

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

Trabalho Prático

Aluno: Luiz Nery

2 de novembro de 2025

Conteúdo

1	Introdução	2
1.1	Contexto do Projeto	2
1.2	Objetivos	2
2	Análise de Test Smells Identificados	3
2.1	Conditional Test Logic (Lógica Condisional em Testes)	3
2.2	Eager Test (Teste Ansioso)	3
2.3	Sensitive Equality (Igualdade Sensível)	4
2.4	Exception Handling (Tratamento Manual de Exceções)	4
2.5	Disabled Test (Teste Desabilitado)	4
3	Configuração da Ferramenta de Detecção Automatizada	6
3.1	Instalação das Dependências	6
3.2	Configuração do ESLint	6
3.3	Regras Configuradas	6
4	Relatório da Ferramenta de Análise Estática	7
4.1	Primeira Execução - Arquivo Original	7
4.2	Análise dos Resultados	7
4.3	Segunda Execução - Arquivo Refatorado	7
5	Processo de Refatoração	8
5.1	Metodologia Aplicada	8
5.2	Exemplo Detalhado de Refatoração	8
5.2.1	Código Original (Com Smells)	8
5.2.2	Código Refatorado (Limpo)	8
5.3	Decisões de Refatoração	9
5.4	Outros Testes Refatorados	9
5.4.1	Correção do Exception Handling	9
5.4.2	Separação do Eager Test	10
6	Resultados e Validação	11
6.1	Métricas Comparativas	11
6.2	Execução dos Testes	11
6.3	Benefícios Alcançados	11
7	Conclusão	12
7.1	Principais Aprendizados	12
7.2	Impacto na Sustentabilidade do Software	12
7.3	Considerações Finais	12
8	Referências	13

1 Introdução

O presente trabalho tem como objetivo identificar, analisar e refatorar *Test Smells* (maus cheiros em testes) em uma suíte de testes de software. *Test Smells* são sintomas no código de teste que indicam problemas mais profundos, tornando os testes frágeis, obscuros, lentos e incapazes de detectar bugs de forma eficaz.

Utilizando ferramentas de análise estática como o ESLint, combinadas com análise manual, este trabalho demonstra como a qualidade dos testes pode ser significativamente melhorada através da aplicação de boas práticas e padrões como o *Arrange, Act, Assert* (AAA).

1.1 Contexto do Projeto

O projeto base consiste em um serviço de gerenciamento de usuários (`UserService`), implementado em JavaScript/Node.js, com testes utilizando o framework Jest. A suíte de testes original (`userService.smelly.test.js`) foi deliberadamente escrita com diversos *Test Smells* para fins didáticos.

1.2 Objetivos

- Identificar manualmente diferentes tipos de *Test Smells*
- Configurar ferramentas de análise estática para detecção automatizada
- Refatorar testes aplicando o padrão AAA e boas práticas
- Validar a melhoria através da eliminação de avisos do linter
- Demonstrar a importância de testes limpos para a sustentabilidade do software

2 Análise de Test Smells Identificados

Durante a análise da suíte de testes original, foram identificados cinco tipos distintos de *Test Smells*, cada um com implicações específicas para a qualidade e manutenibilidade dos testes.

2.1 Conditional Test Logic (Lógica Condisional em Testes)

Localização: Teste “deve desativar usuários se eles não forem administradores”

Descrição: Este smell ocorre quando testes contêm estruturas de controle de fluxo como loops (`for`, `while`) ou condicionais (`if`, `else`). O teste analisado utiliza um loop `for` para iterar sobre diferentes tipos de usuários e estruturas `if/else` para verificar comportamentos específicos.

Riscos e Implicações:

- Dificulta o entendimento do que está sendo testado
- Cria múltiplos caminhos de execução, tornando o teste imprevisível
- Alguns caminhos podem nunca ser executados
- Viola o princípio de responsabilidade única
- Quebra o padrão Arrange-Act-Assert
- Dificulta a identificação de falhas específicas

Código Problemático:

```

1 test('deve desativar usuarios se eles nao forem administradores', () =>
2   {
3     const usuarioComum = userService.createUser('Comum', 'comum@teste.com',
4       30);
5     const usuarioAdmin = userService.createUser('Admin', 'admin@teste.com',
6       40, true);
7     const todosOsUsuarios = [usuarioComum, usuarioAdmin];
8
9     for (const user of todosOsUsuarios) {
10       const resultado = userService.deactivateUser(user.id);
11       if (!user.isAdmin) {
12         expect(resultado).toBe(true);
13         const usuarioAtualizado = userService.getUserById(user.id);
14         expect(usuarioAtualizado.status).toBe('inativo');
15       } else {
16         expect(resultado).toBe(false);
17       }
18     }
19   });
20 
```

Listing 1: Teste com lógica condicional

2.2 Eager Test (Teste Ansioso)

Localização: Teste “deve criar e buscar um usuário corretamente”

Descrição: Um *Eager Test* é aquele que testa múltiplas funcionalidades ou comportamentos em um único caso de teste. No exemplo identificado, um único teste verifica tanto a criação quanto a busca de usuários.

Riscos e Implicações:

- Reduz a granularidade dos testes
- Dificulta identificar qual funcionalidade falhou
- Mistura conceitos e responsabilidades diferentes
- Viola o princípio de responsabilidade única
- Torna os testes menos precisos e informativos

2.3 Sensitive Equality (Igualdade Sensível)

Localização: Teste “deve gerar um relatório de usuários formatado”

Descrição: Testes que verificam detalhes muito específicos da implementação, particularmente formatação exata de strings, são considerados frágeis. Qualquer mudança cosmética na formatação quebra o teste, mesmo que a funcionalidade permaneça correta.

Riscos e Implicações:

- Alto acoplamento com detalhes de implementação
- Quebra facilmente com mudanças não funcionais
- Desencoraja refatoração
- Gera falsos negativos (teste falha mas funcionalidade está correta)
- Aumenta custo de manutenção

2.4 Exception Handling (Tratamento Manual de Exceções)

Localização: Teste “deve falhar ao criar usuário menor de idade”

Descrição: O uso de blocos `try-catch` manuais para verificar exceções em testes é um anti-pattern. Se a exceção não for lançada, o teste passa silenciosamente, criando uma falsa sensação de segurança.

Riscos e Implicações:

- Teste pode passar mesmo quando deveria falhar
- Não garante que a exceção foi realmente lançada
- Código verboso e menos legível
- Dificulta a manutenção

Código Problemático:

```

1 try {
2   userService.createUser('Menor', 'menor@email.com', 17);
3 } catch (e) {
4   expect(e.message).toBe('O usuario deve ser maior de idade.');
5 }
6 // Se a excecao nao for lancada, o teste passa silenciosamente!

```

Listing 2: Tratamento incorreto de exceções

2.5 Disabled Test (Teste Desabilitado)

Localização: Último teste com `test.skip`

Descrição: Testes comentados ou desabilitados representam código morto na suíte de testes e funcionalidades sem cobertura adequada.

Riscos e Implicações:

- Funcionalidade sem cobertura de testes

- Código morto no repositório
- Falsa sensação de cobertura
- Testes incompletos esquecidos indefinidamente

3 Configuração da Ferramenta de Detecção Automatizada

Para automatizar a detecção de *Test Smells*, foi configurado o ESLint versão 9 com o plugin específico para Jest.

3.1 Instalação das Dependências

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

Listing 3: Instalação do ESLint e plugin Jest

3.2 Configuração do ESLint

O arquivo `eslint.config.js` foi criado com as seguintes configurações:

```
1 import jest from 'eslint-plugin-jest';
2
3 export default [
4   {
5     files: ['**/*.js'],
6     languageOptions: {
7       ecmaVersion: 'latest',
8       sourceType: 'module',
9       globals: {
10         require: 'readonly',
11         module: 'readonly',
12         describe: 'readonly',
13         test: 'readonly',
14         expect: 'readonly',
15         beforeEach: 'readonly',
16       },
17     },
18     plugins: {
19       jest,
20     },
21     rules: {
22       ...jest.configs.recommended.rules,
23       'jest/no-disabled-tests': 'warn',
24       'jest/no-conditional-expect': 'error',
25       'jest/no-identical-title': 'error',
26     },
27   },
28 ];
```

Listing 4: Configuração do ESLint v9

3.3 Regras Configuradas

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

4 Relatório da Ferramenta de Análise Estática

4.1 Primeira Execução - Arquivo Original

Ao executar o ESLint no arquivo original contendo os *Test Smells*:

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

Listing 5: Comando de análise

Resultado obtido:

```
C:\Users\Nery\Documents\test-smelly\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)
```

4.2 Análise dos Resultados

A ferramenta detectou automaticamente:

- **4 erros** relacionados a expects condicionais (linhas 44, 46, 49, 73)
- **2 avisos** sobre teste desabilitado (linha 77)
- Total de **6 problemas** identificados

Estes resultados confirmam a análise manual realizada anteriormente, demonstrando a eficácia da ferramenta na detecção automatizada de *Test Smells*.

4.3 Segunda Execução - Arquivo Refatorado

Após a refatoração completa:

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

Listing 6: Validação do arquivo refatorado

Resultado: Nenhum problema encontrado (output vazio).

Este resultado comprova que todos os *Test Smells* foram eliminados com sucesso.

5 Processo de Refatoração

5.1 Metodologia Aplicada

A refatoração foi conduzida seguindo os seguintes princípios:

1. **Padrão AAA (Arrange-Act-Assert):** Estruturação clara de cada teste
2. **Responsabilidade Única:** Um teste para cada comportamento
3. **Nomes Descritivos:** Títulos que explicam o comportamento esperado
4. **Eliminação de Lógica:** Remoção de loops e condicionais
5. **Isolamento:** Testes independentes e autocontidos

5.2 Exemplo Detalhado de Refatoração

5.2.1 Código Original (Com Smells)

```

1 test('deve desativar usuarios se eles nao forem administradores', () =>
2   {
3     const usuarioComum = userService.createUser('Comum', 'comum@teste.com',
4       30);
5     const usuarioAdmin = userService.createUser('Admin', 'admin@teste.com',
6       40, true);
7     const todosOsUsuarios = [usuarioComum, usuarioAdmin];
8
9     for (const user of todosOsUsuarios) {
10       const resultado = userService.deactivateUser(user.id);
11       if (!user.isAdmin) {
12         expect(resultado).toBe(true);
13         const usuarioAtualizado = userService.getUserById(user.id);
14         expect(usuarioAtualizado.status).toBe('inativo');
15       } else {
16         expect(resultado).toBe(false);
17       }
18     }
19   });
20 
```

Listing 7: Teste com múltiplos smells

5.2.2 Código Refatorado (Limpo)

```

1 test('deve desativar usuario comum', () => {
2   // Arrange
3   const usuario = userService.createUser('Comum', 'comum@teste.com', 30)
4   ;
5
6   // Act
7   const resultado = userService.deactivateUser(usuario.id);
8
9   // Assert
10  expect(resultado).toBe(true);
11  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,
17     true);
18
19   // Act
20   const resultado = userService.deactivateUser(admin.id);
21
22   // Assert
23   expect(resultado).toBe(false);
24   const adminAtualizado = userService.getUserById(admin.id);
25   expect(adminAtualizado.status).toBe('ativo');
26 });

```

Listing 8: Testes refatorados seguindo AAA

5.3 Decisões de Refatoração

1. **Separação de Cenários:** O teste original foi dividido em dois testes independentes, cada um focado em um único comportamento (desativar usuário comum vs. não desativar admin).
2. **Eliminação de Lógica Condicional:** O loop `for` e as estruturas `if/else` foram completamente removidos, tornando o fluxo de execução linear e previsível.
3. **Aplicação do Padrão AAA:** Cada teste agora segue claramente as três fases:
 - **Arrange:** Preparação dos dados (criação do usuário)
 - **Act:** Execução da ação (chamada de `deactivateUser`)
 - **Assert:** Verificação dos resultados esperados
4. **Nomes Descritivos:** Cada teste possui um nome que descreve exatamente o comportamento esperado, sem ambiguidades.
5. **Isolamento:** Os testes são completamente independentes e podem ser executados em qualquer ordem.

5.4 Outros Testes Refatorados

5.4.1 Correção do Exception Handling

Antes:

```

1 try {
2   userService.createUser('Menor', 'menor@email.com', 17);
3 } catch (e) {
4   expect(e.message).toBe('O usuario deve ser maior de idade.');
5 }

```

Depois:

```
1 test('deve lancar erro quando usuario e menor de idade', () => {
2   expect(() => {
3     userService.createUser('Menor', 'menor@email.com', 17);
4   }).toThrow('O usuario deve ser maior de idade.');
5 });
```

5.4.2 Separação do Eager Test

O teste original que verificava criação E busca foi separado em:

```
1 test('deve criar um usuario com dados validos', () => {
2   const usuario = userService.createUser('Fulano', 'fulano@teste.com',
3                                         25);
4
5   expect(usuario).toBeDefined();
6   expect(usuario.id).toBeDefined();
7   expect(usuario.nome).toBe('Fulano');
8   expect(usuario.status).toBe('ativo');
9 });
10
11 test('deve retornar usuario quando ID existe', () => {
12   const usuarioCriado = userService.createUser('Fulano', 'fulano@teste.
13   com', 25);
14
15   const usuarioBuscado = userService.getUserById(usuarioCriado.id);
16
17   expect(usuarioBuscado).toBeDefined();
18   expect(usuarioBuscado.nome).toBe('Fulano');
19 });
```

6 Resultados e Validação

6.1 Métricas Comparativas

Métrica	Arquivo Original	Arquivo Refatorado
Número de testes	5	14
Problemas ESLint	6 (4 errors, 2 warnings)	0
Test Smells identificados	5	0
Testes desabilitados	1	0
Padrão AAA aplicado	Parcial	100%
Lógica condicional	Sim	Não

Tabela 1: Comparaçāo entre arquivo original e refatorado

6.2 Execução dos Testes

Todos os testes foram executados com sucesso:

```
PASS  test/userService.clean.test.js
PASS  test/userService.smelly.test.js

Test Suites: 2 passed, 2 total
Tests:       1 skipped, 18 passed, 19 total
Snapshots:   0 total
Time:        0.689 s
```

6.3 Benefícios Alcançados

- **Clareza:** Cada teste é autoexplicativo e fácil de entender
- **Manutenibilidade:** Alterações são mais simples e seguras
- **Confiabilidade:** Testes mais robustos e menos frágeis
- **Granularidade:** Melhor identificação de falhas específicas
- **Documentação:** Os testes servem como documentação viva do sistema

7 Conclusão

Este trabalho demonstrou na prática como *Test Smells* comprometem significativamente a qualidade e eficácia de uma suíte de testes. Mesmo com cobertura de código adequada, testes mal escritos podem falhar em detectar bugs e criar uma falsa sensação de segurança.

7.1 Principais Aprendizados

1. **Testes de qualidade vão além da cobertura:** A presença de *Test Smells* torna testes ineficazes, independentemente da porcentagem de cobertura alcançada.
2. **Ferramentas automatizadas são essenciais:** O ESLint demonstrou ser uma ferramenta valiosa para detecção automatizada de problemas, economizando tempo e garantindo consistência.
3. **Padrões como AAA são fundamentais:** A aplicação rigorosa do padrão Arrange-Act-Assert transforma testes confusos em código limpo e legível.
4. **Simplicidade é crucial:** A eliminação de lógica condicional e loops torna os testes mais previsíveis e confiáveis.
5. **Testes são ativos de longo prazo:** Investir em testes limpos reduz significativamente o custo de manutenção ao longo do ciclo de vida do software.

7.2 Impacto na Sustentabilidade do Software

A refatoração realizada contribui para a sustentabilidade do projeto de várias formas:

- **Redução de falsos positivos/negativos:** Testes mais precisos aumentam a confiança na suíte
- **Facilitação de refatorações futuras:** Testes robustos permitem mudanças com segurança
- **Onboarding mais rápido:** Novos desenvolvedores entendem o sistema através dos testes
- **Documentação viva:** Os testes descrevem o comportamento esperado do sistema
- **Menor custo de manutenção:** Código limpo é mais barato de manter

7.3 Considerações Finais

A combinação de análise manual, ferramentas automatizadas e aplicação rigorosa de boas práticas demonstrou ser extremamente eficaz na melhoria da qualidade dos testes. O ESLint, configurado adequadamente, automatiza a detecção de problemas comuns, enquanto a aplicação consciente de padrões como AAA garante testes claros, concisos e efetivos.

Este trabalho reforça a importância de tratar testes como código de produção, merecendo o mesmo nível de cuidado, revisão e refatoração que o código funcional. Testes limpos não são apenas uma boa prática, são essenciais para a qualidade e sustentabilidade de qualquer projeto de software.

8 Referências

- FOWLER, Martin. TestSmell. Disponível em: <https://martinfowler.com/bliki/TestSmell.html>. Acesso em: 02 nov. 2025.
- JEST COMMUNITY. eslint-plugin-jest. GitHub. Disponível em: <https://github.com/jest-community/eslint-plugin-jest>. Acesso em: 02 nov. 2025.
- MESZAROS, Gerard. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley Professional, 2007.
- VAN DEURSEN, Arie et al. Refactoring Test Code. In: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), 2001.
- Repositório do projeto: <https://github.com/LuizNeryyy/test-smelly>