Refatoração de Testes e Detecção de Test Smells

Teste de Software

Arthur Ferreira Costa

Novembro de 2025

1. Introdução

Este relatório apresenta uma análise detalhada de Test Smells (más práticas em testes automatizados) identificados em uma suíte de testes JavaScript utilizando Jest. O trabalho envolveu a detecção manual e automatizada de smells, seguida pela refatoração completa dos testes para eliminar essas más práticas.

O projeto analisado consiste em um serviço de gerenciamento de usuários (`UserService`) com funcionalidades de criação, busca, desativação e geração de relatórios. A suíte de testes original (`userService.smelly.test.js`) apresentava diversos problemas que comprometiam sua manutenibilidade, confiabilidade e clareza.

2. Análise de Test Smells

2.1 Conditional Expect

Descrição

O smell "Conditional Expect" ocorre quando asserções (`expect`) são colocadas dentro de estruturas condicionais como `if/else`, loops `for`, ou blocos `try/catch`. Este foi o smell mais prevalente no código analisado, com 4 ocorrências detectadas pelo ESLint.

Exemplo do Código Original

```
test('deve falhar ao criar usuário menor de idade', () => {
  try {
    userService.createUser('Menor', 'menor@email.com', 17);
  } catch (e) {
    expect(e.message).toBe('O usuário deve ser maior de idade.');
  }
});
```

Por que é um "Mau Cheiro"

Este padrão é problemático porque: - **Execução Condicional**: A asserção só executa se o bloco `catch` for atingido - **Falso Positivo**: Se a exceção não for lançada (indicando um bug), o teste passa silenciosamente sem fazer nenhuma verificação - **Imprevisibilidade**: Não há garantia de que as asserções foram executadas

Riscos

- Testes que passam mesmo quando o código está quebrado
- Dificuldade em identificar quando uma asserção não foi executada
- Manutenção complexa e propensa a erros
- Falsa sensação de segurança na cobertura de testes

2.2 Eager Test (Teste Ansioso)

Descrição

Um Eager Test é um teste que verifica múltiplas funcionalidades ou comportamentos em um único método de teste, violando o princípio de responsabilidade única.

Exemplo do Código Original

```
test('deve criar e buscar um usuário corretamente', () => {
  // Act 1: Criar
  const usuarioCriado = userService.createUser(
    dadosUsuarioPadrao.nome,
    dadosUsuarioPadrao.email,
    dadosUsuarioPadrao.idade
  );
  expect(usuarioCriado.id).toBeDefined();

  // Act 2: Buscar
  const usuarioBuscado = userService.getUserById(usuarioCriado.id);
  expect(usuarioBuscado.nome).toBe(dadosUsuarioPadrao.nome);
  expect(usuarioBuscado.status).toBe('ativo');
});
```

Por que é um "Mau Cheiro"

- **Múltiplas Responsabilidades**: Testa tanto a criação quanto a busca de usuários
- **Diagnóstico Difícil**: Se o teste falhar, não fica claro qual funcionalidade está quebrada
- **Dependência Entre Funcionalidades**: A falha em criar afeta o teste de busca
- **Violação do Princípio SRP**: Single Responsibility Principle

Riscos

- Dificuldade em identificar a causa raiz de falhas
- Testes menos granulares e informativos
- Mudanças em uma funcionalidade podem quebrar testes de outras
- Relatórios de teste menos precisos

2.3 Lógica Condicional em Testes

Descrição

Este smell ocorre quando testes contêm estruturas de controle como loops `for`, condicionais `if/else`, ou qualquer lógica algorítmica complexa.

Exemplo do Código Original

```
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);
    }
  }
});
```

Por que é um "Mau Cheiro"

- **Complexidade Desnecessária**: Testes devem ser simples e diretos
- **Dificulta Leitura**: Requer análise mental para entender o que está sendo testado
- **Possível Fonte de Bugs**: A própria lógica do teste pode conter erros
- **Viola o Padrão AAA**: Arrange-Act-Assert fica confuso

Riscos

- Bugs no próprio código de teste
- Dificuldade em manter e evoluir os testes
- Tempo maior para compreender o propósito do teste
- Pode mascarar falhas em cenários específicos

3. Processo de Refatoração

3.1 Teste Problemático Selecionado

Escolhi o teste de desativação de usuários como exemplo por combinar múltiplos smells: **Conditional Expect**, **Lógica Condicional** e implicitamente **Eager Test**.

3.2 Código ANTES (Smelly)

```
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);
    }
  }
});
```

Problemas Identificados: - Loop `for` iterando sobre múltiplos casos - Condicional `if/else` determinando quais asserções executar - 3 ocorrências de `expect` condicional (linhas 44, 46, 49) - ESLint reporta 3 erros de `jest/no-conditional-expect` - Testa dois cenários distintos em um único teste

3.3 Código DEPOIS (Clean)

```
describe('deactivateUser', () => {
  test('should deactivate a regular user successfully', () => {
    // Arrange
    const usuarioComum = userService.createUser('Comum', 'comum@teste.com', 30);

    // Act
    const resultado = userService.deactivateUser(usuarioComum.id);

    // Assert
```

```
    expect(resultado).toBe(true);
    const usuarioAtualizado = userService.getUserById(usuarioComum.id);
    expect(usuarioAtualizado.status).toBe('inativo');
  });

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

    // Act
    const resultado = userService.deactivateUser(usuarioAdmin.id);

    // Assert
    expect(resultado).toBe(false);
    const usuarioAtualizado = userService.getUserById(usuarioAdmin.id);
    expect(usuarioAtualizado.status).toBe('ativo');
  });

  test('should return false when deactivating non-existent user', () => {
    // Arrange
    const idInvalido = 'id-que-nao-existe';

    // Act
    const resultado = userService.deactivateUser(idInvalido);

    // Assert
    expect(resultado).toBe(false);
  });
});
```

3.4 Decisões de Refatoração

1. Separação em Testes Independentes

Decisão: Dividir um teste em três testes focados.

Justificativa: Cada teste agora tem uma única responsabilidade, facilitando identificar qual cenário falhou. Adicionei também um terceiro teste para o caso de usuário inexistente, aumentando a cobertura.

2. Eliminação de Lógica Condicional

Decisão: Remover completamente o loop `for` e as estruturas `if/else`.

Justificativa: Testes devem ser declarativos, não algorítmicos. A remoção da lógica torna os testes mais fáceis de ler e menos propensos a bugs.

3. Aplicação Rigorosa do Padrão AAA

Decisão: Estruturar cada teste com comentários explícitos para Arrange, Act, Assert.

Justificativa: Clareza na organização do teste. Qualquer desenvolvedor pode rapidamente entender: - **Arrange**: O que está sendo preparado (criação do usuário) - **Act**: Qual ação está sendo testada (desativação) - **Assert**: O que deve ser verificado (resultado e status)

4. Agrupamento Lógico com `describe`

Decisão: Envolver os testes relacionados em um bloco `describe('deactivateUser')`.

Justificativa: Organização hierárquica que agrupa testes da mesma funcionalidade, melhorando a navegação e a execução seletiva.

5. Nomes Descritivos em Inglês

Decisão: Usar padrão "should [expected behavior]" para nomear os testes.

Justificativa: Convenção amplamente adotada na comunidade, auto-documentação clara do comportamento esperado.

3.5 Impacto da Refatoração

Aspecto	Antes	Depois
Número de testes	1	3
Linhas de código	~15	~35
Complexidade ciclomática	Alta (loop + if)	Baixa (linear)
ESLint errors	3	0
Clareza	Baixa	Alta
Manutenibilidade	Difícil	Fácil

Observação Importante: Embora o número de linhas tenha aumentado, a qualidade, clareza e manutenibilidade melhoraram significativamente. Código de teste deve priorizar clareza sobre concisão.

4. Relatório da Ferramenta ESLint

4.1 Primeira Execução - Detecção de Erros

A execução do comando `npx eslint test/userService.smelly.test.js` na primeira análise resultou em:

```
/Users/oarthurfc/faculdade/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

Erros Detectados (4)

Todos relacionados à regra `jest/no-conditional-expect`:

- **Linhas 44, 46, 49**: Dentro do loop `for` com condicional `if/else`
- **Linha 73**: Dentro do bloco `catch` de um `try/catch`

Avisos Detectados (2)

- **Linha 77 - `jest/no-disabled-tests`**: Teste marcado com `test.skip()`
- **Linha 77 - `jest/expect-expect`**: Teste sem asserções (apenas comentário TODO)

4.3 Automação da Detecção

Configuração do ESLint

Para automatizar a detecção, configurei o ESLint com o plugin Jest (`eslint-plugin-jest`):

```
// eslint.config.js
export default [
  {
    files: ["test/**/*.js", "__tests__/**/*.js"],
    plugins: {
      jest: jestPlugin,
    },
    rules: {
      ...jestPlugin.configs.recommended.rules,
      "jest/no-disabled-tests": "warn",
      "jest/no-conditional-expect": "error",
```

```
      "jest/no-identical-title": "error",
    }
  }
];
```

Benefícios da Automação

1. **Detecção Imediata**: Problemas identificados antes mesmo do commit
2. **Consistência**: Mesmas regras aplicadas em todo o projeto
3. **Educação**: Mensagens claras explicam o problema
4. **Integração CI/CD**: Pode bloquear merges com test smells
5. **Economia de Tempo**: Revisão manual seria lenta e propensa a falhas

Comparação: Manual vs. Automatizado

Aspecto	Detecção Manual	ESLint Automatizado
Tempo	~30 minutos	~2 segundos
Cobertura	Parcial	Completa
Consistência	Variável	100% consistente
Escalabilidade	Não escala	Escala infinitamente
Custo	Alto (tempo humano)	Baixo (uma vez configurado)

4.4 Validação Pós-Refatoração

Após a refatoração, a execução do ESLint no arquivo limpo resultou em:

```
$ npx eslint __tests__/userService.clean.test.js

(sem saída - nenhum erro ou aviso)
```

Resultado: **0 erros, 0 avisos**

Isso confirma que todos os test smells foram eliminados com sucesso.

5. Conclusão

5.1 Importância dos Testes Limpos

A experiência prática deste trabalho demonstrou concretamente que **testes são código** e, como tal, devem ser tratados com o mesmo rigor e cuidado que o código de produção. Testes sujos não apenas falham em fornecer a segurança necessária, mas podem ativamente prejudicar o projeto ao criar:

- **Falsa sensação de segurança**: Testes que passam mesmo quando o código está quebrado
- **Débito técnico**: Testes difíceis de manter tornam-se um fardo
- **Resistência à mudança**: Desenvolvedores evitam refatorar código com testes frágeis
- **Tempo desperdiçado**: Debugging de testes problemáticos consome recursos valiosos

5.2 Valor das Ferramentas de Análise Estática

A utilização do ESLint com o plugin Jest provou ser **essencial** para:

1. Detecção Proativa

Identificar problemas antes que causem impacto, durante o desenvolvimento e não em produção.

2. Padronização

Garantir que toda a equipe siga as mesmas boas práticas, independentemente da experiência individual.

3. Educação Contínua

As mensagens de erro do ESLint servem como micro-treinamentos, ensinando desenvolvedores sobre best practices.

4. Integração com Workflow

Ferramentas automatizadas integram-se perfeitamente em pipelines CI/CD, garantindo qualidade contínua.

5.3 Impacto na Qualidade e Sustentabilidade

A refatoração realizada transformou 5 testes problemáticos em 11 testes limpos e focados. Embora isso represente um aumento no volume de código, os benefícios são significativos:

Qualidade

- **Confiabilidade**: Cada teste verifica exatamente o que afirma verificar

- **Precisão**: Falhas apontam diretamente para o problema
- **Cobertura**: Cenários edge cases agora têm testes dedicados

Sustentabilidade

- **Manutenibilidade**: Testes simples são fáceis de modificar
- **Documentação Viva**: Testes descrevem claramente o comportamento esperado
- **Refatoração Segura**: Base sólida para evoluir o código com confiança

Métricas do Projeto

- **ESLint Errors**: $4 \rightarrow 0$ (100% de redução)
- **ESLint Warnings**: $2 \rightarrow 0$ (100% de redução)
- **Testes Implementados**: $80\% \rightarrow 100\%$
- **Testes com Lógica Condicional**: $2 \rightarrow 0$

5.4 Lições Aprendidas

1. **Testes são investimento, não custo**: Tempo gasto escrevendo testes limpos economiza múltiplos esse tempo em debugging futuro
2. **Automação é força multiplicadora**: Ferramentas como ESLint permitem manter qualidade em escala
3. **Simplicidade é chave**: Testes devem ser tão simples que bugs neles sejam virtualmente impossíveis
4. **AAA não é opcional**: O padrão Arrange-Act-Assert transforma testes em documentação clara
5. **Test Smells são evitáveis**: Com conhecimento e ferramentas adequadas, podemos escrever testes limpos desde o início

5.5 Reflexão Final

Este trabalho evidenciou que a qualidade de software não se limita ao código de produção. **Testes limpos são tão importantes quanto código limpo**. A combinação de boas práticas de escrita de testes (padrão AAA, responsabilidade única, nomes descritivos) com ferramentas de análise estática (ESLint, plugins especializados) cria um ecossistema onde a qualidade é mantida continuamente e de forma sustentável.

Em projetos reais, onde a complexidade é maior e as equipes são distribuídas, essas práticas deixam de ser "boas de ter" e tornam-se **essenciais para a sobrevivência do projeto**. A capacidade de refatorar com confiança, adicionar funcionalidades sem medo, e onboard novos desenvolvedores rapidamente depende diretamente da qualidade da suíte de testes.

O investimento em testes limpos e ferramentas de análise estática é, portanto, um investimento na longevidade e sucesso do projeto de software.

Fim do Relatório