

Disciplina: Teste de Software

Trabalho: Test Smell

Nome: Matheus Hoske Aguiar

Matrícula: 728000

Data: 02/11/2025

Repositório: <https://github.com/matheushoske/test-smelly.git>

RELATÓRIO DE TRABALHO

1. Análise de Test Smells

Após análise manual e automática da suíte de testes original (`userService.smelly.test.js`), foram identificados diversos Test Smells que comprometem a qualidade, manutenibilidade e confiabilidade dos testes. A seguir, são detalhados três dos principais problemas encontrados:

1.1 Conditional Test Logic (Lógica Condicional em Testes)

Localização: Linhas 40-51 do arquivo `userService.smelly.test.js`

Descrição do Problema: O teste "deve desativar usuários se eles não forem administradores" utiliza uma estrutura de loop (`for`) combinada com condicionais (`if/else`) dentro do corpo do teste. Os assertions (`expect`) são executados condicionalmente, dependendo do valor da propriedade `isAdmin` de cada usuário.

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

Por que é um Test Smell: A presença de lógica condicional dentro de testes viola o princípio de simplicidade e clareza. Testes devem ser diretos e fáceis de entender, sem complexidade algorítmica. Quando um teste contém condicionais e loops, ele se torna:

- **Difícil de ler e compreender:** Um desenvolvedor precisa "executar mentalmente" o código para entender o que está sendo testado
- **Frágil:** Se a lógica do teste contiver bugs, o teste pode passar incorretamente ou falhar sem motivo claro
- **Ambíguo:** Não fica claro qual cenário específico está falhando quando o teste quebra

Riscos:

1. **Mascaramento de bugs:** Se o `if` falhar silenciosamente, parte das assertions podem não ser executadas, fazendo o teste passar mesmo com comportamento incorreto

2. **Debugging complexo:** Quando o teste falha, é necessário investigar qual iteração do loop e qual ramo do `if` causou a falha
3. **Manutenibilidade reduzida:** Qualquer alteração na estrutura condicional pode introduzir novos bugs nos próprios testes

1.2 Fragile Test (Teste Frágil)

Localização: Linhas 54-64 do arquivo `userService.smelly.test.js`

Descrição do Problema: O teste "deve gerar um relatório de usuários formatado" depende fortemente da formatação exata da string retornada pelo método `generateUserReport()`. O teste verifica não apenas o conteúdo, mas também a formatação específica, incluindo espaços, quebras de linha e ordem exata dos elementos.

```
const linhaEsperada = `ID: ${usuario1.id}, Nome: Alice, Status: ativo\n`;
expect(relatorio).toContain(linhaEsperada);
```

Por que é um Teste Frágil: Um teste frágil quebra com frequência, não porque o comportamento do código sob teste mudou de forma significativa, mas porque detalhes de implementação foram alterados. Neste caso, qualquer mudança na formatação (espaços extras, ordem dos campos, estilo de formatação) fará o teste falhar, mesmo que o método continue gerando um relatório válido e funcionalmente correto.

Riscos:

1. **Falsos negativos:** O teste falha mesmo quando a funcionalidade principal está correta, gerando ruído e desconfiança na suíte de testes
2. **Custo de manutenção:** Desenvolvedores precisam atualizar testes constantemente devido a mudanças cosméticas, não funcionais
3. **Resistência a melhorias:** A equipe pode evitar melhorar a formatação do relatório por medo de quebrar testes, impedindo evolução do código

1.3 Test without Assertions (Teste sem Assertions)

Localização: Linhas 66-75 e 77-79 do arquivo `userService.smelly.test.js`

Descrição do Problema: Dois problemas distintos relacionados a falta de assertions adequadas:

a) Try-Catch sem garantia de falha (linhas 66-75): O teste "deve falhar ao criar usuário menor de idade" usa um bloco `try/catch` sem verificar se a exceção foi realmente lançada. Se o código de validação for removido ou modificado e a exceção não for lançada, o teste passará silenciosamente, mascarando um bug crítico.

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

b) Teste desabilitado (linhas 77-79): Um teste foi marcado com `test.skip`, tornando-o inativo permanentemente. Esse teste não possui assertions e nunca será executado.

Por que é um Test Smell: Testes existem para verificar comportamentos e detectar regressões. Quando um teste não garante que o comportamento esperado ocorra, ou quando é desabilitado, ele não cumpre sua função principal:

- **Falsa confiança:** O teste passa mesmo quando há bugs, dando sensação errônea de segurança
- **Código morto:** Testes desabilitados acumulam-se no código e raramente são implementados posteriormente
- **Documentação enganosa:** A presença de testes que não testam nada cria confusão sobre o que está realmente coberto

Riscos:

1. **Bugs não detectados:** Regressões podem ser introduzidas sem que os testes falhem, comprometendo a qualidade do produto
2. **Deterioração da suíte:** Testes que não agregam valor poluem a base de código e dificultam a manutenção
3. **Custos de correção:** Bugs descobertos em produção são muito mais caros de corrigir do que quando detectados durante o desenvolvimento

2. Processo de Refatoração

Para demonstrar o processo de refatoração, foi escolhido o teste mais problemático da suíte original: **"deve desativar usuários se eles não forem administradores"**. Este teste apresenta múltiplos Test Smells simultaneamente, incluindo Conditional Test Logic, Eager Test (teste que verifica múltiplas coisas) e baixa legibilidade.

2.1 Código Original (Antes)

```
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:

1. **Conditional Test Logic:** Uso de `if/else` dentro do teste com assertions condicionais
2. **Eager Test:** O teste verifica dois comportamentos diferentes (desativação de usuário comum e proteção de admin) em um único teste
3. **Baixa clareza:** É necessário entender a lógica do loop e condicional para saber o que está sendo testado
4. **Falta de organização:** Não segue o padrão AAA de forma clara

2.2 Código Refatorado (Depois)

```
describe('deactivateUser', () => {
  test('deve desativar usuário comum com sucesso', () => {
    const usuarioComum = userService.createUser('Comum', 'comum@teste.com', 30)

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

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

  test('deve retornar false ao tentar desativar usuário administrador', () => {
    const usuarioAdmin = userService.createUser('Admin', 'admin@teste.com', 40,

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

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

  test('deve retornar false ao tentar desativar usuário inexistente', () => {
    const idInexistente = 'id_que_nao_existe';

    const resultado = userService.deactivateUser(idInexistente);

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

2.3 Decisões de Refatoração e Justificativas

1. Separação em múltiplos testes focados: O teste original foi dividido em três testes independentes, cada um verificando um comportamento específico:

- Desativação bem-sucedida de usuário comum
- Proteção de usuários administradores
- Tratamento de usuário inexistente

Justificativa: Isso elimina o smell "Eager Test" e torna cada teste mais fácil de entender e manter. Quando um teste falha, fica imediatamente claro qual cenário específico está com problema.

2. Remoção completa de lógica condicional: Eliminados todos os `if/else` e loops do corpo dos testes. Cada teste agora é uma sequência linear e direta de Arrange, Act, Assert.

Justificativa: Remove o "Conditional Test Logic" smell. Os testes agora são determinísticos e todas as assertions são sempre executadas, garantindo que nenhum bug passe despercebido.

3. Aplicação rigorosa do padrão AAA (Arrange, Act, Assert): Cada teste está claramente dividido em três seções:

- **Arrange:** Preparação dos dados de teste (criação de usuários, IDs, etc.)
- **Act:** Execução única da ação sendo testada (chamada ao método `deactivateUser`)
- **Assert:** Verificações dos resultados esperados

Justificativa: O padrão AAA melhora significativamente a legibilidade e facilita a identificação de problemas. Qualquer desenvolvedor pode entender rapidamente o que cada teste faz.

4. Organização com `describe` blocks: Os testes relacionados foram agrupados em um bloco `describe('deactivateUser')`, criando uma hierarquia clara e facilitando a navegação na suíte de testes.

Justificativa: Melhora a organização e permite que ferramentas de relatório apresentem resultados de forma mais estruturada.

5. Adição de teste para caso de borda: Foi adicionado um teste para o caso de tentativa de desativação de usuário inexistente, aumentando a cobertura de testes.

Justificativa: Aumenta a confiança na robustez do código e garante que casos de borda são tratados corretamente.

6. Melhorias nas assertions: No teste do administrador, foi adicionada uma verificação explícita de que o status permanece 'ativo', tornando o teste mais completo e claro sobre o comportamento esperado.

Justificativa: Assertions mais completas reduzem ambiguidade e deixam claro o contrato do método.

2.4 Como os Smells foram corrigidos

- **Conditional Test Logic:** Eliminado completamente - nenhum `if/else` ou loop permanece nos testes
 - **Eager Test:** Corrigido - cada teste agora verifica um único comportamento
 - **Fragile Test:** Não aplicável a este teste específico, mas foi corrigido em outros testes da suíte
 - **Test without Assertions:** Corrigido - todos os testes têm assertions explícitas e garantidas
 - **Mystery Guest:** Melhorado - dados de teste são mais explícitos e próximos do seu uso
-

3. Relatório da Ferramenta

3.1 Primeira Execução do ESLint

Ao executar o ESLint no projeto pela primeira vez, após a configuração inicial, foram detectados **6 problemas** no arquivo `userService.smelly.test.js`:

```
C:\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)
```

Detalhamento dos problemas:

1. 4 erros de `jest/no-conditional-expect`:

- Linhas 44, 46, 49: Assertions dentro do bloco `if/else` no teste de desativação
- Linha 73: Assertion dentro do bloco `catch` (que é condicional)

2. 2 warnings:

- `jest/no-disabled-tests`: Teste marcado com `test.skip` na linha 77
- `jest/expect-expect`: O teste desabilitado não possui assertions

3.2 Como a Ferramenta Automatizou a Detecção

O ESLint com o plugin `eslint-plugin-jest` automatizou a detecção de problemas de várias formas:

1. Regras configuradas: As regras especificadas no `.eslintrc.json` permitiram detectar problemas específicos de testes:

- `jest/no-conditional-expect`: Detecta quando assertions são chamadas dentro de estruturas condicionais (if, loops, try/catch), garantindo que todos os expects sejam executados
- `jest/no-disabled-tests`: Identifica testes que foram desabilitados com `skip` ou `todo`, alertando sobre testes que não estão sendo executados
- `jest/expect-expect`: Garante que cada teste contenha pelo menos uma assertion, evitando testes que passam sem verificar nada

2. Análise estática automática: A ferramenta analisou o código sem necessidade de execução, identificando padrões problemáticos através de análise sintática e semântica. Isso é significativamente mais rápido e confiável do que análise manual.

3. Feedback imediato: O ESLint forneceu feedback preciso com:

- Localização exata do problema (arquivo, linha, coluna)
- Tipo de problema (error ou warning)
- Descrição clara da regra violada
- Código da regra para referência

4. Integração com o workflow: A ferramenta pode ser facilmente integrada em:

- Editores de código (para feedback em tempo real)
- Pre-commit hooks (para prevenir commits com problemas)
- Pipelines de CI/CD (para garantir qualidade antes do merge)

3.3 Validação Final

Após a refatoração, o ESLint foi executado novamente no arquivo `userService.clean.test.js`:

```
(nenhum erro ou aviso reportado)
Exit code: 0
```

O código refatorado não apresentou nenhum problema, confirmando que todos os Test Smells foram eliminados e que a suíte de testes agora segue as boas práticas recomendadas.

4. Conclusão

Este trabalho demonstrou a importância crítica da escrita de testes limpos e da utilização de ferramentas de análise estática para garantir a qualidade e sustentabilidade de projetos de software.

4.1 Benefícios dos Testes Limpos

A refatoração realizada evidenciou vários benefícios tangíveis:

Manutenibilidade: Testes limpos são mais fáceis de entender e modificar. Quando um requisito muda, desenvolvedores podem atualizar os testes rapidamente sem precisar decifrar lógica complexa. A separação em testes focados reduz o tempo necessário para identificar e corrigir problemas.

Confiabilidade: Testes sem lógica condicional garantem que todas as assertions sejam executadas, eliminando o risco de bugs passarem despercebidos. A aplicação consistente do padrão AAA torna os testes determinísticos e previsíveis.

Documentação viva: Testes bem escritos servem como documentação do comportamento esperado do sistema. Nomes descritivos e estrutura clara tornam os testes uma fonte confiável de informação sobre como o código deve funcionar.

Deteção de regressões: Testes focados e diretos falham de forma precisa quando regressões são introduzidas, permitindo localização rápida do problema. Isso reduz significativamente o tempo de debugging.

4.2 Papel das Ferramentas de Análise Estática

O ESLint, configurado com regras específicas para testes, provou ser uma ferramenta indispensável:

Automação: A detecção automática de problemas elimina a necessidade de revisão manual tediosa e propensa a erros. A ferramenta consegue analisar grandes bases de código em segundos, identificando padrões problemáticos que poderiam passar despercebidos.

Consistência: Ferramentas de linting garantem que toda a equipe siga os mesmos padrões de qualidade, independente da experiência individual. Isso cria uma cultura de qualidade compartilhada.

Prevenção: Ao integrar o linting no workflow de desenvolvimento (editores, pre-commit hooks, CI/CD), problemas são detectados antes mesmo de serem commitados, reduzindo o custo de correção e mantendo a base de código sempre limpa.

Educação: As mensagens de erro do ESLint educam desenvolvedores sobre boas práticas. Cada alerta é uma oportunidade de aprender e melhorar continuamente.

4.3 Impacto na Sustentabilidade do Projeto

A qualidade dos testes tem impacto direto na sustentabilidade de longo prazo:

Redução de débito técnico: Testes mal escritos se tornam um débito técnico que cresce exponencialmente. Cada vez que um desenvolvedor precisa trabalhar com testes "mal cheirosos", ele perde tempo e pode introduzir novos problemas. Investir em testes limpos desde o início reduz esse débito.

Velocidade de desenvolvimento: Uma suíte de testes confiável e rápida de executar permite que desenvolvedores trabalhem com confiança, fazendo refatorações e adicionando features sem medo de quebrar funcionalidades existentes. Isso acelera o desenvolvimento significativamente.

Custo de manutenção: Testes frágeis e difíceis de entender aumentam o custo de manutenção. Cada mudança no código requer atualização de múltiplos testes complexos, enquanto testes limpos e focados se adaptam mais facilmente a mudanças.

Qualidade do produto: No final, a qualidade dos testes se reflete diretamente na qualidade do produto entregue ao usuário. Testes eficazes detectam bugs antes da produção, melhorando a experiência do usuário e reduzindo custos de suporte e correção.

4.4 Reflexão Final

Este trabalho demonstrou que escrever testes limpos não é apenas uma questão de estética ou de seguir "regras" arbitrárias. É uma prática fundamental que impacta diretamente a capacidade de uma equipe de software entregar valor de forma sustentável e confiável.

A combinação de conhecimento teórico (identificação manual de smells) com ferramentas práticas (ESLint) cria um ambiente onde a qualidade é garantida sistematicamente. A análise estática complementa, mas não substitui, o bom julgamento e conhecimento do desenvolvedor. Juntos, eles formam uma abordagem poderosa para manter a qualidade do código ao longo do tempo.

A lição aprendida é clara: **investir tempo em escrever testes limpos e configurar ferramentas de análise estática não é um custo, é um investimento que se paga continuamente** através de desenvolvimento mais rápido, bugs menos frequentes e equipes mais produtivas.

Fim do Relatório