Análise de Eficácia de Testes com Teste de Mutação

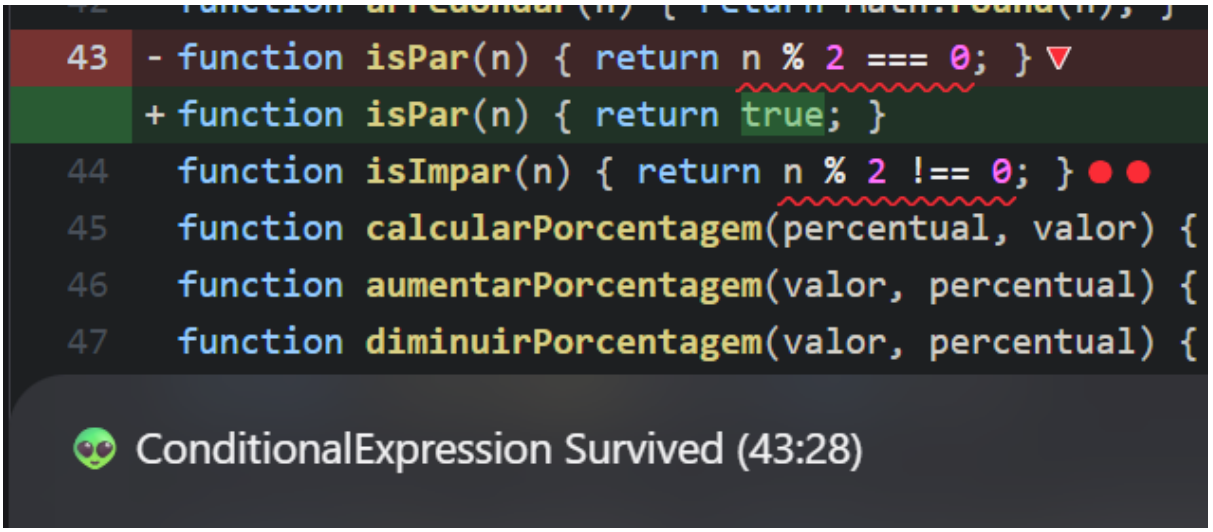
PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS - PUC MINAS
DISCIPLINA: TESTE DE SOFTWARE
ALUNO: ALFREDO LUIS VIEIRA
MATRÍCULA: 807165

1 - Situação inicial

Atualmente, antes de realizar intervenções nos testes, encontramos uma taxa de mutantes mortos de 73.71%, o que equivale a 154 mortos dos 213 totais, e sua cobertura foi de 100%. Estes valores discrepantes tendem a ser um indicio sobre a qualidade inicial dos testes, ou seja, com 100% de cobertura, todas as linhas são testadas, mas a taxa de apenas 73% nos diz que a verificação do comportamento destas linhas não foi muito rigorosa, o que nos leva a questionar a qualidade dos testes implementados inicialmente.

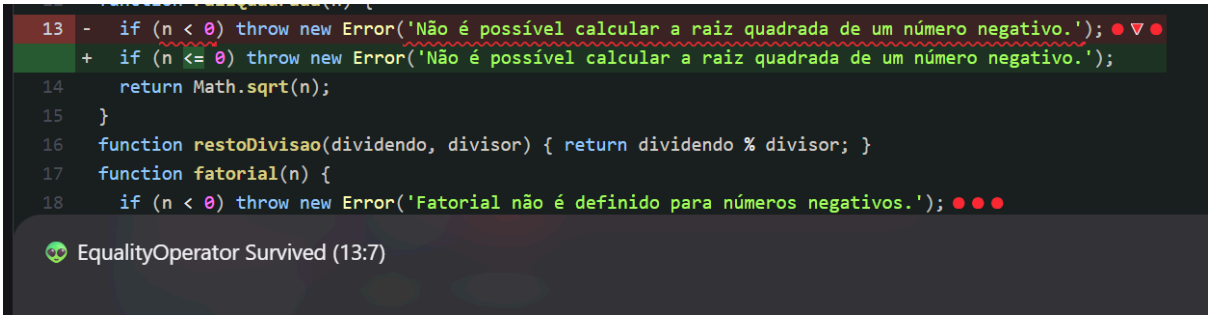
2 - Análise de Mutantes Críticos

Inicialmente, encontramos alguns mutantes críticos, mutantes estes que sobreviveram aos testes iniciais, sendo eles, o mutante, **ID 63**, sobreviveu devido a um teste que validava apenas o "caminho feliz". Este mutante foi aplicado à função isPar (linha 43), onde o Stryker alterou drasticamente a lógica de return n % 2 === 0; para simplesmente return true;. O teste original apenas verificava se isPar(100) retornava true, o que acontecia tanto no código original quanto no código mutado. Como o teste passou em ambos os cenários, ele foi incapaz de detectar que a função agora retornava true para *todas* as entradas, incluindo números ímpares. O mutante sobreviveu porque faltava um teste negativo, como expect(isPar(7)).toBe(false), que teria falhado imediatamente.



O segundo mutante, **ID 15**, sobreviveu por uma razão diferente: a falha em testar caminhos de erro. Na função raizQuadrada (linha 13), o Stryker removeu completamente a verificação

de segurança, trocando if (n < 0) por if (false). O teste original apenas validava uma entrada positiva, raizQuadrada(16). Como esse teste nunca executava a linha que continha o if, a condição de erro nunca era validada. O Stryker pôde, portanto, remover essa verificação de segurança sem que o teste falhasse. Isso expôs uma falha crítica, pois a suíte de testes não garantia que a função impediria o cálculo da raiz de um número negativo. A solução foi adicionar um teste que força a execução desse caminho, esperando que um erro seja lançado: expect(() => raizQuadrada(-1)).toThrow(...).



3 - Solução Implementada

Para "matar" os mutantes que sobreviveram à suíte de testes inicial, a estratégia foi fortalecer os testes existentes, adicionando asserções que cobrissem os cenários negligenciados. A suíte de testes original focava em "caminhos felizes", portanto, as principais correções envolveram a adição de testes para casos de borda, caminhos de erro e validações de resultados negativos.

Primeiramente, para funções que deveriam falhar com entradas inválidas, como `raizQuadrada` e `inverso`, foram adicionados testes `expect().toThrow()`. Isso garantiu que `raizQuadrada(-1)` e `inverso(0)` lançassem os erros corretos, matando mutantes que tentavam remover essas verificações de segurança.

Para funções recursivas ou com loops, como `fatorial`, a fraqueza estava na falta de testes para os casos-base. Ao adicionar testes para `fatorial(0)` e `fatorial(1)`, forçamos a validação da lógica de parada, que os mutantes tentavam corromper. O mesmo princípio foi aplicado a funções de array, como `produtoArray` e `mediaArray`, onde testes com arrays vazios foram implementados para garantir que o valor inicial (`1` para produto, `0` para média) fosse retornado corretamente.

Um problema parecido foi encontrado em `maximoArray` e `minimoArray`. Os mutantes removiam a verificação de array vazio, o que causaria um erro. Adicionar testes que esperavam um `Error` para `maximoArray([])` e `minimoArray([])` validou o caminho de erro e matou esses mutantes.

Uma falha crítica foi observada nas funções booleanas (como `isPar`, `isMaiorQue`, `isDivisivel`, etc.). Os testes originais apenas validavam o resultado `true` (ex:

`isPar(100)`). Isso permitia que mutantes que simplesmente retornavam `true` sobrevivessem. A solução foi adicionar asserções para o resultado `false` em todos esses testes (ex: `expect(isPar(7)).toBe(false);`). Em funções de comparação como `isMaiorQue`, foi crucial testar também o caso de igualdade (`expect(isMaiorQue(5, 5)).toBe(false);`) para matar mutantes que trocavam `>` por `>=`.

Em funções de fórmula, como `celsiusParaFahrenheit` e `fahrenheitParaCelsius`, os testes originais usavam valores (`0` e `32`, respectivamente) que anulavam a parte principal do cálculo. Ao adicionar testes com valores não-zero (como `100` e `212`), garantimos que a lógica de multiplicação e divisão fosse de fato executada e validada, matando mutantes aritméticos.

Finalmente, a função `medianaArray` foi a que exigiu mais correções. O teste original usava um array ímpar e já ordenado. Para corrigir isso, foram adicionados três novos `expect`: um para testar o erro em array vazio, outro com um array ímpar e *desordenado* (para validar a lógica do `.sort()`), e um terceiro com um array de tamanho *par* e desordenado (para validar a lógica de média dos valores centrais). Essa abordagem garantiu que todos os caminhos lógicos da função fossem cobertos.

4 - Resultados Finais

Após as alterações, a taxa de mutantes mortos subiu para **96.71%,** um avanço consideravel em relação ao resultado inicial, mas ao realizar este trabalho, me deparei com um dilema, este sendo que, inicialmente a meta era atingir 98% de mutantes mortos, o que infelizmente não foi possível devido ao número de mutantes equivalentes, estes mutantes são alterações que são inseridas no código, mas que não alteram o comportamento original do código, esses mutantes **não** são mortos nestes testes, o que impede o alcance da meta.

No entanto, é crucial notar que esses mutantes sobreviventes não representam uma fraqueza na suíte de testes. Pelo contrário, a identificação de mutantes equivalentes é, em si, um resultado valioso da análise de mutação.

5 - Conclusão

O trabalho iniciou com uma suíte de testes que, apesar de atingir 100% de cobertura de código, apresentava uma pontuação de mutação de apenas 73.71%. Esta discrepância evidenciou a fraqueza dos testes originais: eles executavam todas as linhas, mas não validavam rigorosamente o comportamento. Prova disso foram os "mutantes críticos" que sobreviveram, como um na função `isPar` que passou a retornar `true` (pois o teste só verificava o "caminho feliz") e outro na `raizQuadrada` que removeu a verificação de erro (pois o teste nunca forneceu um número negativo).

Para corrigir essas falhas, a suíte de testes foi fortalecida com novas asserções focadas em casos de borda (como `fatorial(0)`), caminhos de erro (`expect().toThrow()`) e

resultados negativos (`expect(isPar(7)).toBe(false)`). Essas melhorias elevaram a pontuação de mutação para 96.71%. A meta de 98% não foi atingida devido à identificação de 7 mutantes equivalentes — alterações que não mudam o comportamento real do código, como em `fatorial` e `clamp`. A identificação desses mutantes foi um resultado valioso, comprovando que a suíte de testes final é, na prática, robusta e eficaz para detectar todas as falhas reais.