PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS E INFORMÁTICA
GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

LORRAYNE MARAYZE SILVA DE OLIVEIRA - 816603

DETECÇÃO DE BAD SMELLS E REFATORAÇÃO SEGURA

TESTE DE SOFTWARE

BELO HORIZONTE
2025

PRÉ-ANÁLISE DE SMELLS

Durante a execução dos testes com o comando *'npm test'*, foram identificadas falhas que exigiram correções prévias no código, embora as orientações previssem que os testes funcionariam sem modificações. As falhas observadas foram: erro de importação, falha no teste de geração de relatório HTML completo para o perfil de administrador.

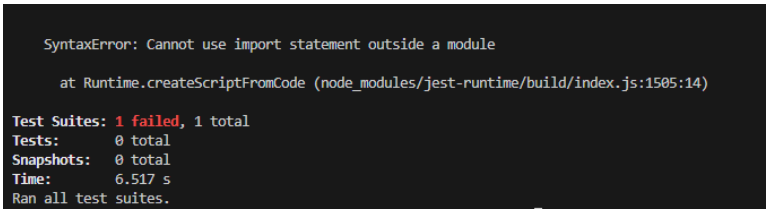


Figura 1. Erro no import

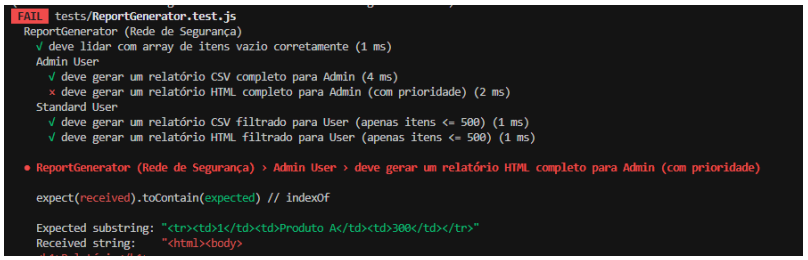


Figura 2. Falha na execução de teste

A correção foi aplicada no método *'generateReport'*, da classe **ReportGenerator.js**, para resolver a falha que indicava a má formatação de strings no HTML. A falha ocorreu devido à inserção de um espaço dentro da tag HTML quando a linha não possuía o atributo de prioridade. A solução visou o ajuste a forma de concatenação do atributo de estilo, mantendo a variável `style` com o valor do atributo ou uma string vazia e criando a variável `styleAttribute`, que recebe um espaço seguido do estilo apenas quando há algum conteúdo.

ANÁLISE DE SMELLS

Durante a análise manual, foram identificados e descritos os Bad Smells, explicando por que são considerados "maus cheiros" e quais riscos representam para a qualidade do código. Um Bad Smell é um indício de que o código pode estar mal estruturado ou difícil de manter, mesmo que ainda funcione corretamente.

1. **Números Mágicos**
   O código usa valores numéricos que têm significado de negócio, mas não estão nomeados de forma clara. Um exemplo é:
   - *if (item.value > 1000)*

   Ao ver esse número no código, não fica evidente o que ele representa, o que prejudica a legibilidade e dificulta as futuras alterações. Idealmente esses números deveriam ser substituídos por constantes nomeadas, para que as regras de negócio fiquem mais claras..

2. **Método Longo**
   O método *'generateReport'* possui cerca de 50 linhas e realiza três tarefas diferentes: geração do cabeçalho, filtragem do conteúdo e criação do rodapé. Esse acúmulo de responsabilidades torna o código mais difícil de entender, testar e modificar.

Dessa forma, seria ideal dividir o método em funções menores, cada uma com uma responsabilidade específica, aplicando a técnica de Extract Method.

3. **Estruturas de Controle Excessivas**
   O código usa muito blocos de if/else aninhados para lidar com o tipo de relatório *(reportType*) e o tipo de usuário (*user.role*).

   Além disso, há repetições como:
   - if (reportType === 'CSV')
   - else if (reportType === 'HTML')

   Essas condições aparecem dentro de loops, o que aumenta a complexidade e dificulta a manutenção. Esse padrão também viola o Princípio Aberto/Fechado (OCP), já que, para adicionar um novo tipo de relatório, seria necessário modificar o código existente.

ANÁLISE MANUAL X AUTOMÁTICA

Com o uso do *eslint-plugin-sonarjs*, foi possível confirmar alguns dos Bad Smells identificados na análise manual. Além disso, a ferramenta mostrou que o método tinha Complexidade Cognitiva igual a 31, enquanto o limite recomendado é 5. Esse número indica que o código está difícil de manter e testar. Porém, somente com a análise manual não seria possível medir o nível de complexidade de uma forma precisa, padronizada e que permite o processo de repetibilidade.

```
 11:3   error  Refactor this function to reduce its Cognitive Complexity from 31 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)
```

Figura 3. Resultado eslint Antes da Refatoração

PROCESSO DE REFATORAÇÃO

O principal problema encontrado foi o das Estruturas de Controle Excessivas (Many If-Else), esse Bad Smell prejudica o Princípio Aberto/Fechado, que diz que o código deve ser aberto para extensão, mas fechado para modificação. No código original, o método violava esse princípio porque repetia várias vezes as checagens do tipo de relatório dentro do loop principal, e essa repetição aumentava a complexidade do código, tornando o método difícil de entender e manter.

```
for (const item of items) {
  if (user.role === 'ADMIN') {
    if (item.value > 1000) {
      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;"' : '';
      const styleAttribute = style ? ` ${style}` : '';

      report += `<tr${styleAttribute}><td>${item.id}</td><td>${item.name}</td><td>${item.value}</td></t
      total += item.value;
    }
  } else if (user.role === 'USER') {
    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;
      }
```

Figura 4. Código Antes da Refatoração

Depois da Refatoração

A formatação do relatório foi extraída para um método auxiliar, o método principal (_processSingleItem) se preocupa apenas em verificar permissões e regras de negócio, enquanto o novo método (_generateReportLine) cuida exclusivamente de definir como o item deve ser formatado (CSV, HTML, etc.). Essa separação não elimina todos os if/else, mas deixa isolados, reduzindo a complexidade dos métodos principais e facilitando futuras alterações.

A técnica de refatoração usada foi o Extract Method, decompondo as condicionais, movendo a lógica de formatação (os if/else do tipo de relatório) para um novo método separado.

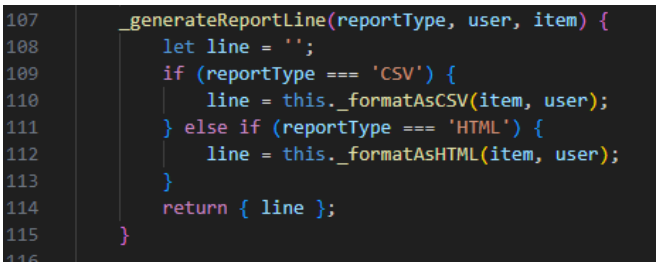
```
_processSingleItem(reportType, user, item) {
    let reportLine = null;
    let itemValue = 0;

    if (user.role === 'ADMIN') {
        this._handleAdminLogic(item);
        const lineData = this._generateReportLine(reportType, user, item);
        reportLine = lineData.line;
        itemValue = item.value;

    } else if (user.role === 'USER' && item.value <= 500) {
        const lineData = this._generateReportLine(reportType, user, item);
        reportLine = lineData.line;
        itemValue = item.value;
    }

    if (reportLine) {
        return { reportLine, itemValue };
    }
    return null; // Item não incluído
}
```

Figura 5. Código Pós Refatoração (Método 1)



```
    _generateReportLine(reportType, user, item) {
        let line = '';
        if (reportType === 'CSV') {
            line = this._formatAsCSV(item, user);
        } else if (reportType === 'HTML') {
            line = this._formatAsHTML(item, user);
        }
        return { line };
    }
```

Figura 6. Código Pós refatoração (Método 2)

CONCLUSÃO

Os testes existentes foram importantes para garantir que a refatoração não alterasse o comportamento do sistema. Durante a divisão do método *'generateReport'* em funções menores, como _generateHeader e _processSingleItem, os testes permitiram verificar se tudo continuava funcionando de forma automática, em vez de revisar manualmente cada caso, bastava rodar o Jest e corrigir os pontos que falhavam. Isso deu segurança para reorganizar o código sem comprometer as funcionalidades originais.

A refatoração reduziu os Bad Smells (o linter não apontou mais nenhum erro grave de smell), essa redução de smells deixou o código ficou mais legível, modular e fácil de manter.