PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS PUC-MG

TESTE DE SOFTWARE

Nome: Paloma Dias de Carvalho – Matrícula: 587900

Professor: Cleiton Silva Tavares

Relatório – Detecção de Bad Smells e Refatoração Segura

01. Análise de Smells

01.01. Múltiplos if/else

Neste caso, pode-se observar a utilização excessiva de if/else, que acaba impactando na manutenção, testes e evolução do software. Testar cada caminho exige múltiplas combinações, assim o número de casos de teste cresce exponencialmente, aumentando o esforço do QA e a chance de erros passarem despercebidos. Além disso, quebram o princípio do Aberto/Fechado (OCP), visto que será necessário editar o código existente.

```
43        } else if (user.role === 'USER') {
44          // Users comuns só veem itens de valor baixo
45          if (item.value <= 500) {
46            if (reportType === 'CSV') {
47              report += `${item.id},${item.name},${item.value},${user.name}\n`;
48              total += item.value;
49            } else if (reportType === 'HTML') {
50              report += `<tr><td>${item.id}</td><td>${item.name}</td><td>${item.value}</td></tr>\n`;
51              total += item.value;
52            }
53          }
54        }
55      }
```

Imagem 1 – Múltiplos if/else

01.02. Método longo (Long Method)

O método generateReport (linha 11), concentra múltiplas responsabilidades, assim, possui uma complexidade de 27 (sendo o recomendado 15). Este fator viola o princípio da responsabilidade única (SRP) do SOLID, onde cada método deve ter um propósito claro. É difícil criar testes mais focados, visto que o método faz várias coisas, além disso o teste pode se tornar mais complexo que o próprio código.

01.03. Duplicação de código (Code Duplication)

O código apresenta situações de duplicação, o que aumenta desnecessariamente a quantidade de linhas de código e parecer mais complexo do que realmente é. Ao realizar a manutenção precisa se lembrar das outras partes duplicadas e se esquecer, pode ser que o sistema apresente inconsistências. A duplicação viola o princípio DRY (*Don't repeat yourself*), e prejudica a coesão e a modularidade.

```
57    // --- Seção do Rodapé ---
58    if (reportType === 'CSV') {
59      report += '\nTotal,,\n';
60      report += `${total},,\n`;
61    } else if (reportType === 'HTML') {
62      report += '</table>\n';
63      report += `<h3>Total: ${total}</h3>\n`;
64      report += '</body></html>\n';
65    }
```

Imagem 2 – Duplicação de código

02. Relatório da Ferramenta

Ao executar o comando npx eslint src/ no código original, o seguinte resultado foi obtido:

```
PS C:\Users\palom\bad-smells-js-refactoring> npx eslint src/

C:\Users\palom\bad-smells-js-refactoring\src\ReportGenerator.js
  11:3   error  Refactor this function to reduce its Cognitive Complexity from 27 to the 5 allowed  sonarjs/cognitive-complexity
  43:14  error  Merge this if statement with the nested one                                         sonarjs/no-collapsible-if

✖ 2 problems (2 errors, 0 warnings)
```

Imagem 3 – Relatório ESLint

O plugin *eslint-plugin-sonarjs* ajudou a identificar problemas estruturais no código que poderiam passar despercebidos em uma análise manual, especialmente relacionados à complexidade cognitiva e à estrutura condicional redundante.

No relatório do ESLint, foi apontado que a função principal do arquivo ReportGenerator.js apresenta complexidade cognitiva 27, enquanto o limite recomendado é 5. Esse tipo de métrica mede quantas decisões, bifurcações e caminhos lógicos o desenvolvedor precisa compreender para entender a função.

Além disso, o plugin também detectou a presença de condições aninhadas que poderiam ser fundidas (*no-collapsible-if*), indicando que há *if* redundantes ou mal estruturados.

Esses alertas automáticos mostram como ferramentas de análise estática como o *SonarJS* complementam a revisão manual, quantificando aspectos de complexidade e estrutura que, embora visíveis, não são triviais de medir sem apoio automatizado.

03. Processo de Refatoração

03.01. Bad Smell Escolhido: Complexidade Cognitiva Alta e Duplicação de Código

Optou-se por abordar o mau cheiro de Complexidade Cognitiva Alta e Método Longo, uma vez que esse problema se mostrou o mais crítico, pois agrupa e potencializa outros aspectos negativos, como duplicação de código e condicionais aninhadas.

A refatoração teve como foco reduzir a complexidade e centralizar a lógica comum, aplicando boas práticas de coesão e responsabilidade única.

```
export class ReportGenerator {
  constructor(database) {
    this.db = database;
  }
  generateReport(reportType, user, items) {
    let report = '';
    let total = 0;

    // --- Seção do Cabeçalho ---
    if (reportType === 'CSV') {
      report += 'ID,NOME,VALOR,USUARIO\n';
    } else if (reportType === 'HTML') {
      report += '<html><body>\n';
      report += '<h1>Relatório</h1>\n';
      report += `<h2>Usuário: ${user.name}</h2>\n`;
      report += '<table>\n';
      report += '<tr><th>ID</th><th>Nome</th><th>Valor</th></tr>\n';
    }

    // --- Seção do Corpo (Alta Complexidade) ---
    for (const item of items) {
      if (user.role === 'ADMIN') {
        // Admins veem todos os itens
        if (item.value > 1000) {
          // Lógica bônus para admins
          item.priority = true;
        }

        if (reportType === 'CSV') {
          report += `${item.id},${item.name},${item.value},${user.name}\n`;
          total += item.value;
        } else if (reportType === 'HTML') {
          const style = item.priority ? ' style="font-weight:bold;"' : '';
          report += `<tr${style}><td>${item.id}</td><td>${item.name}</td><td>${item.value}</td></tr>\n`;
          total += item.value;
        }
      } else if (user.role === 'USER') {
        // Users comuns só veem itens de valor baixo
        if (item.value <= 500) {
          if (reportType === 'CSV') {
            report += `${item.id},${item.name},${item.value},${user.name}\n`;
            total += item.value;
          } else if (reportType === 'HTML') {
            report += `<tr><td>${item.id}</td><td>${item.name}</td><td>${item.value}</td></tr>\n`;
            total += item.value;
          }
        }
      }
    }

    // --- Seção do Rodapé ---
    if (reportType === 'CSV') {
      report += '\nTotal,,\n';
      report += `${total},,\n`;
    } else if (reportType === 'HTML') {
      report += '</table>\n';
      report += `<h3>Total: ${total}</h3>\n`;
      report += '</body></html>\n';
    }

    return report.trim();
  }
}
```

Imagem 4 – Código antes da refatoração

03.02. Técnicas de Refatoração Aplicadas

03.02.01. Extract Method: Dividir um método longo em funções menores, coesas e com propósito bem definido.

Aplicação:

- Criação do método generateHeader() para compor o cabeçalho do relatório;

- Criação do método generateBody() para iterar e processar os itens filtrados conforme o papel do usuário;
- Criação do método generateFooter() para gerar o rodapé com o total;
- Criação do método formatItem() para formatar cada linha de item conforme o tipo de relatório.

Essas extrações reduziram a quantidade de condicionais dentro de um mesmo escopo e deixaram o código mais legível e modular.

03.02.02. Replace Conditional with Polymorphism: Substituir múltiplas condicionais por polimorfismo, delegando o comportamento para subclasses específicas.

Aplicação:

- Implementação de uma classe base Generator, responsável por definir a estrutura comum do relatório (template base);
- Criação de duas subclasses especializadas: CsvGenerator e HtmlGenerator, que estendem Generator e implementam suas próprias versões de formatação;
- O ReportGenerator atua como um fábrica simples, escolhendo dinamicamente qual gerador usar com base no tipo de relatório solicitado (CSV ou HTML).

Essa abordagem elimina a necessidade de longos blocos if/else, tornando o código mais flexível e extensível.

03.03. Problemas identificados

- Complexidade Cognitiva: **27** (limite ideal: **5**);
- Condicionais aninhadas e colapsáveis;
- Duplicação de lógica entre os formatos **CSV** e **HTML**;
- Um único método concentrava várias responsabilidades (violando o SRP — *Single Responsibility Principle*).

```
84    generateReport(reportType, user, items) {
85      const Impl = reportType === "CSV" ? CsvReport : HtmlReport;
86      return new Impl(user, items).build();
87    }
```

Imagem 5 – Código refatorado

```
PS C:\Users\palom\bad-smells-js-refactoring> npm test

> bad-smells-js-refactoring@1.0.0 test
> node --experimental-vm-modules node_modules/jest/bin/jest.js

(node:9224) ExperimentalWarning: VM Modules is an experimental feature and might change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
 PASS  tests/ReportGenerator.refactored.test.js
 PASS  tests/ReportGenerator.test.js

Test Suites: 2 passed, 2 total
Tests:       10 passed, 10 total
Snapshots:   0 total
Time:        0.901 s, estimated 1 s
Ran all test suites.
```

Imagem 6 – npm test após a refatoração

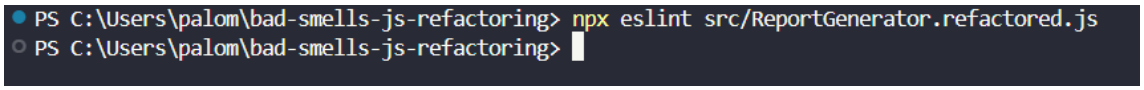

Imagem 7 – Execução do ESLint após a refatoração

03.04 Melhorias alcançadas

- Redução significativa da complexidade cognitiva: Cada método passou a ter uma responsabilidade única, facilitando a leitura e os testes unitários.
- Eliminação da duplicação: A lógica comum foi movida para a classe base, enquanto as subclasses cuidam apenas das diferenças de formato.
- Simplificação de condicionais: O uso de polimorfismo eliminou a necessidade de if/else baseados no tipo de relatório.
- Maior coesão e clareza: Cada classe é responsável por uma única etapa do processo de geração de relatórios.
- Facilidade de extensão: Novos formatos (como PDF ou JSON) podem ser adicionados apenas criando uma nova subclasse, sem alterar o código existente.

04. Conclusão

A realização deste trabalho permitiu compreender, na prática, como técnicas de refatoração podem transformar um código complexo e de difícil manutenção em uma estrutura mais limpa, modular e sustentável.
A análise feita com o ESLint e o plugin SonarJS foi fundamental para identificar problemas que muitas vezes passam despercebidos em uma revisão manual, como a complexidade cognitiva elevada e as condicionais aninhadas desnecessárias.

Com a aplicação dos padrões Extract Method e Replace Conditional with Polymorphism, o código tornou-se mais legível, coeso e preparado para futuras extensões, eliminando duplicações e reduzindo o esforço mental exigido para sua compreensão.
Além da melhoria técnica, o processo reforçou a importância de aplicar princípios como responsabilidade única (SRP), baixo acoplamento e abertura para extensão sem modificação (OCP), que são essenciais para a qualidade de software.

Em resumo, a refatoração não apenas atendeu aos alertas da análise estática, mas também promoveu um código mais fácil de testar, evoluir e manter, demonstrando como boas práticas e ferramentas de apoio são aliadas valiosas na construção de sistemas mais confiáveis e duradouros.