PUC Minas

TESTE DE SOFTWARE

Detecção e Refatoração de Bad Smells

Larissa Pedrosa Silva

801161

Belo Horizonte

2025

1. *Análise de Smells*

a. *Método Longo*

O método *generateReport()* possui 57 linhas e concentra múltiplas responsabilidades, como a geração de cabeçalho, o processamento de itens, a formatação de dados, o cálculo de totais e a criação do rodapé. Essa estrutura viola o princípio da responsabilidade única. Isso é problemático pois compromete a manutenibilidade, uma vez que qualquer alteração em uma parte do relatório exige modificar um método extenso, aumentando o risco de introduzir bugs. Além disso, afeta a testabilidade, já que testar cada funcionalidade isoladamente se torna difícil devido ao forte acoplamento entre as partes. Além disso, prejudica a legibilidade, pois compreender o fluxo completo requer a leitura de todo o método, o que dificulta a navegação e o entendimento do código.

b. *Duplicação de Código*

O código apresenta duplicação na lógica de formatação de itens, repetindo a mesma estrutura de condicionais — como `if (reportType === 'CSV')` e `if (reportType === 'HTML')` — em diferentes partes do código. Essa repetição ocorre no processamento de itens para ADMIN (linhas 35 a 42), no processamento de itens para USER (linhas 46 a 52), na geração de cabeçalho (linhas 16 a 24) e na geração de rodapé (linhas 58 a 65). Esse problema afeta diretamente a manutenção, pois qualquer alteração na lógica de formatação precisa ser feita em vários pontos, aumentando o risco de inconsistências. Também prejudica os testes, já que cada duplicação precisa ser verificada separadamente, ampliando a superfície de testes. Além disso, eleva a probabilidade de bugs, uma vez que correções aplicadas em um local podem ser esquecidas em outros, resultando em comportamentos inconsistentes no sistema.

c. *Complexidade Cognitiva Alta*

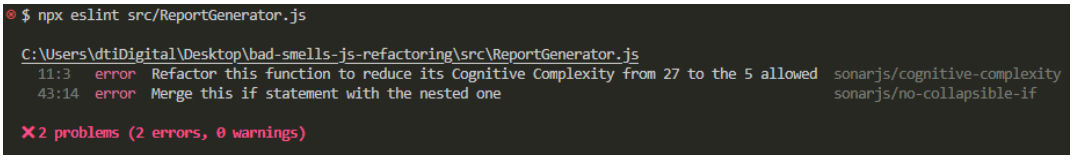
O método *generateReport()* apresenta uma complexidade cognitiva de 27, detectado pelo ESLint, valor muito acima do limite recomendado de 15. Essa complexidade excessiva é resultado de múltiplos níveis de aninhamento de condicionais, com estruturas de if-else dentro de outros if-else e loops. Esse cenário torna o código difícil de manter, pois os desenvolvedores precisam lidar com diversos contextos mentais simultaneamente para compreender o fluxo completo. Além disso, aumenta a complexidade dos testes, já que o número de caminhos de execução a serem verificados cresce exponencialmente. Por fim, eleva o risco de bugs, uma vez que quanto maior a complexidade, maior a probabilidade de ocorrência de erros lógicos não detectados.

2. *Relatório da Ferramenta*

O ESLint foi configurado com o plugin ***eslint-plugin-sonarjs*** para detectar automaticamente code smells.

a. Resultado da Análise Automática

Ao executar ***npx eslint src/ReportGenerator.js*** no código original (antes da refatoração), foram detectados os seguintes problemas:



O método ***generateReport()***, localizado na linha 11, apresenta complexidade cognitiva alta, com valor de 27, acima do limite recomendado. Isso significa que o código é difícil de

entender e manter, pois há muitas decisões e ramificações dentro do mesmo método.

Além disso, na linha 43, foi identificado um problema de condicional aninhada que poderia ser simplificada. Esse tipo de estrutura deixa o código mais confuso e complicado de ler, dificultando o trabalho de quem precisar fazer manutenção no futuro.

b. Como o eslint-plugin-sonarjs Ajudou a Identificar Problemas

O eslint-plugin-sonarjs foi essencial para identificar problemas que uma análise manual deixaria passar. A ferramenta permitiu medir a complexidade cognitiva do método ***generateReport***, mostrando que ele tinha um valor de 27, muito acima do limite recomendado. Enquanto a análise manual apenas apontava que o método era "complexo", o plugin mostrou exatamente quão grave era o problema, com base em fatores como níveis de aninhamento, operadores lógicos e estruturas de controle.

Além disso, o plugin ajudou a detectar padrões específicos de código problemático, como ifs aninhados desnecessários, identificados pela regra ***no-collapsible-if*** na linha 43. Ele indicou de forma precisa onde simplificações poderiam ser feitas, algo que seria mais difícil de perceber apenas lendo o código.

O eslint-plugin-sonarjs também mostrou problemas que passariam despercebidos, mostrando que a complexidade cognitiva é diferente da complexidade ciclomática (número de caminhos de execução). Mesmo que o método tivesse poucos caminhos, ainda era difícil de compreender por conta do excesso de aninhamentos.

Por fim, a ferramenta trouxe consistência e padronização ao processo de revisão de código, já que suas regras seguem os padrões do SonarQube, garantindo que todos os desenvolvedores usem os mesmos critérios de qualidade.

A combinação das duas abordagens resultou em uma análise mais completa e confiável do código.

3. Processo de Refatoração

a. **Smell Mais Crítico: Complexidade Cognitiva Alta**

O smell mais crítico identificado foi a Complexidade Cognitiva Alta (27 vs. limite de 15), que estava diretamente relacionada aos outros smells (Método Longo e Duplicação). A refatoração focou em reduzir essa complexidade através de técnicas de extração de métodos e simplificação de condicionais.

b. Técnicas de Refatoração Aplicadas

i. 1. **Extrair Método -** Quebrar o método longo em métodos menores e mais focados.

1. **filterItemsByUserRole()**: Filtra itens baseado no papel do usuário
2. **processAdminItem():** Processa item específico para admin
3. **generateHeader()**: Gera cabeçalho do relatório
4. **generateBody()**: Gera corpo do relatório
5. **formatItem()**: Formata um item individual
6. **generateFooter()**: Gera rodapé do relatório
7. **calculateTotal()**: Calcula total dos valores

ii. **2. Substituir Número Mágico por Constante**

Antes:

```
if (item.value <= 500) {

}
if (item.value > 1000) {

}
```

Depois:

```
static MAX_USER_VALUE = 500;
static PRIORITY_THRESHOLD = 1000;

if (item.value <= ReportGenerator.MAX_USER_VALUE) {

}
if (item.value > ReportGenerator.PRIORITY_THRESHOLD) {
}
```

iii. ***Decompor Condicional*** **-** Simplificar condicionais extraindo a lógica para métodos específicos.

Antes:

```
for (const item of items) {
  if (user.role === 'ADMIN') {
    if (item.value > 1000) {
      item.priority = true;
    }
    if (reportType === 'CSV') {

    } else if (reportType === 'HTML') {

    }
  } else if (user.role === 'USER') {
    if (item.value <= 500) {
      if (reportType === 'CSV') {

      } else if (reportType === 'HTML') {

      }
    }
  }
}
```

Depois:

```
const filteredItems = this.filterItemsByUserRole(user, items);
const body = this.generateBody(reportType, user, filteredItems);
```

c.

Métricas	**Antes**	**Depois**
Complexidade Cognitiva	27	1
Linhas no método principal	57	7
Métodos extraídos	0	7
Números mágicos	2	0

4. Conclusão

Durante a refatoração, os testes do arquivo ***ReportGenerator.test.js*** foram importantes, funcionando como uma rede de segurança. A cada mudança no código, os testes eram executados para garantir que nada fosse quebrado e que o comportamento do programa continuasse o mesmo.

A redução dos bad smells trouxe várias melhorias para o código. A manutenção ficou mais fácil, pois agora os métodos são menores e têm funções bem definidas, permitindo que futuras mudanças afetem apenas partes específicas do código. A testagem também se tornou mais simples, já que cada método pode ser testado separadamente e a complexidade geral diminuiu. A leitura do código ficou mais clara e organizada, com nomes de métodos descritivos que ajudam a entender o que o programa faz sem precisar de muitos comentários. Além disso, o código está mais flexível, permitindo adicionar

novos formatos de relatório ou tipos de usuários sem grandes modificações. Por fim, a confiabilidade aumentou, já que a eliminação de duplicações e o uso de constantes nomeadas reduzem o risco de erros e deixam o sistema mais estável.