PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS E INFORMÁTICA
GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

LORRAYNE MARAYZE SILVA DE OLIVEIRA - 816603

REFATORAÇÃO DE TESTES E DETECÇÃO DE TEST SMELLS

TESTE DE SOFTWARE

BELO HORIZONTE
2025

ANÁLISE DE SMELLS

Analisando os testes fornecidos, aprimoramos a descrição dos Test Smells identificados, detalhando o porquê de serem considerados "maus cheiros" e os riscos associados, com base nas informações sobre qualidade de código de teste.

Um Test Smell é um sintoma no código de um teste que sugere um problema de design, manutenibilidade ou eficácia, e ignorá-lo pode levar a uma suíte de testes frágil e não confiável.

Teste Ansioso (Eager Test)

O Teste Ansioso é classificado como um Smell de Estrutura e Lógica e ocorre quando um único teste verifica vários métodos ou comportamentos diferentes. Este é o caso do teste que verifica tanto a criação (Act 1: Criar) quanto a busca de um usuário (Act 2: Buscar). O teste deve seguir o padrão "Arrange, Act, Assert" (AAA), onde o "Act" deve executar a única função ou método que está sendo testado. O risco primário deste mau cheiro é a dificuldade no diagnóstico: se o teste falhar, torna-se ambíguo qual dos dois comportamentos distintos (criação ou busca) causou o problema, comprometendo a precisão da suíte de testes.

Lógica Condicional no Teste (Conditional Logic in Test)

A Lógica Condicional no Teste, também é categorizada como um Smell de Estrutura e Lógica. É evidente este smell no teste que usa um loop (for) e uma estrutura condicional (if/else) para desativar usuários e validar dois tipos diferentes de usuários, o comum e o administrador. Os testes devem ser determinísticos e lineares, e a inclusão de if, switch, for ou while é um problema.

O risco deste smell é que o teste pode passar silenciosamente, mesmo que não tenha exercitado uma parte crucial do código: se o loop não encontrar o tipo de usuário que satisfaça uma das condições, as asserções dentro do bloco correspondente não serão executadas, o que significa que o teste não testa nada nesse cenário e pode ocultar um bug de validação. Então, a correção sugerida seria aplicar o princípio de "Um cenário, um teste".

Teste Frágil (Fragile Test)

O Teste Frágil é um Smell de Verificações (Asserts). Ele é encontrado no teste que verifica o relatório de usuários, pois ele está acoplado à implementação exata da string de formatação, em vez de focar apenas no comportamento essencial, como a presença do conteúdo importante. O problema é que o teste quebra com refatorações que não alteram o resultado funcional para o usuário. Por isso, o risco associado a este smell é a quebra desnecessária da suíte: se um desenvolvedor alterar a formatação, mudando a ordem dos campos ou adicionando um caractere sem introduzir um bug, o teste falha "inutilmente". Isso desestimula a refatoração do código de produção e diminui a confiança na suíte, pois ela gera falsos positivos de falha.

ANÁLISE MANUAL X AUTOMÁTICA

A comparação entre a análise manual e a ferramenta ESLint revela uma convergência na identificação da Lógica Condicional no Teste, onde a ferramenta confirmou essa falha reportando múltiplos erros *jest/no-conditional-expect*, validando que as asserções estavam indevidamente dentro de estruturas

condicionais. Apesar disso, o ESLint não conseguiu identificar os smells de design que exigem análise de domínio: o Teste Ansioso, que viola o princípio de um único "Act" no padrão AAA, e o Teste Frágil, que é um mau cheiro por acoplar o teste à implementação da string de formatação. Dessa forma, a ferramenta capturou problemas sintáticos e estruturais de alto risco, mas não conseguiu avaliar a semântica do design do teste, onde a determinação do que é ou não um mau cheiro.

PROCESSO DE REFATORAÇÃO

Um teste problemático e interessante de passar pelo processo de refatoração é o teste que validava a criação de um usuário menor de idade. No arquivo original, o teste apresentava o test smell de “Exception Handling Smell”, pois utilizava um bloco try/catch manual para capturar erros, o que fazia com que o teste passasse mesmo se a exceção não fosse lançada corretamente. Essa abordagem tornou o teste frágil, já que não havia uma verificação explícita de que o erro esperado realmente ocorria.

```
test('deve falhar ao criar usuário menor de idade', () => {
  // Este teste não falha se a exceção NÃO for lançada.
  // Ele só passa se o `catch` for executado. Se a lógica de validação
  // for removida, o teste passa silenciosamente, escondendo um bug.
  try {
    userService.createUser('Menor', 'menor@email.com', 17);
  } catch (e) {
    expect(e.message).toBe('O usuário deve ser maior de idade.');
  }
});
```

Figura 1. Teste Pré Refatoração

Tecnicamente, o smell foi corrigido substituindo o uso do try/catch pela asserção nativa do Jest *expect(...).toThrow()*, que valida se a função lança a exceção esperada e ainda permite verificar a mensagem do erro. O novo teste, nomeado *de* ‘Lançar erro ao tentar cadastrar um usuário menor de 18 anos’, segue a estrutura AAA (Arrange - Act - Assert) e eliminando o comportamento “falso-positivo” e tornando o teste mais alinhado às boas práticas.

```
test('Lançar erro ao tentar cadastrar um usuário menor de 18 anos', () => {
  const criarMenor = () => {
    userService.createUser('Menor de Idade', 'menor@gmail.com', 14);
  };
  expect(criarMenor).toThrow('O usuário deve ser maior de idade.');
});
```

Figura 2. Teste Pós Refatoração

RELATÓRIO DA FERRAMENTA

Na primeira execução do ESLint, como mostrado nas imagens abaixo, a ferramenta encontrou vários problemas no código de testes, totalizando nove ocorrências (sete erros e dois avisos). Entre os erros estavam o uso incorreto de require e module em um ambiente sem configuração adequada para Node.js (no-undef), além de práticas erradas em testes, como o uso de condicionais dentro de expect

(jest/no-conditional-expect) e testes sem asserções ou desativados (*jest/expect-expect e jest/no-disabled-tests).*

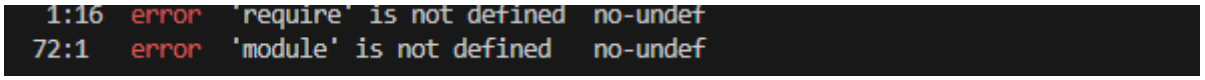

Figura 1. Erros userService.js

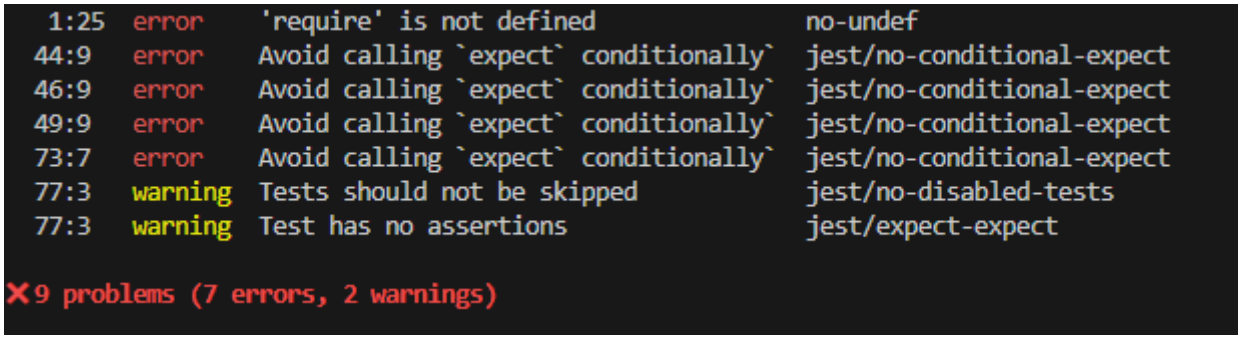

Figura 2. Erros userService.smelly.test.js

O ESLint é uma ferramenta que analisa o código de forma automática, procurando por violações de boas práticas, problemas de estilo e possíveis erros de lógica. Primeiro, ele lê a configuração, converte o código em AST, aplica regras (core + plugins) que percorrem essa AST e gera relatórios de problemas. Ele é extensível, permite correções automáticas quando seguras, e integra no editor/CI. Foi assim que ele detectou os erros nos arquivos de teste e no userService.js com a análise automática da AST contra regras configuradas (por exemplo no-undef e regras do Jest).

Com base nos erros que o ESLint apontou, é possível realizar uma refatoração mais direcionada e rápida, pois fica mais claro os itens que são necessárias correções. Essa automação economiza tempo do responsável pelos testes, porque reduz a necessidade de revisão manual de cada um dos testes e garante que todo o código siga um padrão consistente.

CONCLUSÃO

A escrita de testes limpos é importante, pois torna o código mais claro e fácil de entender, além de facilitar a identificação da causa de falhas durante a execução dos testes. Essa prática também contribui para a manutenção e evolução do sistema, pois deixa o comportamento esperado das funções bem definido e reduz ambiguidades na lógica do código. Testes bem estruturados ajudam outros desenvolvedores a compreender rapidamente o objetivo de cada cenário testado, diminuindo retrabalho e evitando erros repetidos. Além disso, manter uma boa organização nos testes garante que futuras alterações no sistema possam ser validadas com segurança, sem comprometer funcionalidades que já estavam corretas. Dessa forma, a prática de escrever testes limpos atua diretamente na qualidade, na confiabilidade e na sustentabilidade do projeto ao longo do tempo.

O uso de ferramentas de análise estática, como o ESLint, complementa esse processo ao verificar automaticamente erros, más práticas e inconsistências no código. Dessa forma, o desenvolvedor consegue corrigir os problemas de forma mais rápida e padronizada. Em conjunto, essas práticas ajudam a manter a qualidade do software, reduzem a ocorrência de erros e garantem que o projeto permaneça estável ao longo do tempo.