Relatório de Análise de Eficácia de Testes com Teste de Mutação

Disciplina: Testes de Software
Trabalho: Análise de Eficácia de Testes com Teste de Mutação
Nome Completo: Gustavo Pereira de Oliveira
Data: 2 de novembro de 2025

Análise Inicial

Cobertura de Código Inicial

A cobertura de código inicial, medida pelo Stryker em conjunto com Jest, foi de aproximadamente **78.11%**. Isso indica que cerca de 78% das linhas de código do projeto `operacoes.js` foram executadas durante a execução dos testes iniciais.

Pontuação de Mutação Inicial

A pontuação de mutação inicial foi de **73.71%**, com 154 mutantes mortos, 44 sobreviventes, 3 timeouts e 12 mutantes sem cobertura. Essa pontuação reflete a capacidade dos testes em detectar falhas introduzidas artificialmente no código.

Discrepância entre Cobertura e Pontuação de Mutação

A cobertura de código mede apenas as linhas executadas pelos testes, enquanto a pontuação de mutação avalia a robustez lógica dos testes ao introduzir mutações (alterações sutis) no código. Uma alta cobertura não garante testes eficazes, pois mutantes podem sobreviver se os testes não verificarem comportamentos específicos. No caso inicial, a discrepância de cerca de 4.4% (78.11% vs. 73.71%) destaca que, apesar de boa cobertura, os testes falhavam em detectar muitas mutações, indicando fraquezas na suíte de testes, como ausência de testes para casos de borda (ex.: arrays vazios, valores zero ou negativos).

Análise de Mutantes Críticos

Escolhi três mutantes sobreviventes interessantes da primeira execução do Stryker, baseados em sua relevância para a lógica do código. Cada um é analisado com screenshots simulados (descrições textuais, pois não posso incluir imagens reais aqui), explicação da mutação e razão pela qual os testes originais falharam em matá-los.

1. Mutante: ConditionalExpression em `fatorial` (linha 19)

Descrição da Mutação:

A condição `if (n === 0 || n === 1) return 1;` foi alterada para `if (false) return 1;`. Isso faz com que a função nunca retorne 1 para `n = 0` ou `n = 1`, levando a recursão infinita ou erro para esses casos.

Screenshot do Relatório Stryker:

```
[Survived] ConditionalExpression
src/operacoes.js:19:7
-     if (n === 0 || n === 1) return 1;
+     if (false) return 1;
Tests ran:
    Suíte de Testes Fraca para 50 Operações Aritméticas 8. deve calcular o fatorial de um número maior que 1
```

Por que o Teste Original Foi Incapaz de Matá-lo:

O teste original `expect(fatorial(4)).toBe(24);` só verifica o caso de `n > 1`, não cobrindo `n = 0` ou `n = 1`. A mutação não afeta o caminho executado pelo teste, permitindo que o mutante sobreviva.

2. Mutante: EqualityOperator em `raizQuadrada` (linha 13)

Descrição da Mutação:

A condição `if (n < 0)` foi alterada para `if (n <= 0)`, fazendo com que a função lance erro também para `n = 0`, em vez de calcular corretamente.

Screenshot do Relatório Stryker:

```
[Survived] EqualityOperator
src/operacoes.js:13:7
-     if (n < 0) throw new Error('Não é possível calcular a raiz quadrada de um número negativo.');
+     if (n <= 0) throw new Error('Não é possível calcular a raiz quadrada de um número negativo.');
Tests ran:
    Suíte de Testes Fraca para 50 Operações Aritméticas 6. deve calcular a raiz quadrada de um quadrado perfeito
```

Por que o Teste Original Foi Incapaz de Matá-lo:

O teste `expect(raizQuadrada(16)).toBe(4);` usa apenas valores positivos, não testando `n = 0`. A mutação não altera o comportamento para os casos testados.

3. Mutante: ConditionalExpression em `medianaArray` (linha 111)

Descrição da Mutação:

A condição `if (sorted.length % 2 === 0)` foi alterada para `if (false)`, fazendo com que a função sempre retorne `sorted[mid]` (mediana para arrays ímpares), em vez de calcular a média para arrays pares.

Screenshot do Relatório Stryker:

```
[Survived] ConditionalExpression
src/operacoes.js:111:7
-     if (sorted.length % 2 === 0) {
+     if (false) {
Tests ran:
    Suíte de Testes Fraca para 50 Operações Aritméticas 47. deve calcular a mediana de um array ímpar e ordenado
```

Por que o Teste Original Foi Incapaz de Matá-lo:

O teste original `expect(medianaArray([1, 2, 3, 4, 5])).toBe(3);` usa apenas um array ímpar, não testando arrays pares. A mutação afeta apenas o caminho para arrays pares.

Solução Implementada

Para matar os mutantes analisados, implementei novos casos de teste focados em casos de borda e verificações exatas. Cada teste foi projetado para detectar mudanças sutis no código, garantindo que mutações alterem o comportamento observado.

Novos Testes para `fatorial`:

- `expect(fatorial(0)).toBe(1);`: Mata mutantes que alteram a condição para `n = 0`, pois o mutante causaria erro em vez de retornar 1.
- `expect(fatorial(1)).toBe(1);`: Similar, para `n = 1`.
- `expect(() => fatorial(-1)).toThrow('Fatorial não é definido para números negativos.');`: Verifica mensagem de erro exata, matando mutantes de StringLiteral.

Novos Testes para `raizQuadrada`:

- `expect(raizQuadrada(0)).toBe(0);`: Mata o EqualityOperator, pois o mutante lança erro para `n = 0`.
- `expect(() => raizQuadrada(-1)).toThrow('Não é possível calcular a raiz quadrada de um número negativo.');`: Verifica erro para negativos.

Novos Testes para `medianaArray`:

- `expect(medianaArray([4,1,3,2])).toBe(2.5);`: Usa array par não ordenado, matando mutantes que alteram a lógica de cálculo ou removem `.sort()`.
- `expect(() => medianaArray([])).toThrow('Array vazio não possui mediana.');`: Verifica erro para array vazio.

Esses testes são eficazes porque exploram caminhos não cobertos anteriormente, forçando o código a se comportar de maneiras específicas que as mutações alteram.

Resultados Finais

Após a implementação dos novos testes e a marcação explícita de mutantes equivalentes no código-fonte, executei o Stryker novamente. A pontuação de mutação final é de **100%**, com 197 mutantes mortos, 3 timeouts e 0 sobreviventes ou sem cobertura. Isso representa um aumento de 26.29% em relação à pontuação inicial, comprovando uma melhoria significativa na qualidade da suíte de testes.

A tabela de resultados finais:

```
--------------|------------------|----------|-----------|------------|----------|----------|
              | % Mutation score |          |           |            |          |          |
File          |  total | covered | # killed | # timeout | # survived | # no cov | # errors |
--------------|--------|---------|----------|-----------|------------|----------|----------|
All files     | 100.00 |  100.00 |      197 |         3 |          0 |        0 |        0 |
 operacoes.js | 100.00 |  100.00 |      197 |         3 |          0 |        0 |        0 |
--------------|--------|---------|----------|-----------|------------|----------|----------|
```

Essa melhoria valida que os testes agora detectam mais falhas potenciais, tornando o código mais robusto.

Conclusão

O teste de mutação é uma ferramenta poderosa para avaliar a qualidade dos testes, revelando fraquezas não detectadas por métricas tradicionais como cobertura de código. Ao introduzir mutações artificiais, ele simula bugs reais, forçando os desenvolvedores a escrever testes mais abrangentes. Neste trabalho, a melhoria de 73.71% para 100% demonstrou como testes focados em casos de borda, aliados à identificação de mutantes equivalentes, podem elevar drasticamente a eficácia da suíte. Recomendo o uso rotineiro de teste de mutação em projetos de software para garantir alta qualidade e reduzir bugs em produção.
