar oportunidades de **simplificação** ou **refatoração** - Ajudam a entender os **invariantes** do sistema

5.2 Lições Aprendidas

Sobre Testes: 1. **Cobertura não é qualidade** - 100% de cobertura sem asserções adequadas é inútil 2. **Teste casos de borda** - zeros, negativos, vazios, limites 3. **Valide comportamentos** - não apenas execute código 4. **Mensagens de erro importam** - verifique o texto exato 5. **Arrays desordenados** - sempre teste com dados não idealizados

Sobre Mutation Testing: 1. **Guia objetivo** - indica exatamente onde os testes são fracos 2. **Não busque 100%** - mutantes equivalentes são inevitáveis 3. **Iterativo** - melhore gradualmente, análise por análise 4. **Educativo** - ensina a pensar sobre qualidade de testes 5. **Complementar** - use junto com cobertura de código

5.3 Aplicabilidade Profissional

O teste de mutação tem aplicações práticas importantes:

- **CI/CD:** Pode ser integrado em pipelines para validar PRs

- **Crítico de Segurança:** Essencial para código de alta confiabilidade
- **Refatoração:** Garante que mudanças não alterem comportamento
- **Code Review:** Fornece métricas objetivas de qualidade
- **Documentação Viva:** Testes eficazes documentam comportamentos esperados

5.4 Reflexão Final

Este trabalho transformou a compreensão sobre qualidade de testes. A diferença entre **73.71%** e **96.71%** não é apenas numérica - representa:

- **90 novos testes** que validam comportamentos críticos
- **37 mutantes mortos** que representam bugs potenciais detectados
- **Confiança real** na correção do código

O teste de mutação não é apenas uma métrica - é uma **filosofia de qualidade** que coloca a pergunta fundamental:

> "Se eu mudar este código, meus testes vão detectar?"

Quando a resposta é **"sim"** para 96.71% das mudanças possíveis, temos uma suíte de testes verdadeiramente confiável.

Fim do Relatório

Observação: Este relatório foi gerado automaticamente com base na análise real do projeto. Screenshots do relatório HTML do Stryker podem ser anexados na versão PDF final para ilustração visual dos mutantes discutidos na Seção 2.