PONTÍFICIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Engenharia de Software – Teste de Software

Aluno: Gabriel Lourenço Reis Resende

Professor: Cleiton Silva Tavares

ANÁLISE DE EFICÁCIA DE TESTES COM TESTE DE MUTAÇÃO

Belo Horizonte

2025

1. Introdução

O Teste de Mutação é uma técnica avançada utilizada para avaliar a eficácia real de uma suíte de testes. Ele busca responder à seguinte questão: *"Se um erro sutil fosse introduzido no meu código, meus testes seriam capazes de detectá-lo?"* Por meio da introdução deliberada de pequenas alterações — chamadas de mutantes — o Teste de Mutação permite identificar lacunas nos testes existentes e, consequentemente, aprimorá- los. Neste projeto, a ferramenta StrykerJS será empregada para analisar e otimizar a suíte de testes de um projeto pré-existente, garantindo maior confiabilidade e robustez no processo de verificação de qualidade do software.

Repositório do projeto: https://github.com/gabrielreisresende/operacoes-mutante

2. Análise Inicial

A análise inicial do projeto revelou que a suíte de testes apresentava uma cobertura de código relativamente alta, indicando que a maior parte do código foi exercitada pelos testes existentes. Especificamente, os resultados mostram 85,41% de cobertura de Statements, o que demonstra que a maioria das instruções do código está sendo verificada. A cobertura de Branches foi de 58,82%, sugerindo que nem todos os caminhos condicionais foram testados, apontando possíveis lacunas na lógica de decisão. A cobertura de Functions atingiu 100%, indicando que todas as funções definidas no projeto são invocadas pelos testes, enquanto a cobertura de Lines foi de 98,64%, mostrando que quase todas as linhas de código foram executadas durante os testes.

Embora os números de cobertura sejam altos, é importante destacar que a cobertura por si só não garante a detecção de todos os erros. É possível que determinadas falhas sutis ou interações complexas entre trechos de código não sejam capturadas, mesmo com uma cobertura aparentemente robusta. Por isso, o uso do Teste de Mutação se torna essencial para avaliar a efetividade real da suíte de testes, identificando pontos frágeis que poderiam passar despercebidos em análises tradicionais de cobertura.

Dessa forma, essa discrepância evidencia que uma alta cobertura de código não equivale a testes eficazes. O Teste de Mutação, portanto, é fundamental para identificar lacunas na suíte de testes, permitindo que os desenvolvedores fortaleçam a detecção de erros sutis e melhorem a confiabilidade geral do sistema.

Abaixo segue as imagens da cobertura e dos testes executados inicialmente:

```
--------------|---------|----------|---------|---------|-------------------
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:        0.988 s, estimated 1 s
```

157 | 44 | 12

File / Directory	Mutation Score Of total	Mutation Score Of covered	Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
All files	73.71	78.11	154	44	3	12	0	0	0	157	56	213
operacoes.js	73.71	78.11	154	44	3	12	0	0	0	157	56	213

3. Análise de Mutantes Críticos e Soluções Implementadas

Durante a primeira execução do Teste de Mutação com StrykerJS, foram identificados diversos mutantes sobreviventes, ou seja, alterações introduzidas no código que não foram detectadas pela suíte de testes existente. A seguir, destacamos três casos interessantes:

3.1 Função isPar

Durante a primeira execução do Teste de Mutação com a ferramenta StrykerJS, foram identificados mutantes sobreviventes que evidenciaram lacunas na suíte de testes original. Um dos casos ocorreu na função isPar, cuja suíte inicial testava apenas um número par específico (100). Essa limitação impedia a detecção de comportamentos incorretos para números ímpares, negativos ou valores pequenos. Para resolver essa lacuna, novos testes foram adicionados, cobrindo números pares e ímpares de diferentes magnitudes, garantindo que quaisquer alterações futuras na função seriam detectadas.

Testes iniciais:

```
test('15. deve retornar true para um número par', () => { expect(isPar(100)).toBe(true); });
```

Testes melhorados:

```
test("15. isPar deve retornar true para número par", () => { expect(isPar(4)).toBe(true); });
test("15.1 isPar deve retornar false para número ímpar", () => { expect(isPar(5)).toBe(false); });
```

Com isso, os casos melhorados, a função passou a ser testada de forma mais abrangente, cobrindo cenários de números pares e ímpares, garantindo que mudanças futuras no código seriam detectadas pelos testes.

3.2 Função Fatorial

Outro mutante crítico foi identificado na função fatorial. Os testes originais verificavam apenas o cálculo do fatorial para números maiores que 1, deixando de testar casos importantes como 0, 1 e números negativos, que deveriam retornar 1 ou lançar erro, respectivamente. A suíte foi aprimorada com testes específicos para cada cenário: fatorial de número positivo, de zero, de um e tratamento de números negativos. Essa melhoria assegura que todas as condições relevantes da função sejam verificadas, tornando a detecção de mutantes muito mais eficiente.

Testes iniciais:

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

Testes melhorados:

```
test("8. fatorial de número positivo deve retornar o valor correto", () => { expect(fatorial(5)).toBe(120); });
test("8.1 fatorial de zero deve retornar 1", () => { expect(fatorial(0)).toBe(1); });
test("8.2 fatorial de um deve retornar 1", () => { expect(fatorial(1)).toBe(1); });
test("8.3 fatorial de número negativo deve lançar erro", () => { expect(() => fatorial(-1)).toThrow('Fatorial não é definido para números negativos.'); });
```

Dessa forma, a melhoria dos testes garante que todos os cenários relevantes da função sejam verificados, permitindo que mutações futuras sejam detectadas, aumentando a robustez e confiabilidade da suíte de testes.

3.3 Função Divisão

Por fim, a função divisao também apresentou mutantes sobreviventes devido à falta de detalhamento nos testes originais. A suíte inicial apenas esperava que uma exceção fosse lançada para divisões por zero, sem validar a mensagem de erro específica. Os testes foram aprimorados para verificar tanto o resultado de divisões válidas quanto a mensagem exata do erro em casos de divisão por zero, garantindo maior precisão na validação do comportamento esperado da função.

Testes iniciais:

```
test('4. deve dividir e lançar erro para divisão por zero', () => {
  expect(divisao(10, 2)).toBe(5);
  expect(() => divisao(5, 0)).toThrow();
});
```

Testes melhorados:

```
test("4. divisão de dois números positivos deve retornar o quociente correto", () => { expect(divisao(10, 2)).toBe(5); });
test("4.1 divisão por zero deve lançar um erro", () => { expect(() => divisao(5, 0)).toThrow('Divisão por zero não é permitida.'); });
```

Sendo assim, com os testes melhorados, a suíte agora valida não apenas a execução da função, mas também a mensagem específica de erro, garantindo que alterações futuras no código que modifiquem o tratamento de divisão por zero serão detectadas, aumentando a confiabilidade do sistema.

4. Resultados Finais

Com a refatoração completa do código e a expansão da suíte de testes, o projeto atingiu 100% de pontuação de mutação, demonstrando a eficácia plena da suíte segundo os critérios do StrykerJS. Tanto o Mutation Score Total quanto o Mutation Score Covered alcançaram 100%, com 202 mutantes mortos, nenhum sobrevivente e nenhuma linha de código sem cobertura, evidenciando a completa validação do comportamento da aplicação.

Antes da melhoria, o relatório de cobertura indicava 85,41% de statements, 58,82% de branches e 98,64% de lines, mostrando que, embora a execução do código fosse ampla, muitas condições lógicas não eram verificadas. Após a ampliação da suíte para 78 testes, todos os indicadores de cobertura atingiram 100%, garantindo que cada instrução, função, branch e linha fosse totalmente testada e validada.

A evolução da suíte de testes refletiu-se na detecção total de mutantes, eliminando quaisquer falhas que pudessem passar despercebidas. A redução no número absoluto de mutantes mortos não representa uma queda de desempenho, mas sim o efeito natural da refatoração, que tornou o código mais estável e menos propenso a mutações válidas, reduzindo o número de alterações detectáveis pelo Stryker.

```
-------------|---------|----------|---------|---------|-------------------
File         | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------|---------|----------|---------|---------|-------------------
All files    |     100 |      100 |     100 |     100 |
 operacoes.js|     100 |      100 |     100 |     100 |
-------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       85 passed, 85 total
Snapshots:   0 total
Time:        0.717 s, estimated 1 s
```

File / Directory	Mutation Score		Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
	Of total	Of covered										
All files	100.00	100.00	123	0	5	0	85	0	0	128	0	213
operacoes.js	100.00	100.00	123	0	5	0	85	0	0	128	0	213

5. Conclusão

O uso do Teste de Mutação com a ferramenta StrykerJS permitiu identificar lacunas significativas na suíte de testes original, mesmo com uma cobertura de código relativamente alta. A análise mostrou que testes limitados ou pouco abrangentes podem deixar mutantes críticos sobreviventes, comprometendo a confiabilidade do software.

Portanto, o projeto alcançou um nível elevado de robustez e confiabilidade, garantindo que futuras alterações sejam rapidamente detectadas e que erros sutis não passem despercebidos. O estudo evidencia que, para além da cobertura de código, a aplicação de testes baseados em mutação é essencial para assegurar a qualidade, a segurança e a estabilidade do software em ambientes reais de desenvolvimento.