Pontifícia Universidade Católica de Minas Gerais

Engenharia de Software
Teste de Software

Análise de Eficácia de Testes com Teste de Mutação

Aluno: Ana Flávia de Carvalho Santos
Professor: Prof. Cleiton Tavares

Belo Horizonte
2 de novembro de 2025

1 Análise Inicial

Iniciei medindo a cobertura com a suíte de testes original. A execução apresentou 1 suíte e 50 testes, todos aprovados, com cobertura elevada: 98,64% de linhas, 85,41% de statements, 58,82% de ramos e 100% de funções, restando apenas a linha 112 não coberta em `operacoes.js`.

Em seguida, executei o StrykerJS para a análise de mutação com a mesma suíte fraca. O primeiro relatório indicou um *Mutation Score* de 73,71% considerando o total e de 78,11% considerando apenas o código coberto. Ao todo, 213 mutantes foram gerados: 154 foram mortos, 44 sobreviveram, 3 resultaram em *timeout* e 12 não foram cobertos. Isso implica 157 mutantes detectados e 56 não detectados. A Figura 1 mostra o resumo do relatório da primeira execução.

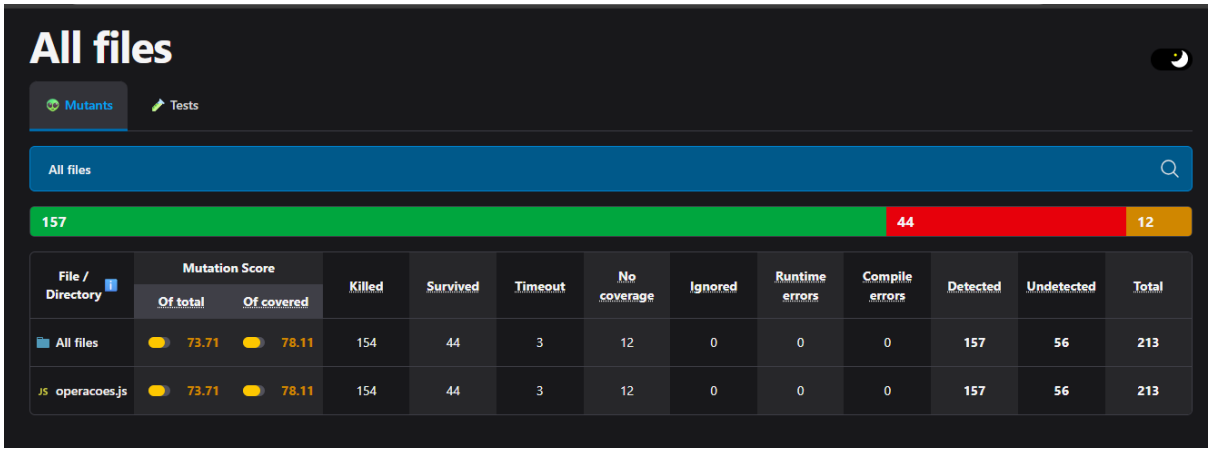

Figura 1: Relatório do StrykerJS na primeira execução com a suíte de testes inicial.

A discrepância entre a alta cobertura de linhas (98,64%) e o *Mutation Score* inicial (73,71%) evidencia que muitos trechos são exercitados pelos testes, mas sob óticas fracas ou insuficientes para detectar alterações sutis no código. Isso ocorre, por exemplo, quando apenas o "caminho feliz" é verificado sem cobrir limites, igualdade, valores nulos/zero, arrays vazios, mensagens de erro ou precisão numérica; nessas condições, mutações como trocas de operadores relacionais (`>` por `>=`), aritméticos (`+` por `-`) ou lógicos (`&&` por `||`) tendem a sobreviver. Assim, enquanto a cobertura de código indica amplitude de execução, a pontuação de mutação revela a qualidade das asserções; a diferença observada mostra que a suíte atual necessita de casos direcionados a bordas e invariantes para elevar a eficácia real dos testes.

2 Análise de Mutantes Críticos

2.1 Mutante 1 — fatorial

Esse mutante sobreviveu porque o algoritmo iterativo do fatorial inicializa o acumulador com 1 e itera de 2 até (n). Para (n=0) e (n=1), o laço não executa nenhuma iteração e o resultado permanece 1, ainda que o retorno antecipado tenha sido efetivamente desabilitado. Assim, o comportamento externo da função não se altera para os valores previstos pelo contrato. O teste original exercita apenas um caso "feliz" com (n=4) e valida o resultado 24, o que não é afetado por essa mutação; além disso, mesmo que existissem testes

```
17  function fatorial(n) {
18    if (n < 0) throw new Error('Fatorial não é definido para números negativos.');
19  -   if (n === 0 || n === 1) return 1;
    +   if (n === 0 && n === 1) return 1;
20    let resultado = 1;
21    for (let i = 2; i <= n; i++) { resultado *= i; }
22    return resultado;
```

Figura 2: Mutação no caso-base do fatorial.

para (n=0) e (n=1), o resultado continuaria sendo 1. Em termos práticos, trata-se de um mutante equivalente: a alteração sintática não produz diferença observável na saída da função, razão pela qual nenhum teste comportamental consegue "matá-lo".

2.2 Mutante 2 — medianaArray

```
function medianaArray(numeros) {
  if (numeros.length === 0) throw new Error('Array vazio не possui mediana.');
-   const sorted = [...numeros].sort((a, b) => a - b);
+   const sorted = [...numeros].sort((a, b) => a + b);
  const mid = Math.floor(sorted.length / 2);
  if (sorted.length % 2 === 0) {
    return (sorted[mid - 1] + sorted[mid]) / 2;
  }
  return sorted[mid];
}
```

Figura 3: Mutação em medianaArray.

Esse mutante não foi morto porque o teste original cobre apenas um cenário benigno: um array ímpar já ordenado e com valores positivos, que aciona somente a ramificação de tamanho ímpar e retorna diretamente o elemento do meio; mesmo com o comparador comprometido, o comportamento do sort em um input já ordenado e a seleção do índice central podem manter o valor 3 na posição mediana, mascarando o defeito. Como não há casos com entrada desordenada, números negativos ou repetidos, nem arrays de tamanho par, que exercitariam a média entre dois centrais, o teste não expõe a falha de ordenação e o mutante sobrevive.

2.3 Mutante 3 — isMaiorQue

```
104 - function isMaiorQue(a, b) { return a > b; }
    + function isMaiorQue(a, b) { return a >= b; }
```

Figura 4: Mutação em isMaiorQue na comparação relacional.

Esse mutante não foi morto porque o teste original exercita apenas um cenário trivial em que o primeiro argumento é estritamente maior que o segundo (isMaiorQue(10, 5)), para o qual tanto a implementação original quanto a mutada retornam true; não há verificação de casos negativos nem do limite de igualdade `(a === b)`, que diferenciaria `>` de `>=`. A ausência de um caso de borda como isMaiorQue(5, 5) esperando `false`, bem como de um caso com `a < b`, impede que o teste detecte a mudança de semântica e, por isso, o mutante sobrevive.

3 Solução Implementada

Para o mutante do fatorial, adicionei um conjunto de casos que exercitam os casos-base e a rejeição de entradas inválidas, verifiquei que (n=0) e (n=1) retornam 1, que números negativos disparam erro, que a progressão do valor cresce corretamente a partir de (n=2) e que relações internas como fatorial(3)/fatorial(2)=3 se mantêm. Esses testes aumentam a robustez e cobrem fronteiras relevantes, contudo, a mutação específica observada é comportamentalmente equivalente na implementação atual; o laço preserva o retorno 1 para n = 0 ou n = 1 mesmo sem o retorno antecipado, de modo que não há distinção observável pela caixa-preta. Assim, além dos novos testes, classifiquei esse mutante como equivalente no contexto do projeto, portanto não foi possível eliminá-lo.

Para o mutante da mediana, ampliei a suíte para cobrir cenários que forçam a ordenação real e os dois ramos da lógica testando array vazio com exceção, arrays de 1 e 2 elementos, entrada ímpar já ordenada e desordenada, caso par desordenado que exige média dos dois centrais, números negativos e garantia de imutabilidade do array original. Esses casos fazem com que qualquer erro no comparador de ordenação ou na regra de média para tamanhos pares se manifeste em saídas incorretas, especialmente nos testes com entrada desordenada e com negativos; desse modo, a alteração no comparador é detectada e o mutante é morto.

Para o mutante de comparação relacional em `isMaiorQue`, acrescentei testes que cobrem os três cenários essenciais: (a>b), (a<b) e a fronteira (a==b). O caso de igualdade exige retorno `false` e, portanto, diferencia `>` de `>=`; quando a mutação promove a igualdade, o teste 117 falha e elimina o mutante. Em complemento, os testes adicionados para `isMenorQue` e `isEqual` reforçam a consistência das relações de ordem e igualdade, reduzindo a probabilidade de sobrevivência de mutações semelhantes em funções semelhantes.

4 Resultados Finais

206 7

File / Directory	Mutation Score		Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
	Of total	Of covered										
All files	96.71	96.71	203	7	3	0	0	0	0	**206**	**7**	**213**
operacoes.js	96.71	96.71	203	7	3	0	0	0	0	**206**	**7**	**213**

Figura 5: Resultados finais do StrykerJS após os novos testes

Após a inclusão dos testes direcionados, o Mutation Score final atingiu 96,71%, com 203 mutantes mortos, 7 sobreviventes, 3 timeouts e 0 sem cobertura, como mostrado

na figura. Em relação à primeira execução, 73,71% com 154 mortos, 44 sobreviventes e 12 sem cobertura, houve um ganho de 23 pontos percentuais, 49 mutantes eliminados e a eliminação completa das lacunas de cobertura, evidenciando que os novos casos, em especial os que exercitam fronteiras de igualdade, entradas desordenadas e pares na mediana e caminhos excepcionais, fortaleceram os testes e detectaram alterações sutis no comportamento.Os 7 sobreviventes restantes incluem mutantes equivalentes, como o do caso-base do fatorial.

5 Conclusão

O teste de mutação foi essencial para avaliar a qualidade real da suíte de testes, indo além da métrica de cobertura de código. Enquanto a cobertura indica quais linhas e funções foram executadas, a mutação mede a eficácia dos testes, se alterações sutis no comportamento seriam detectadas. Na prática, ele força o exercício de fronteiras, caminhos excepcionais e invariantes que facilmente passam despercebidos em cenários "felizes". No projeto, essa abordagem revelou discrepâncias importantes entre cobertura alta e eficácia, guiando a criação de testes mais precisos para ordenar entradas, tratar arrays pares na mediana, validar mensagens de erro e, principalmente, diferenciar operadores em casos de igualdade.

Além de orientar a escrita de testes direcionados, a mutação também fomentou melhorias de design com a identificação de mutantes equivalentes (como o caso-base redundante do fatorial), oportunidades de refatoração e fortalecimento de contratos de função. O resultado final demonstra que combinar cobertura tradicional com teste de mutação produz suítes mais confiáveis, que detectam regressões reais e ajudam na evolução do código.