PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS PUC-MG

TESTE DE SOFTWARE

Nome: Paloma Dias de Carvalho – Matrícula: 587900

Professor: Cleiton Silva Tavares

Relatório – Refatoração de Testes e Detecção de Test Smells

01. Análise de Smells

Os itens abaixo apresentam os 3 *Test Smells* identificados manualmente na suíte de testes original.

01. *Eager Test* (Teste Ansioso)

Esse *smell* aparece no teste “deve criar e buscar um usuário corretamente”, onde múltiplos comportamentos são verificados dentro de um mesmo caso de teste, criação, busca e verificação de status do usuário.
Ele é considerado um “mau cheiro” porque mistura diferentes responsabilidades, o que dificulta entender qual parte do código realmente falhou quando o teste quebra.
O risco é reduzir a clareza e a confiabilidade dos testes, tornando a manutenção mais complexa e as falhas mais difíceis de diagnosticar.

```
18  test('deve criar e buscar um usuário corretamente', () => {
19    // Act 1: Criar
20    const usuarioCriado = userService.createUser(
21      dadosUsuarioPadrao.nome,
22      dadosUsuarioPadrao.email,
23      dadosUsuarioPadrao.idade
24    );
25    expect(usuarioCriado.id).toBeDefined();
26
27    // Act 2: Buscar
28    const usuarioBuscado = userService.getUserById(usuarioCriado.id);
29    expect(usuarioBuscado.nome).toBe(dadosUsuarioPadrao.nome);
30    expect(usuarioBuscado.status).toBe('ativo');
31  });
```

Imagem 1 – Eager Test

02. *Fragile Test* (Teste Frágil)

Esse *smell* aparece no teste “deve gerar um relatório de usuários formatado”, em que o resultado é comparado com uma string exata contendo espaços, quebras de linha e formatação específica.
Esse tipo de teste é considerado frágil porque ele falha com uma simples mudança estética na saída (como um espaço extra, uma quebra de linha diferente ou uma ordem alterada) pode causar uma falha, mesmo que o

comportamento do sistema continue correto.
O risco é gerar falsos negativos: o teste falha sem que o software realmente tenha um defeito funcional, o que atrapalha a evolução do código e reduz a confiança na suíte de testes.

```
54  test('deve gerar um relatório de usuários formatado', () => {
55    const usuario1 = userService.createUser('Alice', 'alice@email.com', 28);
56    userService.createUser('Bob', 'bob@email.com', 32);
57
58    const relatorio = userService.generateUserReport();
59
60    // Se a formatação mudar (ex: adicionar um espaço, mudar a ordem), o teste quebra.
61    const linhaEsperada = `ID: ${usuario1.id}, Nome: Alice, Status: ativo\n`;
62    expect(relatorio).toContain(linhaEsperada);
63    expect(relatorio.startsWith('--- Relatório de Usuários ---')).toBe(true);
64  });
```

Imagem 2 – Fragile Test

03. *Try/catch* incorreto

Esse *smell* está presente no teste “deve falhar ao criar usuário menor de idade”, que utiliza um bloco *try/catch* para verificar exceções. Nesse formato, se o método não lançar a exceção esperada, o teste passa silenciosamente, sem indicar falha.
Ele é considerado um mau cheiro porque mascara erros reais, a ausência da exceção deveria gerar uma falha, mas não gera.
O principal risco é o de falso positivo: o teste “passa” mesmo que a validação tenha sido removida ou esteja incorreta, o que pode permitir que *bugs* cheguem à produção sem serem detectados.

```
66  test('deve falhar ao criar usuário menor de idade', () => {
67    // Este teste não falha se a exceção NÃO for lançada.
68    // Ele só passa se o `catch` for executado. Se a lógica de validação
69    // for removida, o teste passa silenciosamente, escondendo um bug.
70    try {
71      userService.createUser('Menor', 'menor@email.com', 17);
72    } catch (e) {
73      expect(e.message).toBe('O usuário deve ser maior de idade.');
74    }
75  });
```

Imagem 3 – Try/catch incorreto

02. Processo de Refatoração

Optei por refatorar o teste que verifica a criação de um usuário menor de idade, pois ele apresentava um problema sério de confiabilidade. No código original, o teste utilizava um bloco *try/catch* para capturar a exceção

esperada. O grande problema é que, se nenhuma exceção fosse lançada, o teste passaria normalmente, sem indicar falha, o que poderia mascarar um erro real na lógica do sistema.

A primeira decisão foi substituir o *try/catch* pelo *matcher* nativo do *Jest (toThrow)*, que já verifica automaticamente se uma exceção é lançada. Dessa forma, caso o erro não ocorra, o teste falha como deveria, evitando falsos positivos.

Também reorganizei o código seguindo a estrutura *Arrange–Act–Assert*, deixando mais claro o que é preparação, execução e verificação. Assim, o teste ficou mais direto e fácil de entender: primeiro definimos a ação (*act*), depois verificamos se ela lança o erro esperado.

Essa refatoração tornou o teste mais seguro, legível e confiável. Agora ele reflete com precisão o comportamento que queremos garantir, ou seja, o sistema deve realmente impedir a criação de um usuário menor de idade, e o teste só passa se isso for confirmado de forma explícita.

A imagem 4 abaixo, apresenta o código antes da refatoração, bem como a imagem 5 apresenta o código refatorado.

```
66    test('deve falhar ao criar usuário menor de idade', () => {
67      // Este teste não falha se a exceção NÃO for lançada.
68      // Ele só passa se o `catch` for executado. Se a lógica de validação
69      // for removida, o teste passa silenciosamente, escondendo um bug.
70      try {
71        userService.createUser('Menor', 'menor@email.com', 17);
72      } catch (e) {
73        expect(e.message).toBe('O usuário deve ser maior de idade.');
74      }
75    });
```

Imagem 4 – Try/catch antes da refatoração

```
66  test('deve lançar erro ao criar usuário menor de idade', () => {
67    // Arrange
68    const act = () => userService.createUser('Menor', 'menor@email.com', 17);
69
70    // Act + Assert
71    expect(act).toThrow('O usuário deve ser maior de idade.');
72  });
```

Imagem 5 – Código refatorado

03. Relatório da Ferramenta

A imagem 6, apresenta o resultado da primeira execução do ESLint sobre o arquivo userService.smelly.test.js.

A ferramenta identificou automaticamente 4 erros e 2 avisos, destacando problemas relacionados ao uso incorreto de práticas de teste no Jest.

Entre os principais apontamentos estão:

- *jest/no-conditional-expect*: indica que há chamadas ao expect() dentro de estruturas condicionais (if, for, etc.), o que torna o teste dependente de lógica interna e menos confiável;
- *jest/no-disabled-tests*: alerta para a existência de testes desativados com .skip, que reduzem a cobertura da suíte;
- *jest/expect-expect*: aponta testes sem nenhuma asserção, ou seja, que não verificam efetivamente o comportamento do código.

Esses apontamentos mostram como o ESLint automatiza a detecção de *Test Smells* de forma precisa, analisando o código estático e aplicando regras predefinidas para padrões de má prática. Assim, o desenvolvedor não precisa identificar manualmente cada problema, o que aumenta a qualidade, a consistência e a segurança dos testes durante o processo de desenvolvimento.

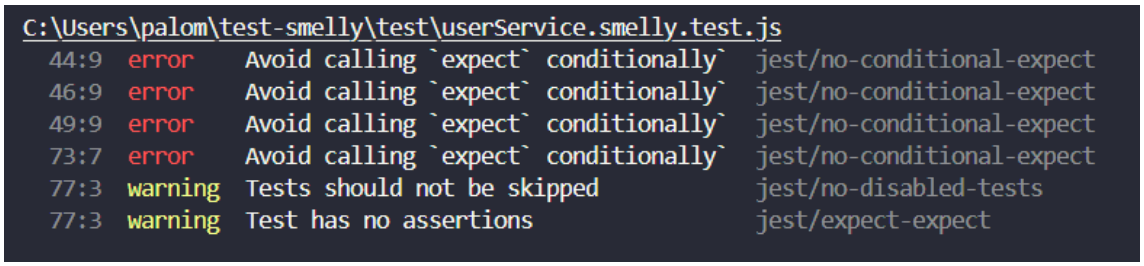

Imagem 6 – Primeira execução ESLint

04. Conclusão

Ao final desta atividade, ficou evidente que escrever testes de forma limpa e estruturada vai muito além de cumprir uma boa prática de programação, é uma forma de cuidar do código e torná-lo mais confiável ao longo do tempo. Testes bem escritos funcionam como uma rede de segurança: ajudam a detectar falhas cedo, facilitam a evolução do sistema e dão confiança para realizar mudanças sem medo de quebrar o que já funciona.

A utilização de ferramentas como o ESLint mostrou o quanto a automação pode contribuir nesse processo. Ao identificar *Test Smells* de forma rápida e precisa, apontando detalhes que poderiam passar despercebidos em uma revisão manual.

Em resumo, escrever testes limpos e utilizar ferramentas de análise estática é investir em qualidade, manutenção e sustentabilidade do projeto. São práticas que economizam tempo no futuro, evitam retrabalhos e, principalmente, ajudam a construir software de maneira mais responsável e profissional.