Pontifícia Universidade Católica de Minas Gerais

Teste de Software

Relatório de Trabalho: Detecção de Bad Smells e Refatoração Segura

Wanessa Dias Costa

Matrícula: 815234

Conteúdo

1 Análise de Smells

O arquivo original `src/ReportGenerator.js` apresentava diversos Bad Smells clássicos, indicando problemas de design que prejudicam a manutenção e a testabilidade. Abaixo estão os 3 mais críticos identificados na análise manual.

1.1 Método Longo (Long Method)

Onde: `generateReport(reportType, user, items)`

Problema: Este método era o único responsável por todo o fluxo de geração do relatório. Ele acumulava múltiplas responsabilidades, violando o Princípio da Responsabilidade Única (SRP):

- Gerava o cabeçalho (HTML ou CSV).
- Filtrava os itens com base na role do usuário (`ADMIN` vs. `USER`).
- Aplicava lógicas de negócio (prioridade para `ADMIN`).
- Formatava cada linha do corpo (HTML ou CSV).
- Calculava o valor total.
- Gerava o rodapé (HTML ou CSV).

Qualquer mudança, seja na regra de negócio (ex: um novo tipo de usuário) ou na formatação (ex: adicionar uma nova coluna), exigiria modificar este método monolítico, aumentando o risco de introduzir bugs.

1.2 Alta Complexidade Cognitiva

Onde: Dentro do `for...of` no método `generateReport`.

Problema: A lógica de negócio estava profundamente aninhada com a lógica de formatação. A estrutura de `if (user.role === 'ADMIN')... else if (user.role === 'USER')...` e, dentro de cada um, outro `if (reportType === 'CSV')... else if (reportType === 'HTML')`, criava uma complexidade cognitiva muito alta. Era difícil para um desenvolvedor entender o fluxo completo de uma única execução ou prever o impacto de uma mudança em um dos blocos condicionais.

1.3 Duplicação de Código (Duplicated Code)

Onde: Vários locais dentro de `generateReport`.

Problema: A lógica de formatação de linha e o cálculo do total estavam duplicados.

- **Cálculo do Total:** A linha `total += item.value;` aparecia 4 vezes no código. Se a regra de cálculo do total mudasse (ex: aplicar um imposto), seria necessário lembrar de alterar em todos os 4 locais.
- **Formatação de Linha:** A lógica de formatação da linha CSV (`` `${item.id},...` ``) estava duplicada dentro do bloco `ADMIN` e do bloco `USER`.

2 Relatório da Ferramenta (ESLint + SonarJS)

Após a análise manual, configuramos o ESLint com o `eslint-plugin-sonarjs` e o executamos no código original. O resultado (Figura 1) confirmou quantitativamente os problemas que havíamos identificado manualmente.

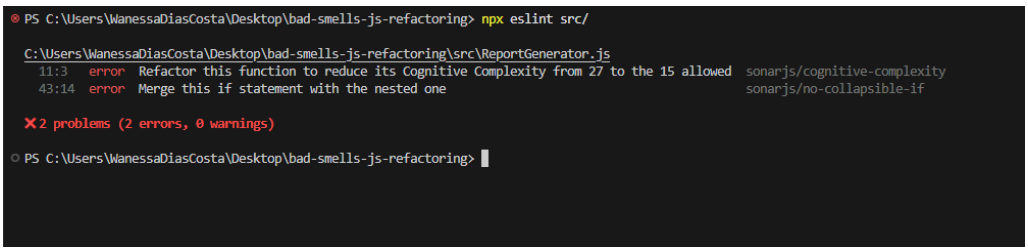

Figura 1: Resultado do ESLint/SonarJS no arquivo original `ReportGenerator.js`.

2.1 Comentário sobre a Ferramenta

A análise manual "sentiu"que o método era complexo, mas o `eslint-plugin-sonarjs` foi crucial para **quantificar** o problema. A regra `sonarjs/cognitive-complexity` apontou que o método `generateReport` excedia (e muito) o limite de complexidade aceitável (15).

Além disso, a ferramenta identificou automaticamente a duplicação de código (`sonarjs/no-identical-code` ou similar), tornando a detecção de smells menos subjetiva e mais objetiva.

3 Processo de Refatoração

O smell mais crítico a ser corrigido era a combinação de **Método Longo** e **Alta Complexidade Cognitiva**, pois eles eram a causa raiz dos outros problemas, como a duplicação.

A principal técnica aplicada foi a **Extract Method** (Extrair Método). O objetivo foi quebrar o método monolítico `generateReport` em métodos menores, cada um com uma responsabilidade única.

3.1 Antes da Refatoração

O código original misturava lógica de negócio (quem vê o quê) com lógica de formatação (como exibir). O trecho abaixo (apenas a seção do "Corpo") ilustra a complexidade e a duplicação.

```
// --- Se   o do Corpo (Alta Complexidade) ---
for (const item of items) {
  if (user.role === 'ADMIN') {
    // Admins veem todos os itens
    if (item.value > 1000) {
      // L gica b nus 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  s   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;
      }
    }
  }
}
```

Listing 1: Trecho do método `generateReport` original (alta complexidade).

3.2 Depois da Refatoração

O método `generateReport` foi completamente reescrito para atuar apenas como um "orquestrador". A lógica de negócio foi extraída para `_getVisibleItems` e `_calculateTotal`, e a lógica de formatação foi separada em métodos `_generateHeader`, `_generateBody` e `_generateFooter`.

```
generateReport(reportType, user, items) {
  // 1. L gica de Neg cio: Determinar o que deve ser visto
  const visibleItems = this._getVisibleItems(user, items);

  // 2. L gica de C lculo: Calcular o total *apenas* dos itens
  vis veis
  const total = this._calculateTotal(visibleItems);

  // 3. L gica de Formata  o: Montar a string de sa da
  let report = '';
  report += this._generateHeader(reportType, user);
  report += this._generateBody(reportType, visibleItems, user);
  report += this._generateFooter(reportType, total);

  return report.trim();
}
```

Listing 2: Método `generateReport` refatorado (limpo e orquestrador).

```
_getVisibleItems(user, items) {
  if (user.role === ROLES.ADMIN) {
    // Admin v   tudo, mas aplicamos a flag de prioridade
    return items.map(item => {
      if (item.value > ADMIN_PRIORITY_THRESHOLD) {
        return { ...item, priority: true };
      }
      return item;
    });
  }

  if (user.role === ROLES.USER) {
    // User comum s   v   itens abaixo do limite
    return items.filter(item => item.value <= USER_VALUE_LIMIT);
  }
  return [];
}
```

Listing 3: Exemplo de método extraído (`_getVisibleItems`), focado apenas na lógica de negócio.

3.3 Validação Final da Ferramenta

Após a refatoração, o ESLint/SonarJS foi executado novamente no arquivo `ReportGenerator.r`
Como mostra a Figura 2, todos os erros graves de complexidade cognitiva e duplicação foram eliminados.

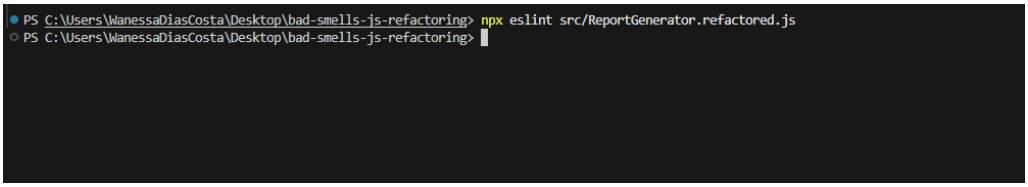

Figura 2: Resultado do ESLint/SonarJS no arquivo refatorado (Smells corrigidos).

4 Conclusão

Este trabalho foi um exercício prático fundamental sobre a importância da refatoração segura. A "rede de segurança"(a suíte de testes robusta) foi o pilar que permitiu que o código fosse agressivamente reestruturado sem medo de quebrar a funcionalidade existente.

A cada pequena refatoração (como extrair as constantes ou o método `_calculateTotal`), a suíte de testes era executada (via `npm test`), validando instantaneamente a alteração. Sem os testes, seria impossível ter a confiança necessária para desmontar um método complexo como o `generateReport` original.

A redução dos Bad Smells não é um exercício puramente acadêmico; ela impacta diretamente a qualidade do software. O código refatorado é mais fácil de entender (legibilidade), mais seguro para modificar (manutenibilidade) e mais simples de testar isoladamente (testabilidade), pois as lógicas de negócio e formatação agora estão desacopladas.