PONTÍFICIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Engenharia de Software – Teste de Software

Aluno: Gabriel Lourenço Reis Resende

Professor: Cleiton Silva Tavares

Detecção de Bad Smells e Refatoração Segura

Belo Horizonte

2025

1. Introdução

O objetivo deste relatório é documentar a análise e refatoração da classe *ReportGenerator.js*, responsável por gerar relatórios em diferentes formatos (CSV e HTML) com base no perfil do usuário. O código original apresentava sinais claros de *Bad Smells,* como métodos longos, complexidade condicional elevada e duplicação de lógica, dificultando a manutenção, testes e extensibilidade. Durante o processo de refatoração, aplicamos boas práticas de *Clean Code*, técnicas como *Extract Method*, *Polymorphism* (*Strategy Pattern)* e eliminação de números mágicos, visando um código mais legível, modular e testável. Nesse relatório, detalhamos a análise manual, os resultados de ferramentas automáticas, o processo de refatoração e os impactos da melhoria de qualidade do código.

Repositório do projeto: https://github.com/gabrielreisresende/bad-smells-js-refactoring

2. Análise de *Smells*

2.1 *Long Method*

O *Bad Smell Long Method* ocorreu no método generateReport tinha mais de 80 linhas, acumulando tarefas diferentes: cabeçalho, corpo, rodapé, formatação HTML/CSV, regras de acesso por usuário. Dessa forma, métodos longos dificultam a leitura, manutenção e teste unitário, pois há muitas responsabilidades misturadas.

2.2 *Conditional Complexity*

O *Bad Smell Conditional Complexity* exitia na lógica de controle tinha condições aninhadas para diferenciar tipo de usuário (ADMIN/USER) e tipo de relatório (CSV/HTML), além de verificações de valor de item. Sendo assim, isso aumenta a complexidade cognitiva, tornando difícil prever todas as ramificações e aumentando o risco de bugs ao adicionar novas regras.

2.3 *Magic Numbers*

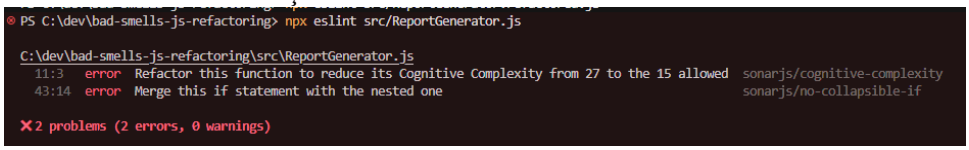
O *Bad Smell Magic Numbers* estava presente em valores como 500 e 1000 estavam hardcoded no método. Com isso, a presença desses valores reduz a legibilidade e dificulta ajustes futuros. Por exemplo, se o limite de visualização do usuário mudar, seria preciso procurar manualmente todos os usos do número 500.

3. Relatório da Ferramenta

Nesse capítulo será apresentado um relatório da ferramenta *ESLint* junto com o *eslint-plugin-sonarjs plugin* para detectar Bad Smells no código de geração de um relatório.

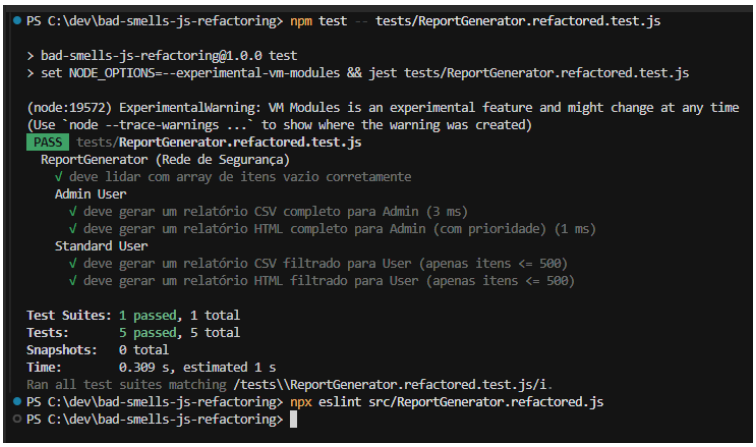
3.1 Cenário Anterior à Refatoração

O *ESLint* identificou a alta complexidade cognitiva (*cognitive-complexity*) e *ifs* aninhados (*no-collapsible-if*). Com isso, essas detecções confirmam os *Bad Smells* manuais, mostrando que o *linter* ajuda a capturar problemas estruturais que podem passar despercebidos, como complexidade oculta e ramificações aninhadas.



3.2 Cenário Refatorado

Após a refatoração, ao executar o linter, não foi detctado nenhum Bad Smell existente no código, e todos os testes unitários foram executados com sucesso.



4. Processo de Refatoração

4.1 Cenário Anterior

O método *generateReport* estava com diversos problemas como: muitas condições aninhdas, duplicação de lógica de formatação e misturada de responsibilidades (regras de negócio + formatação).

Trecho crítico anterior do método:

```
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;
      }
    }
  }
}
```

4.2 Cenário Após Refatoração:

Para realizar a refatoração do código foram aplicadas técnicas como:

Extract Method: Separação de cabeçalho, corpo e rodapé em métodos específicos (*addHeader, addBody, addFooter*).

Polymorphism/Strategy Pattern: Criadas classes separadas para *AdminReportStrategy* e *UserReportStrategy*, eliminando os *if/else* aninhados e facilitando a extensão futura.

Eliminação de duplicação: Métodos auxiliares *addCSVLine* e *addHTMLRow* na classe base, evitando repetir código em cada estratégia.

Dessa forma, o *Smell* mais crítico corrigido foi o *Long Method + Conditional Complexity.*

Trecho crítico após a refatorção do método:

```
class AdminReportStrategy extends BaseReportStrategy {
  addHeader() {···
  }

  addBody(items) {
    for (const item of items) {
      const isPriority = item.value > ADMIN_PRIORITY_LIMIT;
      if (this.reportType === 'CSV') {
        this.addCSVLine(item);
      } else if (this.reportType === 'HTML') {
        const style = isPriority ? 'style="font-weight:bold;"' : '';
        this.addHTMLRow(item, style);
      }
    }
  }
class UserReportStrategy extends BaseReportStrategy {
  addHeader() {···
  }

  addBody(items) {
    const filteredItems = items.filter((item) => item.value <= USER_VIEW_LIMIT);
    for (const item of filteredItems) {
      if (this.reportType === 'CSV') {
        this.addCSVLine(item);
      } else if (this.reportType === 'HTML') {
        this.addHTMLRow(item);
      }
    }
  }
```

5. Conclusão

A refatoração da classe ReportGenerator resultou em um código mais limpo, modular e fácil de manter, reduzindo a complexidade cognitiva, eliminando duplicação e números mágicos, e separando claramente as responsabilidades de formatação e regras de negócio por meio do uso de estratégias (*Strategy Pattern*); essa melhoria não apenas facilita a adição de novos tipos de relatórios ou perfis de usuários, como também aumenta a testabilidade do código, reforçando a importância de ferramentas como *ESLint* e *sonarjs* para detectar problemas estruturais que podem passar despercebidos manualmente e garantindo maior confiabilidade, legibilidade e extensibilidade do *software*.