TESTES DE *SOFTWARE*

PAULA DE FREITAS CAMARGOS
Mª: 788521

Detecção de *Bad Smells* e Refatoração Segura

Belo Horizonte

2025

1. Análise de *smells*

Os *code smells* são sinais de alerta que indicam potenciais problemas no código, sugerindo que algo pode estar mal estruturado ou mal projetado. Eles não são erros ou bugs propriamente ditos já que o *software* funciona, mas representam fragilidades no *design* que, se ignoradas, podem dificultar a evolução do sistema, aumentar o risco de falhas e comprometer a manutenibilidade do código ao longo do tempo.

Esses maus cheiros surgem, em geral, devido a más práticas de programação ou a um crescimento desorganizado do código, e funcionam como indicadores de que o projeto precisa de refatoração. Exemplos comuns incluem classes excessivamente grandes ou métodos longos e sobrecarregados, nos quais uma única função tenta realizar múltiplas tarefas, tornando o entendimento e a manutenção mais difíceis.

Outro caso recorrente é o código duplicado, quando o mesmo trecho aparece repetido em diferentes partes do sistema, aumentando o esforço para aplicar correções e melhorias. Problemas como esses contribuem para um código confuso, complexo e propenso a erros, elevando a dívida técnica e reduzindo a produtividade da equipe. Identificar e tratar *code smells* é essencial para preservar a qualidade, legibilidade e sustentabilidade do software a longo prazo.

Durante a análise manual do código original, foi possível identificar três tipos distintos de *bad smells* que comprometem a qualidade e manutenibilidade do software. O primeiro *smell* identificado foi o Método Longo, onde a função *generateReport* concentrava toda a lógica de geração de relatórios em um único método com mais de cinquenta linhas de código.

Este método realizava múltiplas responsabilidades simultaneamente, incluindo a filtragem de itens baseada no papel do usuário, o processamento de dados, a formatação de cabeçalhos, corpos e rodapés para diferentes tipos de relatórios, e o cálculo de totais. A concentração de tantas responsabilidades em um único método torna o código difícil de entender e modificar, já que qualquer alteração em uma parte específica da lógica pode afetar outras partes do mesmo método.

O segundo *bad smell*identificado foi a Complexidade Condicional Excessiva, caracterizada por múltiplos níveis de aninhamento de estruturas condicionais *if-else* dentro do método principal. Como ilustrado na figura 1, o código apresenta uma estrutura profundamente aninhada que verificava primeiro o papel do usuário (ADMIN ou USER), depois o tipo de relatório (CSV ou HTML), e ainda continha condições adicionais para verificar valores de itens e prioridades.

Esta complexidade condicional aumenta significativamente a complexidade cognitiva do código, tornando-o difícil de seguir mentalmente e aumentando a probabilidade de introdução de erros durante manutenções futuras.

O terceiro *bad smell*, também ilustrado na figura 1, foi a duplicação de código, onde padrões similares de formatação e processamento eram repetidos em diferentes partes do método. Especificamente, a lógica para gerar linhas de relatório CSV e HTML era duplicada tanto para usuários ADMIN quanto para usuários USER, com apenas pequenas variações. Esta duplicação viola o princípio DRY (*don't repeat yourself*) e cria pontos de falha múltiplos, onde uma correção em uma parte do código pode não ser aplicada em outras partes similares, levando a inconsistências e bugs difíceis de rastrear.

Figura 1. Estrutura *if-else* aninhada e código duplicado.

```
    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;
      }
    }
```

2. Relatório da ferramenta

A execução do *ESLint* com o plugin *eslint-plugin-sonarjs* revelou problemas críticos que complementaram e validaram a análise manual realizada anteriormente. A ferramenta detectou que a função *generateReport* possuía uma Complexidade Cognitiva de 27, excedendo significativamente o limite configurado de 15 pontos. A complexidade cognitiva é uma métrica que mede a dificuldade de compreensão de um trecho de código, considerando fatores como aninhamento de estruturas condicionais, loops e operadores lógicos. Um valor elevado indica que o código é mentalmente difícil de processar, aumentando o risco de erros durante desenvolvimento e manutenção.

O *ESLint* também identificou um problema específico de *If* Colapsável na linha 43, onde uma estrutura condicional poderia ser simplificada através da combinação de condições aninhadas. Esta detecção automática demonstrou a importância de ferramentas de análise estática, pois problemas sutis como este podem passar despercebidos durante revisões manuais, especialmente em códigos com múltiplos níveis de aninhamento.

Figura 2. Relatório do *ESLint*.

```
Z:\ws-PUC\sotf-test\bad-smells-js-refactoring\src\ReportGenerator.js
  11:3   error  Refactor this function to reduce its Cognitive Complexity from 27 to the 15 allowed  sonarjs/cognitive-complexity
  43:14  error  Merge this if statement with the nested one                                          sonarjs/no-collapsible-if
```

O *ESLint* foi fundamental para quantificar objetivamente a complexidade do código, fornecendo métricas precisas que validaram as observações qualitativas da análise manual. Enquanto a análise manual identificou problemas estruturais e de design, a análise estática forneceu evidências quantitativas que justificaram a necessidade de refatoração, demonstrando que a complexidade cognitiva estava quase o dobro do limite recomendado.

3. Processo de refatoração

O *bad smell* mais crítico identificado foi o de Complexidade Cognitiva Excessiva, diretamente associado aos padrões de Método Longo e Complexidade Condicional. Esse problema é evidenciado na figura 3, que ilustra um método com alto nível de aninhamento, muitas linhas de código e diversas lógicas distintas, cada uma relacionada a funções diferentes.

```
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();
}
```

Para abordar este problema, foram aplicadas múltiplas técnicas de refatoração de forma sistemática e incremental. A técnica principal aplicada foi o *Extract Method*, onde o método monolítico *generateReport* foi decomposto em métodos menores e mais focados, cada um com uma responsabilidade única e bem definida. Especificamente, foram extraídos os métodos *filterItemsByUserRole*, responsável por filtrar itens baseado no papel do usuário, processItemsForUser, que adiciona informações específicas como prioridade aos itens, *generateHeader*, *generateBody* e *generateFooter*, que geram as diferentes seções do relatório, e *formatItem*, que formata itens individuais.

Figura 4. Exemplificação do método *generateHeader*.

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

A técnica de *decompose conditional* foi aplicada para simplificar as estruturas condicionais complexas, extraindo a lógica de decisão para métodos específicos que retornam valores ou objetos processados, eliminando assim os múltiplos níveis de aninhamento. Por exemplo, a lógica de filtragem que estava embutida dentro de loops e condicionais foi extraída para o método *filterItemsByUserRole*, que utiliza uma abordagem funcional com o método filter do próprio JavaScript.

Para eliminar a duplicação de código, foi aplicada a técnica de Extract Method combinada com a criação de métodos auxiliares reutilizáveis. A lógica de formatação de itens, que estava duplicada para diferentes tipos de usuários e formatos de relatório, foi consolidada no método *formatItem*, que recebe o tipo de relatório, usuário e item como parâmetros, eliminando completamente a duplicação. Além disso, foram introduzidas constantes nomeadas (*MAX_USER_ITEM_VALUE* e *PRIORITY_THRESHOLD*) para substituir números mágicos que estavam espalhados pelo código.

4. Conclusão

Os testes automatizados atuam como uma verdadeira rede de segurança durante a refatoração, garantindo que as alterações no código não introduzam novos erros nem comprometam funcionalidades já existentes. Com uma boa cobertura de testes, é possível refatorar com confiança, focando na melhoria do design e na eliminação de problemas estruturais sem medo de quebrar o sistema.

A redução de *bad smells* contribui diretamente para a melhoria da qualidade geral do software, tornando o código mais limpo, legível e fácil de manter. Além de diminuir a complexidade e o risco de falhas, um código livre de smells favorece a colaboração entre desenvolvedores, acelera o desenvolvimento de novas funcionalidades e aumenta a longevidade do sistema. Em conjunto, testes bem estruturados e refatorações orientadas à qualidade formam a base para um software mais estável, que seja sustentável e que facilite e permita a evolução do *software*.