Relatório: Análise de Eficácia de Testes com Teste de Mutação

Disciplina: TESTE DE SOFTWARE

Trabalho: Análise de Eficácia de Testes com Teste de Mutação

Aluno: Lúcio Alves Almeida Neto

Matrícula: 865809

1.

Análise Inicial

Qual foi a cobertura de código inicial reportada pelo Jest?

R: Após executar o comando `npm test -- --coverage`, o relatório de cobertura de código indicou uma cobertura inicial de `96.71%`.

P: Qual foi a pontuação de mutação (Mutation Score) inicial reportada pelo StrykerJS?

R: A primeira execução do StrykerJS com o comando `npx stryker run` resultou em uma pontuação de mutação inicial de 73.71%.

P: Houve discrepância entre a cobertura de código inicial e a pontuação de mutação inicial? Por quê? R: Sim, houve uma grande discrepância. Enquanto a cobertura de código era alta (`96.71%`), indicando que quase todo o código era *executado* pelos testes, a pontuação de mutação (73.71%) foi significativamente menor. Isso demonstra a limitação da métrica de cobertura de código: ela não garante a eficácia dos testes. Um teste pode passar por uma linha de código, mas não verificar adequadamente seu comportamento ou falhar ao detectar bugs sutis. O StrykerJS introduziu 213 mutantes válidos (`totalValid` = `killed` + `timeout` + `survived` + `noCoverage` = 154 + 3 + 44 + 12) e os testes iniciais só detectaram 157 deles (`totalDetected` = `killed` + `timeout` = 154 + 3), deixando 44 mutantes sobreviverem dentro do código coberto e 12 em código não coberto, evidenciando as fraquezas da suíte de testes original, apesar da alta cobertura.

2.

Análise de Mutantes Críticos

```
function divisao(a, b) {
  if (b === 0) throw new Error('Divisão por zero não é permitida.');
  return a / b;
}
function potencia(base, expoente) { return Math.pow(base, expoente); }
function raizQuadrada(n) {
  if (n < 0) throw new Error('Não é possível calcular a raiz quadrada de um número negativo.');
  return Math.sqrt(n);
}
function restoDivisao(dividendo, divisor) { return dividendo % divisor; }
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;
}
```

Exemplo de Mutante 1 (Função `raizQuadrada`)

P: Qual mutante sobreviveu e onde? R: Um dos mutantes que sobreviveu (indicado pelos pontos laranja na Figura 1) foi um `ConditionalExpression` na linha 13 do arquivo `src/operacoes.js`, dentro da função `raizQuadrada`.

P: O que essa mutação fez no código? R: A mutação alterou a condição `if (n < 0)` para `if (false)`. Isso significa que a verificação que impede o cálculo da raiz quadrada para números negativos (e lança um erro) foi efetivamente removida pela mutação.

P: Por que o teste original foi incapaz de matá-la? R: O teste original para `raizQuadrada` (`test('6. deve calcular a raiz quadrada de um quadrado perfeito', () => { expect(raizQuadrada(16)).toBe(4); });`) usava apenas a entrada `n = 16`. Como 16 não é menor que `0`, a condição `if (n < 0)` já era falsa no código original. Ao ser substituída por `if (false)`, o comportamento da função para a entrada 16 permaneceu inalterado (continuou calculando `Math.sqrt(16)`). O teste não incluía um caso para verificar o comportamento da função com entradas negativas (por exemplo, `raizQuadrada(-1)`), que é exatamente o cenário que essa condição protege.

Exemplo de Mutante 2 (Função `fatorial`)

P: Qual mutante sobreviveu e onde? R: Outro mutante sobrevivente (pontos laranja na Figura 1) foi um `ConditionalExpression` na linha 18 do arquivo `src/operacoes.js`, na função `fatorial`.

P: O que essa mutação fez no código? R: Similar ao caso anterior, a mutação trocou a condição `if (n < 0)` por `if (false)`, removendo a validação que lança um erro quando se tenta calcular o fatorial de um número negativo.

P: Por que o teste original foi incapaz de matá-la? R: O teste original (`test('8. deve calcular o fatorial de um número maior que 1', () => { expect(fatorial(4)).toBe(24); });`) usava apenas a entrada `n = 4`. Como 4 não é negativo, a condição `if (n < 0)` já era falsa. A mutação para `if (false)` não mudou o fluxo do programa para essa entrada específica. Faltava um teste que verificasse se a função lançava corretamente o erro para entradas como `fatorial(-1)`.

Exemplo de Mutante 3 (Função `fatorial`)

P: Qual mutante sobreviveu e onde? R: Um mutante `LogicalOperator` (indicado pelos pontos laranja na Figura 1) sobreviveu na linha 19 da função `fatorial`.

P: O que essa mutação fez no código? R: A mutação alterou a condição `if (n === 0 || n === 1)` para `if (n === 0 && n === 1)`. A condição original retorna `true` se n for `0` OU 1, fazendo a função retornar 1 imediatamente nesses casos. A condição mutada só seria `true` se n fosse `0` E 1 ao mesmo tempo, o que é impossível, tornando-a efetivamente `false`.

P: Por que o teste original foi incapaz de matá-la? R: O teste original usava `fatorial(4)`, onde a condição `n === 0 || n === 1` já era falsa. Mesmo que tivéssemos testes para `fatorial(0)` e `fatorial(1)`, este mutante *ainda sobreviveria* (tornando-se um mutante equivalente para esses casos). Isso ocorre porque, se a condição mutada (`false`) for avaliada, o código prossegue para o loop `for (let i = 2; i <= n; i++)`. Para n=0 ou n=1, o loop não executa nenhuma vez, e a função retorna o valor inicial de `resultado`, que é 1 – o mesmo valor que seria retornado pelo `if` original. Como o resultado final não muda para as entradas 0 e 1, nenhum teste consegue detectar essa mutação específica.

3.Solução Implementada

P: Quais novos casos de teste foram escritos para matar os mutantes analisados (exemplificados na Seção 2)?

R:

Para o Mutante da `raizQuadrada` (Linha 13 - `ConditionalExpression`): Foi adicionado um teste que verifica especificamente se a função lança o erro esperado ao receber uma entrada negativa:
JavaScript
expect(() => raizQuadrada(-1)).toThrow('Não é possível calcular a raiz quadrada de um número negativo.');

-

Para o Mutante do `fatorial` (Linha 18 - `ConditionalExpression`): Similarmente, foi adicionado um teste para verificar o lançamento do erro com entrada negativa:
JavaScript
expect(() => fatorial(-1)).toThrow('Fatorial não é definido para números negativos.');

-

Para o Mutante do `fatorial` (Linha 19 - `LogicalOperator`): Embora identificado como provavelmente equivalente, testes adicionais para os casos base `fatorial(0)` e `fatorial(1)` já haviam sido incluídos em etapas anteriores para garantir a robustez, mesmo que não matem especificamente essa mutação:
JavaScript
expect(fatorial(1)).toBe(1);

expect(fatorial(0)).toBe(1);

- (Observação: Nenhum teste adicional foi criado especificamente para este mutante, pois ele foi classificado como equivalente).

P: Por que esses novos testes são eficazes (ou porque não foram adicionados mais testes)?

R:

- Testes de Erro (`toThrow`): Os testes adicionados para `raizQuadrada(-1)` e `fatorial(-1)` são eficazes porque visam diretamente a condição mutada. Se a mutação (`ConditionalExpression` para `false`) remover a checagem `n < 0`, a função não lançará mais o erro esperado para entradas negativas. Isso causa uma falha no teste `toThrow`, "matando" o mutante.
- Mutante Equivalente (`fatorial` Linha 19): O mutante que troca `||` por && na linha 19 (`if (n === 0 || n === 1)`) não altera o resultado final da função para as entradas 0 ou 1, pois a lógica do loop `for` subsequente já lida corretamente com esses casos, retornando 1. Como o comportamento observável não muda, nenhum teste consegue diferenciá-lo do código original, classificando-o como um mutante equivalente. Por isso, nenhum teste adicional foi focado em matá-lo.

(Observação Geral: Outros testes foram adicionados durante o processo de melhoria para cobrir casos de borda e alternativos em diversas outras funções (ex: `mediaArray([])`, `isPar(7)`, `isImpar(100)`, `clamp` nos limites, `mdc` com zero, `isPrimo` com 0/1/não-primos, `produtoArray([])`, `medianaArray` com arrays pares/não ordenados), contribuindo para o aumento geral da pontuação de mutação, mesmo que não diretamente relacionados aos exemplos específicos desta seção.)

4.

Resultados Finais

P: Qual foi a pontuação de mutação final alcançada após a melhoria da suíte de testes? R: Após adicionar os novos casos de teste e executar `npx stryker run` novamente, a pontuação de mutação final alcançada foi de 96%.

P: Como isso comprova a melhoria na qualidade da suíte de testes?

R: O aumento significativo da pontuação de mutação (de 73.71% para 96%) demonstra que a suíte de testes agora é muito mais eficaz em detectar bugs sutis. Muitos dos 44 mutantes que originalmente sobreviveram foram "mortos" pelos novos testes, indicando que as fraquezas identificadas foram corrigidas.

P: Por que a pontuação não atingiu 100% (ou a meta de 98%)?

R: Alguns mutantes nas linhas 19 (`fatorial`), 84 (`produtoArray`), 88 e 89 (`clamp`) sobreviveram. Uma análise detalhada indica que estes são mutantes equivalentes. As alterações introduzidas por eles (remover checagens redundantes ou trocar < por <= e >

por >= nos limites do `clamp`) não alteram o comportamento observável das funções para as entradas testadas (e provavelmente para nenhuma entrada). Portanto, não é possível criar testes que falhem especificamente para essas mutações sem alterar a lógica original do código.

5.

Conclusão

P: Qual a importância do teste de mutação como ferramenta de avaliação da qualidade de testes?

R: Este trabalho demonstrou que a cobertura de código, isoladamente, é uma métrica insuficiente para garantir a qualidade de uma suíte de testes. O Teste de Mutação, ao introduzir pequenas falhas (mutações) no código e verificar se os testes existentes as detectam, oferece uma avaliação muito mais precisa da eficácia dos testes. Ele ajuda a identificar casos de borda não testados, asserções fracas e cenários não cobertos, guiando o desenvolvedor na criação de testes mais robustos e confiáveis. Embora mutantes equivalentes possam impedir a obtenção de 100%, o processo de análise e melhoria baseado na pontuação de mutação eleva significativamente a capacidade da suíte de testes de encontrar bugs reais.

```
    √ 9. deve calcular a média de um array com múltiplos elementos
    √ 10. deve somar um array com múltiplos elementos (1 ms)
    √ 11. deve encontrar o valor máximo em um array
    √ 12. deve encontrar o valor mínimo em um array (1 ms)
    √ 13. deve retornar o valor absoluto de um número negativo
    √ 14. deve arredondar um número para cima
    √ 15. deve retornar true para um número par
    √ 16. deve retornar true para um número ímpar
    √ 17. deve calcular uma porcentagem simples
    √ 18. deve aumentar um valor em uma porcentagem (4 ms)
    √ 19. deve diminuir um valor em uma porcentagem
    √ 20. deve inverter o sinal de um número positivo
    √ 21. deve calcular o seno de 0
    √ 22. deve calcular o cosseno de 0
    √ 23. deve calcular a tangente de 0
    √ 24. deve calcular o logaritmo natural de Euler
    √ 25. deve calcular o logaritmo na base 10
    √ 26. deve arredondar para baixo
    √ 27. deve arredondar para cima
    √ 28. deve calcular a hipotenusa de um triângulo retângulo
    √ 29. deve converter graus para radianos
    √ 30. deve converter radianos para graus (1 ms)
    √ 31. deve calcular o MDC de dois números
    √ 32. deve calcular o MMC de dois números
    √ 33. deve verificar que um número é primo
    √ 34. deve calcular o 10º termo de Fibonacci
    √ 35. deve calcular o produto de um array
    √ 36. deve manter um valor dentro de um intervalo (clamp)
    √ 37. deve verificar se um número é divisível por outro
    √ 38. deve converter Celsius para Fahrenheit
    √ 39. deve converter Fahrenheit para Celsius
    √ 40. deve calcular o inverso de um número
    √ 41. deve calcular a área de um círculo
    √ 42. deve calcular a área de um retângulo
    √ 43. deve calcular o perímetro de um retângulo
    √ 44. deve verificar se um número é maior que outro
    √ 45. deve verificar se um número é menor que outro (1 ms)
    √ 46. deve verificar se dois números são iguais
    √ 47. deve calcular a mediana de um array ímpar e ordenado
    √ 48. deve calcular o dobro de um número
    √ 49. deve calcular o triplo de um número
    √ 50. deve calcular a metade de um número (1 ms)

--------------|---------|----------|---------|---------|-------------------
File          | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
--------------|---------|----------|---------|---------|-------------------
All files     |   85.41 |    58.82 |     100 |   98.64 |
 operacoes.js |   85.41 |    58.82 |     100 |   98.64 | 112
--------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       50 passed, 50 total
Snapshots:   0 total
Time:        0.734 s, estimated 1 s
Ran all test suites.
PS C:\Users\kingl\OneDrive\Área de Trabalho\Atividade-de-testes\test-mutacao-lucio\operacoes-mutante-main>
```

```
[Survived] ConditionalExpression
src/operacoes.js:19:7
-     if (n === 0 || n === 1) return 1;
+     if (false || n === 1) return 1;
Tests ran:
    Suíte de Testes Aprimorada para 50 Operações Aritméticas 8. deve calcular o fatorial, tratar casos base e lançar erro para negativo


[Survived] ConditionalExpression
src/operacoes.js:19:18
-     if (n === 0 || n === 1) return 1;
+     if (n === 0 || false) return 1;
Tests ran:
    Suíte de Testes Aprimorada para 50 Operações Aritméticas 8. deve calcular o fatorial, tratar casos base e lançar erro para negativo


[Survived] ConditionalExpression
src/operacoes.js:84:7
-     if (numeros.length === 0) return 1;
+     if (false) return 1;
Tests ran:
    Suíte de Testes Aprimorada para 50 Operações Aritméticas 35. deve calcular o produto de um array e tratar array vazio


[Survived] EqualityOperator
src/operacoes.js:88:7
-     if (valor < min) return min;
+     if (valor <= min) return min;
Tests ran:
    Suíte de Testes Aprimorada para 50 Operações Aritméticas 36. deve manter um valor dentro de um intervalo (clamp) e testar limites


[Survived] EqualityOperator
src/operacoes.js:89:7
-     if (valor > max) return max;
+     if (valor >= max) return max;
Tests ran:
    Suíte de Testes Aprimorada para 50 Operações Aritméticas 36. deve manter um valor dentro de um intervalo (clamp) e testar limites


Ran 1.54 tests per mutant on average.
-------------|------------------|----------|----------|-----------|----------|----------|
             | % Mutation score |          |          |           |          |          |
File         |  total | covered | # killed | # timeout | # survived | # no cov | # errors |
-------------|--------|---------|----------|-----------|------------|----------|----------|
All files    |  96.71 |   96.71 |      203 |         3 |          7 |        0 |        0 |
 operacoes.js |  96.71 |   96.71 |      203 |         3 |          7 |        0 |        0 |
-------------|--------|---------|----------|-----------|------------|----------|----------|
23:16:37 (20148) INFO HtmlReporter Your report can be found at: file:///C:/Users/kingl/OneDrive/%C3%81rea%20de%20Trabalho/Atividade-de-testes/test-m
ain/reports/mutation/mutation.html
23:16:37 (20148) INFO MutationTestExecutor Done in 21 seconds.
PS C:\Users\kingl\OneDrive\Área de Trabalho\Atividade-de-testes\test-mutacao-lucio\operacoes-mutante-main>
```