Refatoração de Testes e Detecção de Test Smells

Nome da Disciplina: TESTE DE SOFTWARE

Nome do Trabalho: Refatoração de Testes e Detecção de Test Smells

Nome Completo: Lúcio Alves Almeida Neto

Matrícula: 803696

---

1. Análise de Smells

Quais foram os 3 principais Test Smells (maus cheiros) encontrados na suíte de testes original e quais os riscos associados a eles?

Resposta:

1. Lógica Condicional:

> **O que é:** É o uso de estruturas como `if`/`else` ou `try`/`catch` dentro de um caso de teste, fazendo com que a verificação (`expect`) só seja executada se uma condição for atendida.
>
> **Exemplo no código:** Os testes `'deve desativar usuários...'` (com `if`/`else`) e `'deve falhar ao criar usuário menor de idade'` (com `try`/`catch`).
>
> **Risco:** O risco é o **falso positivo**. Um teste pode passar (ficar "verde") sem nunca ter, de fato, testado o comportamento esperado. Se a lógica da aplicação mudar e o `if` não for mais ativado, o teste continuará passando, escondendo um possível bug.

2. Teste Desabilitado:

> **O que é:** É um teste que foi explicitamente pulado pelo desenvolvedor, geralmente usando `test.skip` ou `xtest`.
>
> **Exemplo no código:** O teste `test.skip('deve retornar uma lista vazia...')`.
>
> **Risco:** É uma **dívida técnica**. O teste existe, mas não está sendo executado, deixando uma parte do comportamento da aplicação sem cobertura de testes. Com o tempo, a equipe pode esquecer de "despular" o teste, e uma regressão (um novo bug) pode ser introduzida nessa funcionalidade sem que ninguém perceba.

3. Teste Frágil:

> **O que é:** Um teste que quebra facilmente com mudanças irrelevantes no código-fonte, como uma alteração na formatação de uma string.

Exemplo no código: O teste `'deve gerar um relatório de usuários formatado'` verificava uma string exata: `const linhaEsperada = \ID: ${usuario1.id}, Nome: Alice, Status: ativo\n`;`.

Risco: Alto custo de manutenção. Se um dev adicionar um espaço extra ou mudar a ordem dos campos no relatório (uma mudança que não afeta a funcionalidade), o teste falhará. Isso faz com que a equipe perca tempo corrigindo testes que não encontraram bugs reais.

---

2. Processo de Refatoração

Pergunta: Demonstre o processo de refatoração de um teste problemático, mostrando o "Antes" e "Depois" e explicando as decisões tomadas.

Resposta:

O test problemático escolhido era o `'deve desativar usuários se eles não forem administradores'`, pois ele continha dois "smells" de uma vez: Lógica Condicional e Eager Test.

Antes (do arquivo `userService.smelly.test.js`)

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];



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

Depois (do arquivo `userService.clean.test.js`)

O teste foi dividido em dois testes menores, focados e sem lógica condicional, seguindo o padrão Arrange, Act, Assert (AAA).

JavaScript

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

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

```
  // Assert
  expect(resultado).toBe(true);
  expect(usuarioAtualizado.status).toBe('inativo');
});

test('NÃO deve desativar um usuário administrador', () => {
  // Arrange
  const usuarioAdmin = userService.createUser('Admin', 'admin@teste.com', 40,
true);

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

  // Assert
  expect(resultado).toBe(false);
  expect(usuarioAtualizado.status).toBe('ativo'); // Deve permanecer ativo
});
```

Explicação das Decisões de Refatoração

1. **Separação de Responsabilidade:** O teste original testava dois cenários de uma só vez. Isso foi quebrado em dois testes separados, onde cada um tem uma única responsabilidade.
2. **Eliminação da Lógica Condicional:** Ao separar os testes, o loop `for` e a estrutura `if`/`else` foram completamente removidos.

3. **Correção do "Conditional Expect":** A remoção do `if` garantiu que o `expect` sempre será executado em 100% das vezes, eliminando o erro `jest/no-conditional-expect` apontado pelo ESLint.
4. **Clareza (Padrão AAA):** Os novos testes são mais de boa de ler. É possível identificar imediatamente o que está sendo organizado, executado e verificado.

---

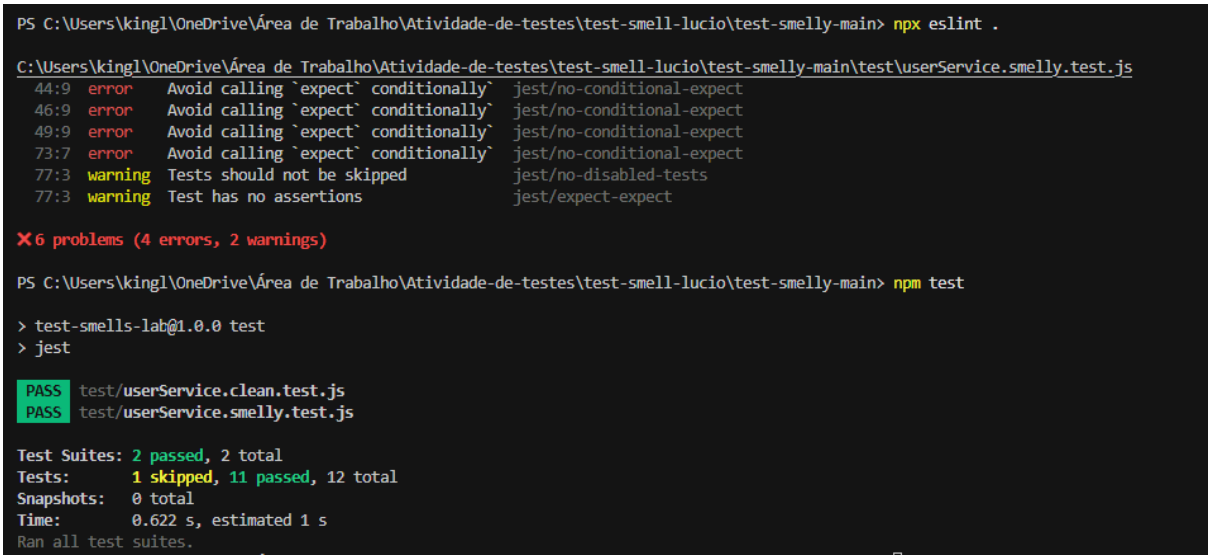
3. Relatório da Ferramenta

Pergunta: Como a ferramenta de análise estática (ESLint) auxiliou na detecção dos smells?

Resposta:

A ferramenta ESLint, configurada com o `eslint-plugin-jest`, foi fundamental para automatizar a detecção dos "maus cheiros". Enquanto a análise manual (Etapa 2) foi subjetiva, a ferramenta foi objetiva e precisa.

Abaixo está o print da primeira execução do linter, que apontou os problemas no código original:



Comentário sobre o resultado:

O ESLint automatizou a detecção de forma precisa:

Ele imediatamente encontrou **4 instâncias** de "expect condicional" (`jest/no-conditional-expect`), que são os `expect` dentro dos `if` e do `try/catch`.

Ele nos avisou sobre o teste desabilitado (`jest/no-disabled-tests`).

Ele serviu como uma ajuda, apontando exatamente quais testes precisavam de atenção.

---

4. Conclusão

Pergunta: Qual a importância de escrever testes limpos e utilizar ferramentas de análise estática para a qualidade e sustentabilidade de um projeto?

Resposta:

Este trabalho demonstrou que ter 100% de cobertura de testes não significa nada se os testes forem "mal cheirosos".

Escrever testes limpos é fundamental para a sustentabilidade do projeto. Testes limpos funcionam como uma documentação viva e confiável. Quando um teste limpo falha, ele aponta com precisão onde o bug está. Por outro lado, testes "smelly" (como os fracos ou condicionais) podem esconder bugs ou quebrar por motivos errados, gerando desconfiança na equipe e aumentando o custo de manutenção.

As ferramentas de análise estática (como o ESLint) atuam como "guardiões da qualidade". Elas forçam a equipe a seguir boas práticas de forma automática, garantindo um padrão de código consistente. Elas economizam tempo de revisão de código, pois detectam problemas óbvios antes que o código chegue a outro desenvolvedor.

Em conjunto, testes limpos e análise estática criam uma "rede de segurança" que dá à equipe a confiança necessária para refatorar o código, adicionar novas funcionalidades e garantir que o software permaneça robusto e fácil de manter a longo prazo.