Nome da Disciplina

Refatoração de Testes e Detecção de Test Smells

Relatório de Atividade Prática

Aluno: **Seu Nome Completo**

Matrícula: **Sua Matrícula**

3 de novembro de 2025

1 Análise de Smells

A suíte de testes original (‘userService.smelly.test.js‘) apresenta vários ”Test Smells”, que são sintomas de problemas mais profundos no código de teste. A seguir, são descritos três dos smells mais críticos encontrados.

1.1 Lógica Condicional no Teste

Descrição: O teste ‘deve desativar usuários se eles não forem administradores‘ utiliza uma estrutura de repetição (‘for‘) e uma condicional (‘if‘) para verificar dois cenários distintos em um único bloco de teste.

Por que é um ”mau cheiro”? Testes devem ser simples e ter um fluxo de execução direto. A presença de lógica condicional dentro de um teste o torna complexo, difícil de ler e de entender qual é a sua real intenção. Idealmente, cada teste deve verificar um único comportamento.

Risco: O principal risco é a criação de testes que podem passar sem verificar todos os cenários. Se a lógica dentro do ‘if‘ for alterada ou se um novo tipo de usuário for adicionado, o teste pode se tornar ineficaz sem que isso seja percebido, gerando falsos positivos.

1.2 Teste Frágil (Brittle Test)

Descrição: No teste ‘deve gerar um relatório de usuários formatado‘, a asserção verifica uma string exata (‘const linhaEsperada = ...‘) para validar o conteúdo do relatório.

Por que é um ”mau cheiro”? Um teste frágil quebra por motivos irrelevantes à lógica de negócio. Qualquer pequena alteração na formatação do relatório — como a adição de um espaço, uma quebra de linha ou a mudança na ordem das informações — fará com que o teste falhe, mesmo que a funcionalidade principal (gerar o relatório com os dados corretos) esteja intacta.

Risco: Manutenção custosa. Cada pequeno ajuste na saída de dados exigirá uma atualização manual no teste, o que consome tempo e desvia o foco da validação da lógica. Isso torna a suíte de testes um obstáculo em vez de uma ferramenta de segurança.

1.3 Expect Condicional (Teste de Exceção Ruim)

Descrição: O teste ‘deve falhar ao criar usuário menor de idade‘ utiliza um bloco ‘try/-catch‘ para verificar se uma exceção é lançada. A asserção (‘expect‘) está dentro do bloco ‘catch‘.

Por que é um ”mau cheiro”? Esta é uma prática perigosa. Se a função ‘createUser‘ for alterada e deixar de lançar a exceção, o fluxo do teste nunca entrará no ‘catch‘, e o teste passará silenciosamente, sem nenhuma verificação. Isso cria um falso positivo, dando a impressão de que tudo está certo quando, na verdade, um bug foi introduzido.

Risco: O maior risco é a perda de confiança na suíte de testes. Um bug crítico (como a perda de uma validação de regra de negócio) pode passar despercebido, chegando ao ambiente de produção.

2 Processo de Refatoração

O teste mais problemático foi, sem dúvida, ‘deve desativar usuários se eles não forem administradores‘, pois continha múltiplos smells. Abaixo, apresentamos o ”Antes”e o ”Depois”da refatoração.

2.1 Antes

```
test('deve desativar usu rios se eles n  o 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 s   roda para o usu rio comum.
      expect(resultado).toBe(true);
      const usuarioAtualizado = userService.getUserById(user.id);
      expect(usuarioAtualizado.status).toBe('inativo');
    } else {
      // E este s   roda para o admin.
      expect(resultado).toBe(false);
    }
  }
});
```

Listing 1: Teste original com Lógica Condicional e múltiplas responsabilidades.

2.2 Depois

```
test('deve desativar um usu rio comum com sucesso', () => {
  // Arrange: Cria um usu rio que n o    administrador
  const usuarioComum = userService.createUser('Comum', 'comum@teste.
     com', 30);

  // Act: Tenta desativar o usu rio
  const resultado = userService.deactivateUser(usuarioComum.id);

  // Assert: Verifica se a desativa  o funcionou
  expect(resultado).toBe(true);
  const usuarioAtualizado = userService.getUserById(usuarioComum.id)
     ;
  expect(usuarioAtualizado.status).toBe('inativo');
});

test('n  o deve desativar um usu rio administrador', () => {
  // Arrange: Cria um usu rio administrador
  const usuarioAdmin = userService.createUser('Admin', 'admin@teste.
     com', 40, true);
```

```
17
18    // Act: Tenta desativar o administrador
19    const resultado = userService.deactivateUser(usuarioAdmin.id);
20
21    // Assert: Verifica que a desativa  o  n  o  funcionou
22    expect(resultado).toBe(false);
23    const usuarioAtualizado = userService.getUserById(usuarioAdmin.id)
          ;
24    expect(usuarioAtualizado.status).toBe('ativo');
25 });
```

Listing 2: Testes refatorados, cada um com uma única responsabilidade.

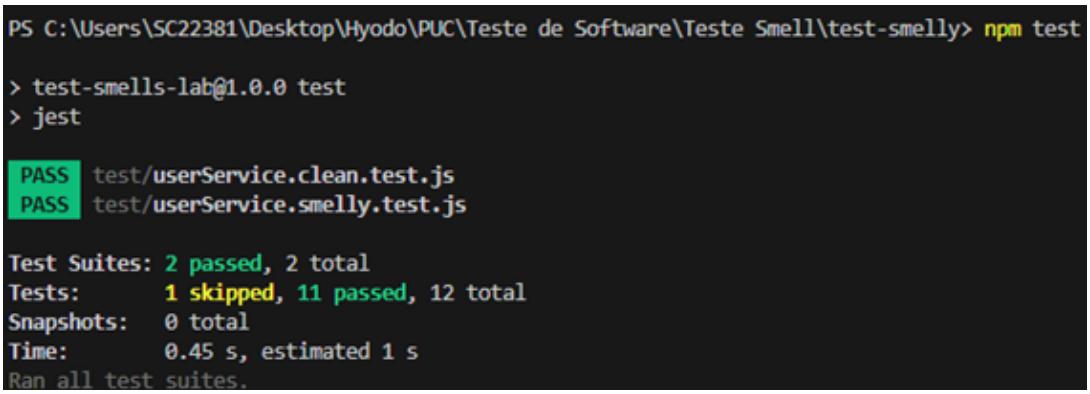

Figura 1: Resultado final da execução dos testes (‘npm test‘), mostrando que a refatoração foi bem-sucedida e não quebrou a funcionalidade existente.

2.3 Decisões e Correções

A refatoração seguiu os seguintes princípios:

- **Divisão de Responsabilidades:** O teste original foi dividido em dois, cada um verificando um cenário específico e independente (desativar usuário comum e tentar desativar administrador). Isso elimina o smell de ”Lógica Condicional”.
- **Padrão AAA (Arrange, Act, Assert):** Ambos os novos testes seguem claramente o padrão AAA, com seções bem definidas para preparação do cenário, execução da ação e verificação do resultado. Isso torna o código mais legível e fácil de entender.
- **Nomes Descritivos:** Os nomes dos novos testes (‘deve desativar um usuário comum com sucesso‘ e ‘não deve desativar um usuário administrador‘) são explícitos e descrevem exatamente o comportamento que estão validando.

Como resultado, os testes se tornaram mais robustos, fáceis de manter e mais claros em sua intenção, corrigindo todos os smells identificados no código original.

3 Relatório da Ferramenta

A ferramenta de análise estática ESLint, configurada com o plugin para Jest, foi fundamental para automatizar a detecção dos Test Smells. A primeira execução do comando ‘npx eslint .‘ já revelou problemas críticos na suíte de testes original.

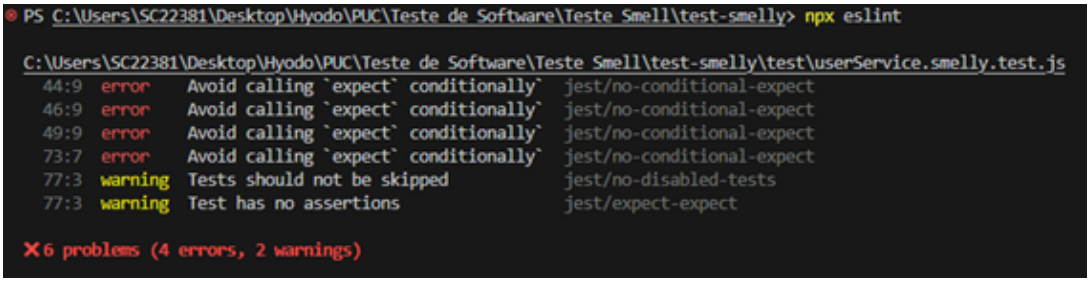

Figura 2: Resultado da primeira execução do ESLint, mostrando 4 erros e 2 avisos.

A Figura 2 mostra que o ESLint identificou automaticamente:

- **4 erros de 'jest/no-conditional-expect':** Apontou exatamente as linhas onde o 'expect' estava dentro de uma estrutura condicional, validando nossa análise manual sobre o smell de "Expect Condicional".
- **1 aviso de 'jest/no-disabled-tests':** Identificou o teste que foi pulado com 'test.skip', chamando a atenção para uma funcionalidade não testada.
- **1 aviso de 'jest/expect-expect':** Alertou que o teste pulado não possuía asserções.

A ferramenta automatizou a detecção de padrões de código ruins, economizando tempo e garantindo que nenhum smell do tipo que ela detecta passe despercebido. Ela complementa a análise manual, que é necessária para identificar problemas mais sutis, como o "Teste Frágil"ou a falta de clareza no nome de um teste.

4 Conclusão

Este trabalho prático demonstrou de forma clara a importância de se escrever testes limpos e de utilizar ferramentas de análise estática no ciclo de desenvolvimento de software. A refatoração da suíte de testes, guiada pela identificação de "Test Smells", transformou um código frágil e obscuro em um conjunto de testes claros, focados e robustos.

A aplicação do padrão AAA e a eliminação de lógica condicional nos testes não apenas melhoraram a legibilidade, mas também aumentaram a eficácia da suíte em detectar bugs. A ferramenta ESLint provou ser uma aliada poderosa, automatizando a detecção de anti-padrões e servindo como uma guarda de qualidade constante.

Em suma, a combinação de boas práticas de engenharia de software com ferramentas automatizadas é essencial para construir projetos que são não apenas funcionais, mas também sustentáveis e seguros a longo prazo. Testes limpos são um investimento que se paga em confiança e agilidade durante toda a vida útil do software.