Puc Minas

Engenharia de Software

Disciplina: Testes de Software

Análise de Eficácia de Testes com Teste de Mutação

Trabalho Prático - StrykerJS

Nome: Luiz Nery

02 de novembro de 2025

Conteúdo

1 Introdução **3**
1.1 Objetivos . . . 3

2 Análise Inicial **3**
2.1 Cobertura de Código Inicial . . . 3
2.2 Pontuação de Mutação Inicial . . . 4
2.3 Discrepância entre Cobertura e Eficácia . . . 4

3 Análise de Mutantes Críticos **4**
3.1 Mutante Crítico #1: Comparadores de Limites . . . 5
3.1.1 Localização e Descrição . . . 5
3.1.2 Mutações que Sobreviveram . . . 5
3.1.3 Por que os Testes Originais Falharam? . . . 5
3.2 Mutante Crítico #2: Função isPrimo . . . 6
3.2.1 Localização e Descrição . . . 6
3.2.2 Mutações que Sobreviveram . . . 6
3.2.3 Por que os Testes Originais Falharam? . . . 6
3.3 Mutante Crítico #3: Conversão de Temperatura . . . 7
3.3.1 Localização e Descrição . . . 7
3.3.2 Mutações que Sobreviveram . . . 7
3.3.3 Por que os Testes Originais Falharam? . . . 7

4 Soluções Implementadas **8**
4.1 Solução para Comparadores de Limites . . . 8
4.2 Solução para isPrimo . . . 8
4.3 Solução para Conversão de Temperatura . . . 10
4.4 Outras Melhorias Implementadas . . . 11
4.4.1 Validação de Mensagens de Erro Específicas . . . 11
4.4.2 Testes para Arrays Vazios . . . 11
4.4.3 Testes para Função Clamp (Limites) . . . 11
4.4.4 Testes para Mediana com Ordenação . . . 11

5 Resultados Finais **12**
5.1 Evolução da Pontuação de Mutação . . . 12
5.2 Análise Comparativa . . . 12
5.3 Métricas de Cobertura Final . . . 12
5.4 Impacto Prático da Melhoria . . . 13
5.5 Mutantes Remanescentes . . . 13

6 Conclusão **14**
6.1 Principais Aprendizados . . . 14
6.1.1 1. Cobertura de Código Qualidade de Testes . . . 14
6.1.2 2. Teste de Mutação Revela Fraquezas Ocultas . . . 14
6.1.3 3. Testes Eficazes Requerem Pensamento Crítico . . . 14
6.1.4 4. Boundary Conditions São Críticos . . . 14
6.2 Importância do Teste de Mutação na Engenharia de Software . . . 15
6.2.1 Vantagens Práticas . . . 15

6.2.2 Quando Usar 15
6.3 Limitações e Considerações 15
6.4 Reflexão Final 16
6.5 Próximos Passos 16

7 Referências **16**

1 Introdução

Este trabalho tem como objetivo demonstrar na prática a diferença fundamental entre **cobertura de código** e **eficácia de testes**. Utilizando a ferramenta StrykerJS para análise de teste de mutação, foi possível identificar fraquezas em uma suíte de testes que, apesar de apresentar alta cobertura de código, não garantia a detecção efetiva de bugs.

O teste de mutação funciona introduzindo pequenas alterações (mutações) no código fonte e verificando se os testes são capazes de detectá-las. Um mutante que "sobrevive" indica uma fraqueza na suíte de testes, pois uma alteração no código passou despercebida.

1.1 Objetivos

Os principais objetivos deste trabalho foram:

- Configurar e executar análise de teste de mutação em um projeto JavaScript
- Interpretar relatórios do StrykerJS e identificar mutantes sobreviventes
- Analisar criticamente as fraquezas da suíte de testes
- Escrever testes eficazes focados em eliminar mutantes específicos
- Demonstrar melhoria mensurável através do aumento da pontuação de mutação
- Compreender a diferença entre cobertura de código e qualidade de testes

2 Análise Inicial

2.1 Cobertura de Código Inicial

A primeira etapa consistiu em executar a suíte de testes existente e gerar o relatório de cobertura de código utilizando Jest. Os resultados iniciais foram:

Tabela 1: Métricas de Cobertura de Código Inicial

Métrica	**Percentual**
Statements (Declarações)	85,41%
Branches (Ramificações)	58,82%
Functions (Funções)	**100%**
Lines (Linhas)	98,64%
Número de Testes	**50**

À primeira vista, estes números parecem excelentes: 100% de cobertura de funções e 98,64% de cobertura de linhas. No entanto, como será demonstrado, alta cobertura não garante testes eficazes.

2.2 Pontuação de Mutação Inicial

Após executar o StrykerJS pela primeira vez com o comando `npx stryker run`, os resultados revelaram uma realidade diferente:

Tabela 2: Resultados da Primeira Análise de Mutação

Métrica	**Valor**
Mutation Score	**73,71%**
Mutantes Criados	213
Mutantes Mortos (Killed)	154
Mutantes Sobreviventes (Survived)	**56**
Timeouts	3

2.3 Discrepância entre Cobertura e Eficácia

A comparação entre as métricas de cobertura e a pontuação de mutação revela um problema crítico:

- **Cobertura de Linhas:** 98,64% (aparentemente excelente)
- **Mutation Score:** 73,71% (preocupante)
- **Diferença:** 24,93 pontos percentuais

Por que isso acontece?

A cobertura de código mede apenas se o código foi *executado* durante os testes, mas não verifica se o comportamento foi *corretamente validado*. Um teste pode executar uma função inteira sem fazer asserções adequadas sobre os valores de retorno ou comportamentos esperados.

Por exemplo, o teste original da função `isMaiorQue()`:

```
test('deve verificar se um numero e maior que outro', () => {
    expect(isMaiorQue(10, 5)).toBe(true);
});
```

Listing 1: Teste original (fraco)

Este teste executa a função (cobertura = 100%), mas não testa o caso crítico onde `a === b`. Quando o StrykerJS muta `a > b` para `a >= b`, o teste continua passando, pois `10 >= 5` também retorna `true`.

3 Análise de Mutantes Críticos

Durante a análise do relatório HTML gerado pelo StrykerJS (`reports/mutation/mutation.html`), foram identificados diversos mutantes sobreviventes. Três casos foram selecionados para análise detalhada por representarem categorias importantes de falhas em testes.

3.1 Mutante Crítico #1: Comparadores de Limites

3.1.1 Localização e Descrição

Arquivo: `src/operacoes.js`
Linhas: 104-106
Funções afetadas: `isMaiorQue()`, `isMenorQue()`, `isEqual()`

O código original implementa funções simples de comparação:

```
function isMaiorQue(a, b) { return a > b; }
function isMenorQue(a, b) { return a < b; }
function isEqual(a, b) { return a === b; }
```

Listing 2: Código original das funções de comparação

3.1.2 Mutações que Sobreviveram

O StrykerJS gerou as seguintes mutações que **não foram detectadas** pelos testes:

```
// Mutante 1: Operador > virou >=
function isMaiorQue(a, b) { return a >= b; }

// Mutante 2: Operador < virou <=
function isMenorQue(a, b) { return a <= b; }

// Mutante 3: Condicional virou constante true
function isMaiorQue(a, b) { return true; }
function isMenorQue(a, b) { return true; }
function isEqual(a, b) { return true; }
```

Listing 3: Mutantes sobreviventes - Comparadores

3.1.3 Por que os Testes Originais Falharam?

Os testes originais apenas verificavam casos onde `a != b`:

```
test('44. deve verificar se um numero e maior que outro', () => {
    expect(isMaiorQue(10, 5)).toBe(true);
});
test('45. deve verificar se um numero e menor que outro', () => {
    expect(isMenorQue(5, 10)).toBe(true);
});
test('46. deve verificar se dois numeros sao iguais', () => {
    expect(isEqual(7, 7)).toBe(true);
});
```

Problema identificado:

- Não testava o caso `a === b` para `isMaiorQue()` e `isMenorQue()`
- Para `isMaiorQue(10, 5)`, tanto `>` quanto `>=` retornam `true`
- Não testava valores onde a comparação deveria retornar `false`
- Não testava `isEqual()` com valores diferentes

Este é um exemplo clássico de falha em testar **boundary conditions** (condições de fronteira).

3.2 Mutante Crítico #2: Função isPrimo

3.2.1 Localização e Descrição

Arquivo: `src/operacoes.js`
Linhas: 73-78
Função afetada: `isPrimo()`

Código original:

```
function isPrimo(n) {
  if (n <= 1) return false;
  for (let i = 2; i < n; i++) {
    if (n % i === 0) return false;
  }
  return true;
}
```

Listing 4: Implementação da função isPrimo

3.2.2 Mutações que Sobreviveram

Total de 8 mutantes sobreviventes nesta função, incluindo:

```
// Mutante 1: <= virou <
if (n < 1) return false;

// Mutante 2: false virou true
if (n <= 1) return true;

// Mutante 3: < virou >=
for (let i = 2; i >= n; i++)

// Mutante 4: % virou *
if (n * i === 0) return false;

// Mutante 5: Bloco do loop removido
for (let i = 2; i < n; i++) {}
```

Listing 5: Exemplos de mutantes sobreviventes

3.2.3 Por que os Testes Originais Falharam?

O teste original verificava apenas um caso:

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

Casos NÃO testados:

- Números menores ou iguais a 1 (0, 1, -5)
- O número 2 (menor número primo)
- Números pares maiores que 2 (4, 6, 8)
- Números compostos ímpares (9, 15, 21)
- Validação de que números não-primos retornam `false`

Testar apenas um caminho "feliz" (7 é primo) deixa diversos mutantes vivos, pois não exercita adequadamente as condições de guarda e o loop de verificação.

3.3 Mutante Crítico #3: Conversão de Temperatura

3.3.1 Localização e Descrição

Arquivo: `src/operacoes.js`
Linhas: 93-94
Funções afetadas: `celsiusParaFahrenheit()`, `fahrenheitParaCelsius()`

Código original:

```
function celsiusParaFahrenheit(celsius) {
    return (celsius * 9/5) + 32;
}

function fahrenheitParaCelsius(fahrenheit) {
    return (fahrenheit - 32) * 5/9;
}
```

Listing 6: Funções de conversão de temperatura

3.3.2 Mutações que Sobreviveram

```
// Mutante 1: * virou /
return (celsius / 9/5) + 32;

// Mutante 2: /5 virou *5
return (celsius * 9 * 5) + 32;

// Mutante 3: * virou /
return (fahrenheit - 32) / 5/9;

// Mutante 4: /9 virou *9
return (fahrenheit - 32) * 5 * 9;
```

Listing 7: Mutantes nas fórmulas matemáticas

3.3.3 Por que os Testes Originais Falharam?

Os testes originais verificavam apenas um ponto específico:

```
test('38. deve converter Celsius para Fahrenheit', () => {
    expect(celsiusParaFahrenheit(0)).toBe(32);
});

test('39. deve converter Fahrenheit para Celsius', () => {
    expect(fahrenheitParaCelsius(32)).toBe(0);
});
```

Problema crítico:

Quando `celsius = 0`, a parte `(celsius * 9/5)` resulta em 0, independente de a operação estar correta ou não. Assim, tanto `(0 * 9/5)` quanto `(0 / 9/5)` ou `(0 * 9 * 5)` resultam em 0, e a função retorna 32 em todos os casos.

Lição aprendida: Testar apenas valores que anulam parte da lógica (como 0 em multiplicações) não valida adequadamente a correção da fórmula matemática.

4 Soluções Implementadas

4.1 Solução para Comparadores de Limites

Para eliminar os mutantes das funções de comparação, foram adicionados testes que cobrem todos os casos possíveis:

```
// Testes para isMaiorQue
test('44. deve verificar se um numero e maior que outro', () => {
    expect(isMaiorQue(10, 5)).toBe(true);
});
test('44b. deve retornar false quando numeros sao iguais', () => {
    expect(isMaiorQue(5, 5)).toBe(false);
});
test('44c. deve retornar false quando primeiro numero e menor', () => {
    expect(isMaiorQue(3, 8)).toBe(false);
});

// Testes para isMenorQue
test('45. deve verificar se um numero e menor que outro', () => {
    expect(isMenorQue(5, 10)).toBe(true);
});
test('45b. deve retornar false quando numeros sao iguais', () => {
    expect(isMenorQue(5, 5)).toBe(false);
});
test('45c. deve retornar false quando primeiro numero e maior', () => {
    expect(isMenorQue(10, 3)).toBe(false);
});

// Testes para isEqual
test('46. deve verificar se dois numeros sao iguais', () => {
    expect(isEqual(7, 7)).toBe(true);
});
test('46b. deve retornar false quando numeros sao diferentes', () => {
    expect(isEqual(7, 8)).toBe(false);
});
```

Listing 8: Testes aprimorados para comparadores

Por que esses testes são eficazes?

- **Testam boundary conditions:** O caso `a === b` é crítico para distinguir `>` de `>=`
- **Testam ambos os resultados:** Verificam tanto `true` quanto `false`
- **Cobrem todas as combinações:** a $>$ b, a $<$ b, a $===$ b
- **Impedem substituições:** `a > b` não pode ser substituído por `a >= b` sem falhar

Resultado: 6 mutantes eliminados nesta categoria.

4.2 Solução para isPrimo

A estratégia foi testar sistematicamente todos os casos limites e caminhos possíveis:

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

```
4
5  test('33b. deve retornar false para numeros menores ou iguais a 1', ()
       => {
6      expect(isPrimo(0)).toBe(false);
7      expect(isPrimo(1)).toBe(false);
8      expect(isPrimo(-5)).toBe(false);
9  });
10
11 test('33c. deve retornar true para o numero 2 (menor primo)', () => {
12     expect(isPrimo(2)).toBe(true);
13 });
14
15 test('33d. deve retornar false para numeros pares maiores que 2', () =>
       {
16     expect(isPrimo(4)).toBe(false);
17     expect(isPrimo(6)).toBe(false);
18     expect(isPrimo(8)).toBe(false);
19 });
20
21 test('33e. deve retornar false para numeros compostos impares', () => {
22     expect(isPrimo(9)).toBe(false);
23     expect(isPrimo(15)).toBe(false);
24 });
```

Listing 9: Suíte completa de testes para isPrimo

Cobertura de casos:

Tabela 3: Casos de Teste para isPrimo

Caso	**Justificativa**
n ≤ 1	Testa condição de guarda inicial
n $= 2$	Menor número primo, caso especial
n par > 2	Devem ser detectados rapidamente
n ímpar composto	Exercita o loop completo
n primo	Valida resultado positivo

Por que esses testes são eficazes?

- Testam a condição de guarda (`n <= 1`)
- Validam que o operador `<=` não pode ser `<`
- Exercitam o loop com diferentes valores
- Garantem que `%` não pode ser substituído por `*`
- Verificam que o loop não pode ser removido

Resultado: 8 mutantes eliminados.

4.3 Solução para Conversão de Temperatura

A chave foi testar múltiplos valores que validam toda a fórmula matemática:

```
// Celsius para Fahrenheit
test('38. deve converter Celsius para Fahrenheit', () => {
    expect(celsiusParaFahrenheit(0)).toBe(32);
});
test('38b. deve converter 100C para 212F', () => {
    expect(celsiusParaFahrenheit(100)).toBe(212);
});
test('38c. deve converter -40C para -40F', () => {
    expect(celsiusParaFahrenheit(-40)).toBe(-40);
});
test('38d. deve converter 37C para 98.6F (temperatura corporal)', () => {
    expect(celsiusParaFahrenheit(37)).toBeCloseTo(98.6, 1);
});

// Fahrenheit para Celsius
test('39. deve converter Fahrenheit para Celsius', () => {
    expect(fahrenheitParaCelsius(32)).toBe(0);
});
test('39b. deve converter 212F para 100C', () => {
    expect(fahrenheitParaCelsius(212)).toBe(100);
});
test('39c. deve converter -40F para -40C', () => {
    expect(fahrenheitParaCelsius(-40)).toBe(-40);
});
test('39d. deve converter 98.6F para 37C (temperatura corporal)', () => {
    expect(fahrenheitParaCelsius(98.6)).toBeCloseTo(37, 1);
});
```

Listing 10: Testes robustos para conversão de temperatura

Estratégia de valores escolhidos:

- **0°C / 32°F:** Ponto de congelamento da água
- **100°C / 212°F:** Ponto de ebulição da água
- **-40°C / -40°F:** Ponto onde as escalas coincidem
- **37°C / 98.6°F:** Temperatura corporal (valor não-inteiro)

Por que esses testes são eficazes?

- Valores não-zero garantem que todas as operações matemáticas são validadas
- O ponto -40 detecta erros de sinal
- Valores conhecidos permitem verificação imediata
- Impossível que mutações matemáticas (`*` → `/`, `/` → `*`) passem despercebidas

Resultado: 4 mutantes eliminados.

4.4 Outras Melhorias Implementadas

Além dos três casos críticos analisados, diversas outras melhorias foram implementadas:

4.4.1 Validação de Mensagens de Erro Específicas

```
// Antes (fraco)
expect(() => divisao(5, 0)).toThrow();

// Depois (forte)
expect(() => divisao(5, 0)).toThrow('Divisao por zero nao e permitida.')
    ;
```

Isso elimina mutantes que alteram strings literais nas mensagens de erro.

4.4.2 Testes para Arrays Vazios

```
test('9b. deve retornar 0 para array vazio', () => {
    expect(mediaArray([])).toBe(0);
});

test('11b. deve lancar erro para array vazio (maximoArray)', () => {
    expect(() => maximoArray([])).toThrow('Array vazio');
});
```

4.4.3 Testes para Função Clamp (Limites)

```
test('36b. deve retornar minimo quando valor e menor que min', () => {
    expect(clamp(-5, 0, 10)).toBe(0);
});
test('36c. deve retornar maximo quando valor e maior que max', () => {
    expect(clamp(15, 0, 10)).toBe(10);
});
test('36d. deve retornar min quando valor e igual a min', () => {
    expect(clamp(0, 0, 10)).toBe(0);
});
test('36e. deve retornar max quando valor e igual a max', () => {
    expect(clamp(10, 0, 10)).toBe(10);
});
```

4.4.4 Testes para Mediana com Ordenação

```
test('47c. deve calcular a mediana de array desordenado', () => {
    expect(medianaArray([5, 1, 3, 2, 4])).toBe(3);
});
test('47g. deve calcular mediana com valores negativos', () => {
    expect(medianaArray([-5, -1, -3])).toBe(-3);
    expect(medianaArray([-10, -5, 0, 5])).toBe(-2.5);
});
```

5 Resultados Finais

5.1 Evolução da Pontuação de Mutação

A tabela abaixo apresenta a evolução completa dos resultados ao longo do processo de melhoria:

Tabela 4: Evolução Completa dos Resultados

Métrica	**Inicial**	**1ª Melhoria**	**Final**	**Variação**
Mutation Score	73,71%	94,84%	**96,71%**	+23,00 pp
Mutantes Mortos	154	199	**203**	+49
Mutantes Sobreviventes	56	11	**7**	-49
Timeouts	3	3	3	0
Número de Testes	50	92	**99**	+49

5.2 Análise Comparativa

Antes	**Depois**
• 50 testes	• 99 testes (+98%)
• 73,71% mutation score	• 96,71% mutation score
• 56 mutantes vivos	• 7 mutantes vivos (-87,5%)
• Testes superficiais	• Testes robustos
• Apenas caminhos felizes	• Casos positivos e negativos
• Sem boundary conditions	• Boundary conditions testadas

Figura 1: Comparação: Antes vs Depois

5.3 Métricas de Cobertura Final

Interessantemente, as métricas tradicionais de cobertura de código permaneceram praticamente inalteradas:

Tabela 5: Cobertura de Código: Inicial vs Final

Métrica	**Inicial**	**Final**
Statements	85,41%	85,41%
Branches	58,82%	58,82%
Functions	100%	100%
Lines	98,64%	98,64%
Mutation Score	**73,71%**	**96,71%**

Isso demonstra de forma conclusiva que **cobertura de código não mede qualidade de testes**. A cobertura permaneceu a mesma, mas a eficácia dos testes aumentou

dramaticamente.

5.4 Impacto Prático da Melhoria

A diferença entre 73,71% e 96,71% de mutation score representa:

- **Antes:** Apenas 3 em cada 4 bugs introduzidos seriam detectados
- **Depois:** Praticamente todos os bugs (97 em cada 100) seriam detectados
- **Redução de risco:** 87,5% menos mutantes escapando

Em um cenário real de produção, isso significa:

- Menos bugs em produção
- Maior confiança ao refatorar código
- Detecção precoce de regressões
- Redução de custos com correção de bugs

5.5 Mutantes Remanescentes

Os 7 mutantes que ainda sobrevivem são casos extremamente específicos:

1. **Fatorial - Condições booleanas internas (4 mutantes):**
   - Mutações em subexpressões do `if (n === 0 || n === 1)`
   - Casos onde `||` é substituído por `&&`
   - Difíceis de eliminar sem refatorar a lógica
2. **produtoArray - Array vazio (1 mutante):**
   - Condicional `if (false)` em vez de `if (numeros.length === 0)`
3. **Clamp - Operadores de comparação (2 mutantes):**
   - < vs <= e > vs >= em casos limítrofes

Estes mutantes representam menos de **3,3%** do total e são aceitáveis considerando que:

- São casos extremos de otimização
- Não representam bugs reais
- Exigiriam refatoração do código fonte para eliminar
- A pontuação de 96,71% já é considerada excelente

6 Conclusão

6.1 Principais Aprendizados

Este trabalho demonstrou de forma prática e mensurável conceitos fundamentais sobre qualidade de software:

6.1.1 1. Cobertura de Código Qualidade de Testes

O projeto iniciou com 98,64% de cobertura de linhas e 100% de cobertura de funções, métricas que indicariam uma suíte de testes excelente. No entanto, a pontuação de mutação de apenas 73,71% revelou que aproximadamente 1 em cada 4 mutantes (bugs potenciais) não seria detectado.

A cobertura mede se o código foi *executado*. O teste de mutação mede se o código foi *corretamente testado*.

6.1.2 2. Teste de Mutação Revela Fraquezas Ocultas

Os 56 mutantes sobreviventes iniciais apontaram com precisão cirúrgica as fraquezas da suíte de testes:

- Falta de testes para boundary conditions (valores limites)
- Ausência de casos negativos (quando deve retornar false)
- Testes apenas de "caminhos felizes"
- Validações superficiais de exceções
- Fórmulas matemáticas testadas apenas com valores nulos

6.1.3 3. Testes Eficazes Requerem Pensamento Crítico

Não é suficiente escrever um teste que "passa". É necessário questionar:

- Este teste detectaria se o operador `>` fosse trocado por `>=`?
- Testei o caso onde os valores são iguais?
- Testei valores que anulam partes da lógica (como 0 em multiplicações)?
- Testei tanto resultados positivos quanto negativos?
- As asserções são específicas o suficiente?

6.1.4 4. Boundary Conditions São Críticos

A maioria dos mutantes sobreviventes estava relacionada a condições de fronteira:

- `>` vs `>=`
- `<` vs `<=`
- `n === 0` vs `n === 1`
- Arrays vazios vs arrays com um elemento

Testar apenas valores "no meio" deixa estas fronteiras vulneráveis.

6.2 Importância do Teste de Mutação na Engenharia de Software

O teste de mutação não é apenas uma ferramenta acadêmica, mas uma técnica essencial para projetos críticos:

6.2.1 Vantagens Práticas

1. **Detecção de Gaps:** Identifica precisamente onde a suíte de testes é fraca
2. **Melhoria Guiada:** Cada mutante sobrevivente aponta um teste específico a adicionar
3. **Confiança em Refatoração:** Alta pontuação de mutação permite refatorar com segurança
4. **Prevenção de Regressões:** Bugs sutis são detectados antes de chegarem à produção
5. **Cultura de Qualidade:** Eleva o padrão de escrita de testes da equipe

6.2.2 Quando Usar

Teste de mutação é especialmente valioso em:

- Sistemas financeiros (onde bugs podem custar dinheiro)
- Aplicações médicas (onde bugs podem custar vidas)
- Infraestrutura crítica (sistemas de controle, segurança)
- Bibliotecas e frameworks (usados por muitos desenvolvedores)
- Código legado sendo refatorado

6.3 Limitações e Considerações

É importante reconhecer que o teste de mutação tem custos:

- **Tempo de execução:** Cada mutante requer execução completa da suíte
- **Recursos computacionais:** Pode ser pesado para projetos grandes
- **Mutantes equivalentes:** Alguns mutantes não mudam o comportamento real
- **Não substitui revisão de código:** É uma ferramenta complementar

No entanto, para código crítico, estes custos são justificáveis.

6.4 Reflexão Final

Este trabalho transformou a compreensão sobre qualidade de testes. Antes, acreditava-se que alta cobertura era sinônimo de bons testes. Após a experiência prática com teste de mutação, ficou claro que:

> *"Cobertura de código mede se o código foi executado. Teste de mutação mede se o código foi corretamente testado. A diferença é fundamental."*

A melhoria de 73,71% para 96,71% na pontuação de mutação representa não apenas números, mas uma mudança de mentalidade: de "testes que passam" para "testes que realmente protegem o código".

Para projetos onde qualidade é crítica, o teste de mutação não é opcional — é essencial.

6.5 Próximos Passos

Com base nesta experiência, recomenda-se:

1. Integrar teste de mutação no CI/CD do projeto
2. Estabelecer threshold mínimo de mutation score (ex: 95%)
3. Executar mutação apenas em código crítico (otimização de tempo)
4. Treinar a equipe em técnicas de boundary testing
5. Considerar ferramentas como Stryker para outros projetos

7 Referências

1. StrykerJS Official Documentation. Disponível em: https://stryker-mutator.io/docs/
2. Mutation Testing Elements - Supported Mutators. Disponível em: https://stryker-mutator.io/docs/mutation-testing-elements/supported-mutators/
3. Repositório do Projeto Base: https://github.com/CleitonSilvaT/operacoes-mutante
4. Repositório da Solução: https://github.com/LuizNeryy/operacoes-mutante
5. Jest Testing Framework. Disponível em: https://jestjs.io/
6. Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. IEEE transactions on software engineering, 37(5), 649-678.