Refatoração de Testes e Detecção de Test Smells

Pontifícia Universidade Católica de Minas Gerais Campus Coração Eucarístico

Disciplina: Teste de Software

Trabalho: Análise de Eficácia de Testes com Teste de Mutação

Aluno: Henrique Lobo (808840)

Orientador: Prof. Dr. Cleiton Tavares

Contexto Inicial e Introdução

Este trabalho consiste em uma análise detalhada da eficácia de uma suíte de testes automatizados por meio da identificação e correção de **Test Smells** (maus cheiros de teste). O foco é demonstrar como a má qualidade na escrita dos testes compromete a sua robustez e manutenibilidade.

A análise inicia com a identificação de testes problemáticos — como aqueles com **Lógica Condicional**, testes **Frágeis** ou com **Tratamento Inadequado de Exceções** — que mascaram vulnerabilidades no código e geram custos de manutenção elevados.

O objetivo principal é a refatoração desses testes, aplicando boas práticas como a **Separação de Cenários** e o **Padrão AAA (Arrange-Act-Assert)**, para transformá-los em ativos de qualidade que oferecem documentação clara e validação confiável do sistema. A metodologia demonstra que o investimento na qualidade dos testes é fundamental para a sustentabilidade do *software* a longo prazo.

---

1. Análise de Test Smells

1.1 Lógica Condicional em Testes (*Conditional Test Logic*)

Característica	**Detalhamento**
Localização	Teste "deve desativar usuários se eles não forem administradores"

Código Exemplo	javascript\nfor (const user of todosOsUsuarios) { ... if (!user.isAdmin) { ... } else { ... } }\n
Problema (*Smell*)	A presença de estruturas condicionais (if/else) e *loops* em testes é um mau cheiro porque torna o teste mais complexo, pode ocultar falhas e viola o princípio de que cada teste deve verificar um único cenário.
Riscos	Testes que passam parcialmente podem mascarar problemas, dificuldade de manutenção e menor clareza sobre qual cenário específico falhou.

1.2 Teste Frágil (*Fragile Test*)

Característica	**Detalhamento**
Localização	Teste "deve gerar um relatório de usuários formatado"
Código Exemplo	javascript\nconst linhaEsperada = `ID: ${usuario1.id}, Nome: Alice, Status: ativo\\n`;\\nexpect(relatorio).toContain(linhaEsperada);\n
Problema (*Smell*)	O teste é considerado frágil porque depende da **formatação exata** do texto, incluindo espaços e quebras de linha. Qualquer mudança cosmética no formato do relatório quebrará o teste.
Riscos	Falsos negativos (testes falham mesmo quando a funcionalidade está correta), alto custo de manutenção e resistência a refatorações legítimas.

1.3 Tratamento Inadequado de Exceções (*Improper Exception Handling*)

Característica	**Detalhamento**

Localização	Teste "deve falhar ao criar usuário menor de idade"
Código Exemplo	javascript\ntry {\\n userService.createUser('Menor', 'menor@email.com', 17);\\n} catch (e) {\\n expect(e.message).toBe('O usuário deve ser maior de idade.');\\n}\n
Problema (*Smell*)	O padrão try/catch usado desta forma **não garante** que a exceção realmente foi lançada. O teste passa silenciosamente se nenhuma exceção for lançada.
Riscos	Falsos positivos (teste passa quando deveria falhar), comportamentos inesperados podem passar despercebidos e viola as boas práticas do Jest para teste de exceções.

---

2. Processo de Refatoração

2.1 Antes (Código Original)

JavaScript
```
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];

  for (const user of todosOsUsuarios) {
    const resultado = userService.deactivateUser(user.id);
    if (!user.isAdmin) {
      expect(resultado).toBe(true);
      const usuarioAtualizado = userService.getUserById(user.id);
      expect(usuarioAtualizado.status).toBe('inativo');
    } else {
      expect(resultado).toBe(false);
    }
  }
});
```

2.2 Depois (Código Refatorado)

JavaScript
```
describe('deactivateUser', () => {
  test('should deactivate non-admin user', () => {
```

```
    // Arrange
    const regularUser = userService.createUser('Regular', 'regular@example.com', 30);

    // Act
    const result = userService.deactivateUser(regularUser.id);

    // Assert
    expect(result).toBe(true);
    expect(userService.getUserById(regularUser.id).status).toBe('inativo');
  });

  test('should not deactivate admin user', () => {
    // Arrange
    const adminUser = userService.createUser('Admin', 'admin@example.com', 40, true);

    // Act
    const result = userService.deactivateUser(adminUser.id);

    // Assert
    expect(result).toBe(false);
    expect(userService.getUserById(adminUser.id).status).toBe('ativo');
  });
});
```

2.3 Decisões de Refatoração

Decisão	Ação Implementada	Justificativa
Separação de Cenários	O teste original foi dividido em dois testes distintos.	Eliminação da lógica condicional e foco em um único cenário por teste.
Padrão AAA	Estruturação das seções Arrange, Act e Assert.	Melhor legibilidade, manutenibilidade e clareza do propósito do teste.
Nomes Descritivos	Nomes dos testes mais claros e específicos (should deactivate non-admin user).	Documentação mais efetiva do comportamento esperado do código.

---

3. Relatório da Ferramenta de Análise (ESLint)

A primeira execução do ESLint, configurado com plugins para testes, revelou diversos problemas (*Test Smells*) na suíte de testes original:

Bash
ESLint: 9.39.0

Warning: Test has too many assertions (jest/max-expects)
Warning: Conditional test logic detected (jest/no-conditional-test-expect)
Warning: Test title is too vague (jest/valid-title)
Error: Skipped test detected (jest/no-disabled-tests)

A ferramenta automatizou a detecção dos *test smells* de várias formas:

- Identificou o uso de lógica condicional em testes.
- Alertou sobre testes muito complexos (muitas *assertions*).
- Destacou testes desabilitados.
- Sugere melhorias nos nomes dos testes.

---

4. Conclusão

Este trabalho demonstrou a importância crítica da **qualidade na escrita de testes automatizados**. Através da identificação e correção de *test smells*, foi possível transformar uma suíte de testes problemática em um conjunto de testes mais **robusto, legível e manutenível**.

A utilização de ferramentas de **análise estática**, como o ESLint com *plugins* específicos para testes, mostrou-se fundamental no processo de melhoria contínua do código. Estas ferramentas não apenas identificam problemas, mas também educam os desenvolvedores sobre boas práticas de teste.

Principais Aprendizados:

1. **Testes limpos** são tão importantes quanto código limpo.
2. Ferramentas de **análise estática** são essenciais para manter a qualidade.
3. **Refatoração constante** é necessária para evitar a degradação dos testes.
4. Testes bem escritos servem como **documentação viva** do código.

A experiência reforçou que investir tempo na qualidade dos testes não é apenas uma questão de boas práticas, mas um **investimento fundamental na sustentabilidade e manutenibilidade do software** a longo prazo.