PONTÍFICIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Engenharia de Software – Teste de Software

Aluno: Gabriel Lourenço Reis Resende

Professor: Cleiton Silva Tavares

Refatoração de Testes e Detecção de Test Smells

Belo Horizonte

2025

1. Introdução

A confiabilidade de um sistema de software é essencialmente ligada à qualidade de sua suíte de testes. No entanto, o surgimento de "Test Smells" (maus cheiros de código de teste) – como Lógica Condicional, Testes Frágeis e o Teste Eager – compromete seriamente a clareza, a manutenibilidade e a confiança nos resultados da verificação. Este projeto concentrou-se na análise detalhada e na refatoração de uma suíte de testes para a classe *UserService*. Dessa forma, a metodologia empregada baseou-se na aplicação rigorosa do padrão Arrange, Act, Assert (AAA) para estruturar testes de forma lógica e isolada, complementada pelo uso de análise estática via ESLint. O objetivo final foi transformar um conjunto de testes confuso em um ativo de verificação robusto e legível, eliminando os anti-padrões e garantindo maior sustentabilidade no processo de desenvolvimento de software.

Repositório do projeto: https://github.com/gabrielreisresende/test-smelly

2. Análise de Smells

2.1 Lógica Condicional e Eager Test

Teste: deve desativar usuários se eles não forem administradores.

Explicação do Smell: Um teste limpo deve focar em um único conceito e comportamento. Sendo assim, este teste violava essa regra ao utilizar um for loop e uma estrutura if/else para testar dois cenários distintos (desativação de usuário comum e falha na desativação de administrador) em uma única função. A Lógica Condicional (Conditional Logic in Test) é um *smell* direto, pois obscurece o caminho de execução e dificulta a compreensão do que está sendo testado.]

Risco:

- Baixa Clareza: É difícil saber rapidamente qual falha corresponde a qual cenário.
- Quebra Falsa: Se a lógica de desativação do usuário comum quebrar, o teste falha, mas a falha do administrador (que deveria retornar false) pode estar funcionando corretamente, mas o relatório de teste será confuso.

2.2 Test Passes Silently

Teste: deve falhar ao criar usuário menor de idade.

Explicação do Smell: O teste utilizava um bloco try/catch para validar a exceção de menoridade. Se a validação de idade fosse removida do código de produção (impedindo o lançamento da exceção), o bloco try seria executado, o bloco catch seria ignorado e o teste passaria sem executar nenhuma assertova. Dessa forma, o teste executada da maneira errada, sem detectar a execução esperada.

Risco:

- Falsa Confiança: O teste reporta sucesso mesmo quando a regra de negócio (maioridade) não está mais sendo aplicada, escondendo um bug de validação crucial.
- Anti-Padrão Jest: Ignora o método nativo toThrow() do Jest, que é projetado para lidar com exceções de forma segura e explícita.

2.3 Fragile Test

Teste: deve gerar um relatório de usuários formatado.

Explicação do Smell: O teste verificava a saída do relatório utilizando expect(relatorio).toContain(onde a linha esperada era uma string formatada com espaços, vírgulas e \n exatos. Isso cria

um teste frágil (Fragile Test), onde uma pequena mudança na implementação (como adicionar um espaço extra ou mudar a ordem das colunas no relatório) faria o teste falhar, mesmo que o comportamento principal de gerar o relatório com os dados corretos estivesse preservado.

Risco:

- Manutenção Constante: O teste precisa ser atualizado a cada mudança de estilo/-formatação.
- Foco na Implementação: O teste verifica como o relatório é formatado, e não se os dados corretos foram incluídos.

3. Processo de Refatoração

Foi escolhido o teste de desativação de usuário (Smell 1) para demonstrar a aplicação do padrão Arrange, Act, Assert (AAA) e a eliminação de lógica condicional.

3.1 Cenário Inicial (Smell)

O código abaixo, além de conter a Lógica Condicional, violava o princípio do Teste Eager (fazer muitas coisas em um só teste).

```
test('deve desativar usuários se eles não forem administradores', () => {
  const usuarioComum = userService.createUser('Comum', 'comum@teste.com', 30);
  const usuarioAdmin = userService.createUser('Admin', 'admin@teste.com', 40, true);

  const todosOsUsuarios = [usuarioComum, usuarioAdmin];

  // O teste tem um loop e um if, tornando-o complexo e menos claro.
  for (const user of todosOsUsuarios) {
    const resultado = userService.deactivateUser(user.id);
    if (!user.isAdmin) {
      // Este expect só roda para o usuário comum.
      expect(resultado).toBe(true);
      const usuarioAtualizado = userService.getUserById(user.id);
      expect(usuarioAtualizado.status).toBe('inativo');
    } else {
      // E este só roda para o admin.
      expect(resultado).toBe(false);
    }
  }
});
```

3.2 Cenário Refatorado (Teste limpo)

O teste foi dividido em dois testes separados, focados e sem nenhuma lógica condicional, seguindo o padrão AAA:

```
test("GIVEN a normal user WHEN deactivating THEN returns true and status 'inactive'", () => {
  const defaultUser = userService.createUser("Pedro", "pedro@teste.com", 31);

  const desativation = userService.deactivateUser(defaultUser.id);
  const resultedUser = userService.getUserById(defaultUser.id);

  expect(desativation).toBe(true);
  expect(resultedUser.status).toBe("inativo");
});
```

```
test("GIVEN an admin user WHEN trying to deactivate THEN returns false", () => {
  const admin = userService.createUser("Helena", "helena@teste.com", 45, true);

  const result = userService.deactivateUser(admin.id);

  expect(result).toBe(false);
});
```

3.3 Decisões de Refatoração

Separação: O *Eager Test* foi dividido em dois testes distintos, garantindo que cada teste tenha um único motivo para falhar, tornando a depuração imediata.

Eliminação de Condicionais: O for *loop* e o if/else foram completamente removi- dos, seguindo a regra de que testes devem ser declarações estáticas de comporta- mento.

Padrão AAA: A estrutura de cada novo teste é clara:

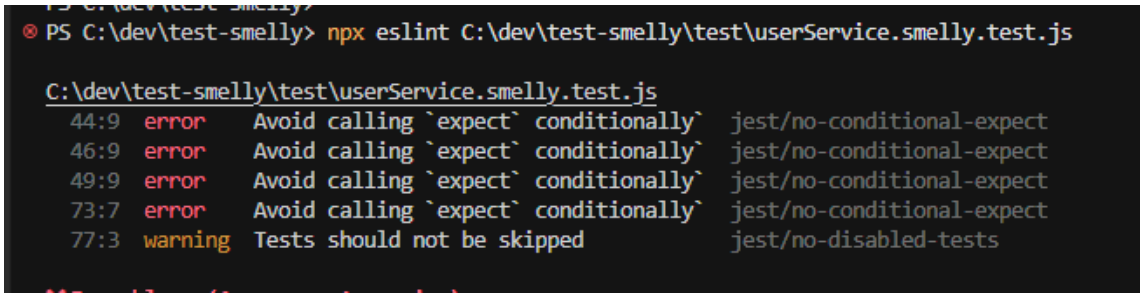
- Arrange: Inicialização da userService e criação do usuário
- Act: Execução do método sob teste (deactivateUser).
- Assert: Verificação dos resultados

4. Relatório da Ferramenta

A execução inicial do ESLint foi crucial para automatizar a detecção de test smells no código e para validar a estrutura e elaboração dos testes do projeto.

Erros detectados:

- **Avoid calling `expect` conditionally**:
    - Detectou e apontou diretamente o Smell 1 (Lógica Condicional), forçando a refatoração dos testes para eliminar os if/else aninhados em blocos de expect.
- **Test should not be skipped**:
    - Detectou a presença de test.skip e alertou sobre testes desabilitados.



5. Conclusão

A refatoração da suíte de testes e a utilização do ESLint demonstram a interligação vital entre testes de alta qualidade e ferramentas de análise estática. A eliminação de *test smells* como a Lógica Condicional e o Teste *Eager* resultou em testes que são mais legíveis e mais resilientes a mudanças no código de produção, reduzindo a fragilidade da implementação.

Além disso, a adoção do padrão Arrange-Act-Assert(AAA) e a verificação de comportamento aumentam a sustentabilidade do projeto. Com isso, ferramentas como o ESLint agem automatizando a aplicação de padrões de limpeza (como jest/no- conditional-expect) e impedindo que *smells* críticos, como Lógica Condicional, sejam introduzidos no código, garantindo a qualidade a longo prazo.