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

1 Análise Inicial

Este trabalho demonstra a diferença fundamental entre **cobertura de código** e **eficácia de testes** utilizando teste de mutação com StrykerJS. O teste de mutação introduz pequenas alterações (mutações) no código e verifica se os testes as detectam. Mutantes que "sobrevivem" indicam fraquezas na suíte de testes.

1.1 Cobertura de Código vs Mutation Score

A análise inicial revelou uma discrepância crítica entre as métricas tradicionais e a eficácia real:

Tabela 1: Comparação: Cobertura de Código vs Eficácia de Testes

Métrica	**Inicial**	**Final**	**Variação**
Cobertura de Linhas	98,64%	98,64%	0
Cobertura de Funções	100%	100%	0
Mutation Score	**73,71%**	**96,71%**	**+23,00 pp**
Mutantes Mortos	154	203	+49
Mutantes Sobreviventes	56	7	-49
Número de Testes	50	99	+49

Observação Crítica: Apesar de 100% de cobertura de funções e 98,64% de linhas, apenas 73,71% dos mutantes foram detectados inicialmente. Isso prova que **alta cobertura não garante testes eficazes**.

2 Análise de Mutantes Críticos

Foram selecionados 3 mutantes sobreviventes que representam categorias importantes de falhas:

2.1 Mutante #1: Comparadores de Limites

Código Original: `function isMaiorQue(a, b) { return a > b; }`

Mutante Sobrevivente: `a > b` → `a >= b`

Problema: O teste original apenas verificava `isMaiorQue(10, 5)`, que retorna `true` tanto para `>` quanto `>=`. Faltava testar o caso crítico `a === b`.

Solução Implementada:

```
test('deve retornar false quando numeros sao iguais', () => {
    expect(isMaiorQue(5, 5)).toBe(false); // Mata o mutante!
});
```

Este teste força a distinção entre `>` e `>=`, eliminando 6 mutantes relacionados a boundary conditions.

2.2 Mutante #2: Função 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;
}
```

Mutantes Sobreviventes: 8 mutantes, incluindo:

- `n <= 1` → `n < 1`
- `i < n` → `i >= n`
- `n % i` → `n * i`

Problema: O teste apenas verificava `isPrimo(7) === true`, não testando casos limites.
Solução: Testes para todos os casos críticos:

```
expect(isPrimo(0)).toBe(false);  // Testa n <= 1
expect(isPrimo(1)).toBe(false);  // Testa limite
expect(isPrimo(2)).toBe(true);   // Menor primo
expect(isPrimo(4)).toBe(false);  // Par nao-primo
expect(isPrimo(9)).toBe(false);  // Impar composto
```

Resultado: 8 mutantes eliminados.

2.3 Mutante #3: Conversão de Temperatura

Código: `celsiusParaFahrenheit(c) = (c * 9/5) + 32`
Mutantes: `*` → `/`, `/5` → `*5`
Problema Crítico: Testar apenas `celsiusParaFahrenheit(0)` não valida a fórmula, pois `0 * 9/5 = 0` e `0 / 9/5 = 0` produzem o mesmo resultado!
Solução: Testar valores não-zero que validam toda a fórmula:

```
expect(celsiusParaFahrenheit(100)).toBe(212);  // Ebulicao
expect(celsiusParaFahrenheit(-40)).toBe(-40);  // Ponto de igualdade
expect(celsiusParaFahrenheit(37)).toBeCloseTo(98.6, 1); // Temperatura
   corporal
```

Lição Aprendida: Valores que anulam parte da lógica (como 0 em multiplicações) não validam adequadamente a correção da implementação.

3 Resultados Finais

3.1 Evolução da Pontuação

Tabela 2: Evolução em Três Fases

Fase	Mutation Score	Sobreviventes	Testes
Inicial	73,71%	56	50
1ª Melhoria	94,84%	11	92
Final	**96,71%**	**7**	**99**

3.2 Impacto Prático

A melhoria de 73,71% para 96,71% representa:

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

Principais melhorias implementadas:

1. **Boundary conditions:** Testes para valores limites (`a === b`, arrays vazios, zero)
2. **Casos negativos:** Validação de quando funções devem retornar `false`
3. **Mensagens específicas:** `toThrow('mensagem exata')` em vez de `toThrow()`
4. **Múltiplos valores:** Fórmulas matemáticas testadas com valores diversos
5. **Ordenação e edge cases:** Arrays desordenados, valores negativos, casos especiais

3.3 Mutantes Remanescentes

Os 7 mutantes sobreviventes (3,3% do total) são casos extremamente específicos de condições booleanas internas que exigiriam refatoração do código fonte. A pontuação de 96,71% é considerada excelente para produção.

4 Conclusão

4.1 Principais Aprendizados

1. Cobertura Qualidade: O projeto manteve 98,64% de cobertura, mas a eficácia aumentou de 73,71% para 96,71%. Cobertura mede se o código foi *executado*, teste de mutação mede se foi *corretamente testado*.

2. Boundary Conditions são Críticos: A maioria dos mutantes sobreviventes relacionava-se a casos limites: `>` vs `>=`, `n === 0` vs `n === 1`, arrays vazios. Testar apenas valores "no meio" deixa fronteiras vulneráveis.

3. Pensamento Crítico é Essencial: Não basta escrever testes que "passam". É necessário questionar: este teste detectaria se o operador mudasse? Testei valores iguais? Testei resultados negativos?

4. Teste de Mutação Revela Fraquezas Ocultas: Os 56 mutantes iniciais apontaram precisamente onde os testes eram fracos: falta de boundary conditions, ausência de casos negativos, validações superficiais.

4.2 Importância Prática

O teste de mutação é essencial para:

- **Sistemas críticos:** Financeiros, médicos, infraestrutura
- **Refatoração segura:** Alta mutation score permite refatorar com confiança
- **Prevenção de regressões:** Detecta bugs sutis antes da produção
- **Cultura de qualidade:** Eleva o padrão de escrita de testes

4.3 Reflexão Final

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

Este trabalho transformou a compreensão sobre qualidade de testes. A melhoria de 73,71% para 96,71% 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**.

4.4 Recomendações

1. Integrar teste de mutação no CI/CD
2. Estabelecer threshold mínimo (ex: 95%)
3. Focar em código crítico (otimização de tempo)
4. Treinar equipe em boundary testing

Repositório: https://github.com/LuizNeryy/operacoes-mutante
Referências:

1. StrykerJS Documentation: https://stryker-mutator.io/docs/
2. Jia, Y., & Harman, M. (2011). An analysis and survey of mutation testing. IEEE TSE, 37(5), 649-678.