PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS E INFORMÁTICA
BACHARELADO EM ENGENHARIA DE SOFTWARE

MATHEUS NOLASCO MIRANDA SOARES

MATRÍCULA: 688826

Análise de Eficácia de Testes com Teste de Mutação

Testes de Software

Análise Inicial

Cobertura de Código Inicial

A execução inicial da suíte de testes apresentou os seguintes resultados de cobertura de código:

- Statements: 85.41%
- Branch: 58.82%
- Funcs: 100%
- Lines: 98.64%

Pontuação de Mutação Inicial

Ao executar o Stryker pela primeira vez, foram gerados 213 mutantes, com os seguintes resultados:

- Mutation Score (total): 73.71%
- Mutation Score (covered): 78.11%
- Mutantes mortos: 154
- Mutantes sobreviventes: 44
- Mutantes sem cobertura: 12
- Timeout: 3
- Erros: 0

Discrepância entre Cobertura de Código e Pontuação de Mutação

Observa-se uma discrepância significativa entre a alta cobertura de linhas (98.64%) e a pontuação de mutação (73.71%). Essa diferença de aproximadamente 25 pontos percentuais revela uma limitação importante das métricas tradicionais de cobertura de código.

Embora 98.64% das linhas de código sejam executadas durante os testes, isso não garante que essas linhas estejam adequadamente testadas. A cobertura de código mede apenas se uma linha foi executada, mas não verifica se:

1. Todas as condições foram testadas (Branch Coverage de apenas 58.82% confirma isso)
2. Os valores de retorno são validados corretamente
3. Casos extremos (edge cases) são cobertos
4. Tratamentos de erro são eficazes

Os 44 mutantes sobreviventes e 12 mutantes sem cobertura indicam que a suíte de testes possui lacunas qualitativas. Testes que apenas "passam" pelo código não são suficientes, é necessário que eles validem comportamentos específicos para garantir robustez.

Análise de Mutantes Críticos

Mutante 1: Remoção da Ordenação na Função medianaArray

Código Original:

```
const sorted = [...numeros].sort((a, b) => a - b);
```

Mutação Aplicada:

```
const sorted = [...numeros];
```

Teste que executou:

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

Por que o mutante sobreviveu:

O Stryker removeu a chamada ao método .sort(), eliminando a ordenação do array. No entanto, o teste original utilizava um array já ordenado [1, 2, 3, 4, 5]. Como o array estava previamente em ordem crescente, a ausência da ordenação não alterou o resultado final, ou seja, a mediana continuou sendo 3.

Problema Qualitativo:

Este mutante revela uma falha crítica na estratégia de teste: o teste não valida a funcionalidade completa da função. A ordenação é essencial para o cálculo correto da mediana quando o array está desordenado, mas o teste original não cobriu esse cenário. Isso demonstra que o teste estava apenas verificando se a função retornava o valor correto para um caso específico, mas não se o algoritmo estava correto.

Mutante 2: Alteração da Condição de Validação em fatorial

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:

```
if (n <= 0) throw new Error('Fatorial não é definido para números negativos.');
```

Teste que executou:

```
test('8. deve calcular o fatorial de um número maior que 1', () => { expect(fatorial(4)).toBe(24); });
```

Por que o mutante sobreviveu:

O Stryker alterou o operador de comparação de < para <=, fazendo com que a condição agora também seja verdadeira para n = 0. Isso significa que fatorial(0) passaria a lançar uma exceção em vez de retornar 1, o que está matematicamente incorreto (por definição, 0! = 1).

O teste original apenas validava o cálculo para n = 4, sem cobrir o caso base n = 0. Como o teste não exercitava esse cenário limite, o mutante sobreviveu.

Problema Qualitativo:

Este mutante expõe a ausência de testes de casos base e fronteira. Funções matemáticas frequentemente possuem casos especiais (como 0! = 1, 1! = 1) que precisam ser testados explicitamente. Sem esses testes, a suíte não garante que o comportamento está correto para todos os inputs válidos, apenas para alguns casos "normais".

Mutante 3: Retorno Constante true na Função isPar

Código Original:

```
function isPar(n) { return n % 2 === 0; }
```

Mutação Aplicada:

```
function isPar(n) { return true; }
```

Teste que executou:

```
test('15. deve retornar true para um número par', () => { expect(isPar(10)).toBe(true); });
```

Por que o mutante sobreviveu:

O Stryker substituiu toda a lógica de verificação por um retorno constante true. O teste original apenas verificava se a função retornava true para um número par (10). Como o mutante sempre retorna true, o teste passou sem detectar o problema.

Problema Qualitativo:

Este é um exemplo clássico de teste unilateral. O teste valida apenas o caso positivo (número par), mas não o caso negativo (número ímpar). Uma função de verificação booleana precisa de, no mínimo, dois testes:

- Um para quando a condição é verdadeira
- Um para quando a condição é falsa

Sem o teste do caso negativo, a suíte não garante que a lógica de verificação esteja implementada corretamente, apenas que ela retorna `true` quando deveria, mas não verifica se retorna `false` quando deveria.

Identificação dos Testes que Mataram os Mutantes

Mutante 1: Remoção da Ordenação em medianaArray

Testes que mataram este mutante:

```
test('NOVO - 47.3 deve calcular a mediana de um array ímpar não ordenado', () => { expect(medianaArray([5, 1, 3, 2, 4])).toBe(3);});

test('NOVO - 47.4 deve calcular a mediana de um array par não ordenado', () => { expect(medianaArray([4, 1, 3, 2])).toBe(2.5);});

test('NOVO - 47.5 deve calcular a mediana corretamente com números negativos e positivos não ordenados', () => { expect(medianaArray([-2, 4, 1, 3])).toBe(2); });
```

Por que esses testes mataram o mutante:
Os testes 47.3, 47.4 e 47.5 utilizam arrays não ordenados. Quando o mutante remove o `.sort()`, esses arrays permanecem desordenados e o cálculo da mediana retorna valores incorretos, fazendo os testes falharem.

Mutante 2: Alteração de n < 0 para n <= 0 em fatorial

Teste que matou este mutante:

```
test('NOVO - 8.2 deve retornar 1 para n = 0', () => {
expect(fatorial(0)).toBe(1); });
```

Por que esse teste matou o mutante:
O teste passa n = 0 e espera que a função retorne 1 (pois 0! = 1 por definição matemática).

- Código original: n < 0 é false para n = 0 → não lança erro → continua e retorna 1
- Mutante: n <= 0 é true para n = 0 → lança erro

Como o teste espera o valor `1` mas o mutante lança uma exceção, o teste falha e mata o mutante.

Mutante 3: Retorno Constante true em isPar

Teste que matou este mutante:

```
test('NOVO - 15.1 deve retornar false para um número ímpar', () => { expect(isPar(3)).toBe(false); });
```

Por que esse teste matou o mutante:
O teste passa um número ímpar (3) e espera false como retorno. Quando o mutante substitui a lógica por return true, o teste recebe true em vez de false, falhando e matando o mutante.

Resultados Finais

Pontuação de Mutação Final

Após a implementação dos novos casos de teste, a execução final do Stryker apresentou os seguintes resultados:

- Mutation Score (total): 97.65%
- Mutation Score (covered): 97.65%
- Mutantes mortos: 205
- Mutantes sobreviventes: 5
- Timeout: 3
- Sem cobertura: 0
- Erros: 0

Comparação e Melhoria Obtida

Métrica	Inicial	Final	Melhoria
Mutation Score	73.71 %	97.65 %	+23.94%
Mutantes mortos	154	205	+51 mutantes
Mutantes sobreviventes	44	5	-39 mutantes
Sem cobertura	12	0	-12 mutantes

A suíte de testes passou de 73.71% para 97.65% de pontuação de mutação, representando uma melhoria de 23.94 pontos percentuais. Isso significa que a qualidade da suíte de testes foi significativamente aprimorada, eliminando 39 dos 44 mutantes sobreviventes originais e cobrindo todos os 12 mutantes que estavam sem cobertura.

Por Que Não Foi Possível Atingir 98%?

Apesar dos esforços para criar testes adicionais, 5 mutantes permaneceram vivos. Após análise detalhada, foi constatado que esses mutantes são mutantes equivalentes — ou seja, mutações que não alteram o comportamento observável do código para nenhum input possível.

Mutantes Equivalentes Identificados

1. Mutantes na função fatorial (linha 19):

```
if (n === 0 || n === 1) return 1;

let resultado = 1;

for (let i = 2; i <= n; i++) { resultado *= i; }
```

Mutações sobreviventes:

- `if (false) return 1;`
- `if (n === 0 && n === 1) return 1;`
- `if (false || n === 1) return 1;`
- `if (n === 0 || false) return 1;`

Por que são equivalentes:
O loop for (let i = 2; i <= n; i++) nunca executa para n = 0 ou n = 1, pois a condição 2 <= 0 e 2 <= 1 é falsa desde o início. Portanto, mesmo sem o if, a função retorna resultado = 1 (valor inicial). A verificação explícita if (n === 0 || n === 1) return 1; é redundante para o comportamento final.

2. Mutante na função produtoArray (linha 84):

```
if (numeros.length === 0) return 1;

return numeros.reduce((acc, val) => acc * val, 1);
```

Mutação sobrevivente:

- `if (false) return 1;`

Por que é equivalente:
O método reduce com valor inicial 1 já retorna 1 para arrays vazios sem executar o callback. A verificação if (numeros.length === 0) return 1; é redundante porque o comportamento é idêntico com ou sem ela.

Conclusão dos Resultados

O teste de mutação, vale destacar, revelou ser uma ferramenta indispensável para avaliar a verdadeira qualidade de uma suíte de testes. Este trabalho demonstrou que as métricas tradicionais de cobertura de código, embora úteis, contam apenas parte da história. A cobertura mede execução. O teste de mutação, por outro lado, mede validação.

A discrepância observada entre 98.64% de cobertura de linhas e 73.71% de pontuação de mutação deixou uma lição clara: executar código não é o mesmo que testá-lo. É possível ter alta

cobertura e, ao mesmo tempo, ter testes frágeis que não detectam defeitos reais. O teste de mutação expõe essas fragilidades ao simular erros de programação e verificar se os testes conseguem identificá-los.

Durante este trabalho, ficou evidente que o teste de mutação funciona como um guia objetivo. Ele aponta lacunas específicas, como a ausência de testes para casos negativos, casos extremos ou validações de algoritmos. A propósito, essa precisão elimina o aspecto subjetivo da criação de testes. Em vez de adivinhar o que testar, o desenvolvedor recebe feedback concreto sobre onde a suíte falha.

Vale lembrar, entretanto, que o teste de mutação também tem limitações. Mutantes equivalentes existem e são inevitáveis em códigos com redundâncias. Isso, na prática, transforma a ferramenta em algo além de validação de testes: ela se torna um indicador de qualidade do próprio código de produção.

A melhoria de 73.71% para 97.65% obtida neste trabalho comprova o valor prático da abordagem. Em ambientes reais, onde bugs podem ter custos elevados, o teste de mutação oferece confiança de que a suíte de testes é uma proteção real contra regressões. Não se trata de substituir outras práticas de qualidade, mas de complementá-las de forma crucial. O teste de mutação garante que os testes façam o que prometem: proteger o software.