Pontifícia Universidade Católica de Minas Gerais

PUC Minas

Curso: Engenharia de Software

Disciplina: Teste de Software

Professor: Cleiton Tavares

Refatoração de Testes e Detecção de Test Smells

Aluno: Luiz Nery

2 de novembro de 2025

1 Análise de Test Smells

Durante a análise da suíte de testes em `userService.smelly.test.js`, foram identificados três principais *Test Smells* que comprometem a qualidade dos testes:

1.1 Smell #1: Conditional Test Logic

Localização: Teste "deve desativar usuários se eles não forem administradores"

Problema: O teste contém um loop `for` e estruturas condicionais `if/else`, criando múltiplos caminhos de execução dentro de um único teste.

Por que é um mau cheiro: Testes devem ter um fluxo de execução linear e previsível. A presença de lógica condicional torna o teste complexo, dificulta o entendimento e viola o padrão Arrange-Act-Assert. Diferentes caminhos podem ser executados dependendo dos dados, tornando o teste menos determinístico.

Riscos:

- Dificulta identificar qual cenário falhou
- Alguns caminhos podem nunca ser testados
- Viola princípio de responsabilidade única
- Torna debugging mais complexo

1.2 Smell #2: Eager Test

Localização: Teste "deve criar e buscar um usuário corretamente"

Problema: Um único teste verifica duas funcionalidades distintas: criação de usuário E busca de usuário.

Por que é um mau cheiro: Cada teste deve focar em uma única responsabilidade. Quando múltiplas funcionalidades são testadas juntas, perde-se granularidade e clareza. Se o teste falhar, não fica claro qual das funcionalidades causou o problema.

Riscos:

- Baixa granularidade na identificação de falhas
- Mistura conceitos diferentes
- Dificulta manutenção e evolução dos testes
- Reduz a clareza sobre o que está sendo testado

1.3 Smell #3: Exception Handling

Localização: Teste "deve falhar ao criar usuário menor de idade"

Problema: Uso de bloco `try-catch` manual para verificar se uma exceção é lançada, ao invés de utilizar os mecanismos apropriados do framework de teste.

Por que é um mau cheiro: Se a exceção não for lançada, o código dentro do `try` executa normalmente e o bloco `catch` é ignorado, fazendo o teste passar silenciosamente mesmo quando deveria falhar. Isso cria uma falsa sensação de segurança.

Riscos:

- Teste passa mesmo quando exceção não é lançada
- Não garante que o comportamento esperado ocorra
- Código verboso e menos idiomático
- Dificulta compreensão da intenção do teste

2 Processo de Refatoração

2.1 Teste Escolhido: Conditional Test Logic

Este foi o teste mais problemático, apresentando lógica condicional complexa.

Código ANTES (userService.smelly.test.js):

Listing 1: Teste com smells - linhas 34-52

```
1  test('deve desativar usuarios se eles nao forem administradores', () =>
       {
2    const usuarioComum = userService.createUser('Comum', 'comum@teste.com
       ', 30);
3    const usuarioAdmin = userService.createUser('Admin', 'admin@teste.com
       ', 40, true);
4    const todosOsUsuarios = [usuarioComum, usuarioAdmin];
5
6    for (const user of todosOsUsuarios) {
7      const resultado = userService.deactivateUser(user.id);
8      if (!user.isAdmin) {
9        expect(resultado).toBe(true);
10       const usuarioAtualizado = userService.getUserById(user.id);
11       expect(usuarioAtualizado.status).toBe('inativo');
12     } else {
13       expect(resultado).toBe(false);
14     }
15   }
16 });
```

Código DEPOIS (userService.clean.test.js):

Listing 2: Testes refatorados seguindo AAA

```
1  test('deve desativar usuario comum', () => {
2    // Arrange
3    const usuario = userService.createUser('Comum', 'comum@teste.com', 30)
       ;
4
5    // Act
6    const resultado = userService.deactivateUser(usuario.id);
7
8    // Assert
9    expect(resultado).toBe(true);
10   const usuarioAtualizado = userService.getUserById(usuario.id);
11   expect(usuarioAtualizado.status).toBe('inativo');
12 });
13
14 test('nao deve desativar usuario admin', () => {
15   // Arrange
16   const admin = userService.createUser('Admin', 'admin@teste.com', 40,
       true);
17
18   // Act
19   const resultado = userService.deactivateUser(admin.id);
20
21   // Assert
```

```
22   expect(resultado).toBe(false);
23   const adminAtualizado = userService.getUserById(admin.id);
24   expect(adminAtualizado.status).toBe('ativo');
25 });
```

2.2 Decisões de Refatoração

1. **Separação em dois testes independentes:** Cada cenário (usuário comum e admin) agora possui seu próprio teste focado, eliminando a necessidade de loops e condicionais.
2. **Aplicação rigorosa do padrão AAA:** Cada teste segue claramente as três fases - Arrange (preparação), Act (execução), Assert (verificação).
3. **Nomes descritivos:** Os títulos "deve desativar usuário comum" e "não deve desativar usuário admin" deixam claro o comportamento esperado.
4. **Fluxo linear:** Eliminação completa de lógica condicional garante que cada teste tenha apenas um caminho de execução.
5. **Isolamento:** Os testes são completamente independentes e podem ser executados em qualquer ordem.

Estas mudanças transformaram um teste complexo e obscuro em dois testes simples, claros e autoexplicativos, melhorando significativamente a manutenibilidade e eficácia.

3 Relatório da Ferramenta ESLint

3.1 Configuração

O ESLint foi configurado com o plugin `eslint-plugin-jest` para detectar automaticamente *Test Smells*:

```
npm install --save-dev eslint eslint-plugin-jest
```

Principais regras configuradas:

- `jest/no-conditional-expect`: Detecta expects dentro de condicionais
- `jest/no-disabled-tests`: Alerta sobre testes desabilitados
- `jest/no-identical-title`: Previne títulos duplicados

3.2 Primeira Execução - Detecção de Problemas

Comando executado:

```
npx eslint test/userService.smelly.test.js
```

Resultado:

```
C:\...\test\userService.smelly.test.js
  44:9  error    Avoid calling ‘expect‘ conditionally‘
                 jest/no-conditional-expect
  46:9  error    Avoid calling ‘expect‘ conditionally‘
                 jest/no-conditional-expect
  49:9  error    Avoid calling ‘expect‘ conditionally‘
                 jest/no-conditional-expect
  73:7  error    Avoid calling ‘expect‘ conditionally‘
                 jest/no-conditional-expect
  77:3  warning  Tests should not be skipped
                 jest/no-disabled-tests
  77:3  warning  Test has no assertions
                 jest/expect-expect

 6 problems (4 errors, 2 warnings)
```

3.3 Como a Ferramenta Automatizou a Detecção

O ESLint identificou automaticamente:

- **4 erros críticos** de lógica condicional (linhas onde `expect` estava dentro de `if`)
- **2 avisos** sobre teste desabilitado com `test.skip`
- Confirmou todos os smells identificados na análise manual

A ferramenta economizou tempo ao automatizar a detecção, garantiu consistência na identificação de problemas e forneceu feedback imediato sobre a localização exata dos smells.

3.4 Segunda Execução - Validação

Após refatoração:

```
1 npx eslint test/userService.clean.test.js
```

Resultado: Nenhum problema encontrado (0 errors, 0 warnings).
Todos os testes passaram com sucesso:

```
Test Suites: 2 passed, 2 total
Tests:       1 skipped, 18 passed, 19 total
```

4 Conclusão

A refatoração de testes demonstrou que alta cobertura de código não garante eficácia. Os *Test Smells* identificados tornavam a suíte frágil, difícil de manter e incapaz de detectar bugs com confiança.

4.1 Principais Conquistas

- Eliminação completa de lógica condicional nos testes
- Aplicação 100% do padrão Arrange-Act-Assert
- Redução de 6 problemas do ESLint para zero
- Aumento de 5 para 14 testes, melhorando granularidade
- Testes mais claros, concisos e autodocumentados

4.2 Impacto na Qualidade do Software

A escrita de testes limpos e a utilização de ferramentas de análise estática como o ESLint contribuem significativamente para a qualidade e sustentabilidade do projeto:

Testes Limpos:

- Servem como documentação viva do comportamento esperado
- Facilitam refatorações com segurança
- Reduzem custo de manutenção a longo prazo
- Aceleram onboarding de novos desenvolvedores
- Aumentam confiança na suíte de testes

Análise Estática Automatizada:

- Detecta problemas automaticamente durante desenvolvimento
- Garante consistência na aplicação de boas práticas
- Economiza tempo em code reviews
- Previne introdução de novos smells
- Fornece feedback imediato aos desenvolvedores

4.3 Reflexão Final

Este trabalho reforça que testes devem receber o mesmo nível de cuidado e atenção que o código de produção. A combinação de análise manual criteriosa com ferramentas automatizadas mostrou-se extremamente eficaz na identificação e correção de problemas.

O padrão AAA, quando aplicado rigorosamente, transforma testes confusos em código limpo e autoexplicativo. A eliminação de lógica condicional garante previsibilidade e confiabilidade. E ferramentas como o ESLint garantem que essas boas práticas sejam mantidas ao longo do tempo.

Investir em qualidade de testes não é apenas uma boa prática técnica, é essencial para a sustentabilidade e evolução de qualquer projeto de software moderno.

Repositório GitHub: `https://github.com/LuizNeryy/test-smelly`