Detecção de Bad Smells e Refatoração Segura

Teste de software - PUC Minas

Maisa Pires de Andrade

807268

Análise de Smells

Durante a leitura e análise manual do código original do arquivo src/ReportGenerator.js, foram identificados três tipos principais de Bad Smells que dificultavam a manutenção e aumentavam o risco de falhas em futuras modificações.

O primeiro problema foi o método longo. O método generateReport concentrava toda a lógica de geração, formatação, cálculo e filtragem dos relatórios em um único bloco com dezenas de linhas. Esse tipo de estrutura torna o código difícil de compreender e de testar isoladamente. Qualquer mudança em uma parte do método pode afetar o comportamento de outras seções, o que aumenta a chance de introduzir erros. Além disso, dificulta a reutilização de partes específicas do código.

O segundo smell encontrado foi a alta complexidade condicional. Havia vários blocos if/else aninhados verificando tanto o tipo de relatório quanto o papel do usuário. Essa estrutura torna o fluxo de controle confuso e difícil de expandir. Por exemplo, adicionar um novo tipo de usuário ou formato de relatório exigiria alterar múltiplas partes do código, o que viola o princípio de extensão sem modificação (Open/Closed Principle).

O terceiro smell foi a duplicação de código. As seções que geravam o corpo do relatório repetiam a mesma lógica tanto para o formato CSV quanto para HTML, mudando apenas o modo de concatenação das strings. Essa duplicação aumenta o esforço de manutenção, pois qualquer correção ou ajuste precisaria ser replicado em vários trechos.

Esses três smells comprometiam diretamente a testabilidade e a legibilidade do código. Métodos extensos e condicionais complexas dificultam a criação de testes unitários específicos, pois muitas funcionalidades ficam misturadas em um único escopo.

Relatório da Ferramenta

A execução do comando npx eslint src/ com o plugin eslint-plugin-sonarjs confirmou os problemas detectados manualmente. O resultado foi o seguinte:

/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

✖ 2 problems (2 errors, 0 warnings)

O aviso sobre cognitive complexity indicou que o método principal estava com uma complexidade de 27, muito acima do limite recomendado de 15. Essa métrica considera o número de condicionais, ramificações e aninhamentos necessários para compreender o

fluxo lógico de uma função. Já o erro no-collapsible-if apontou a presença de condicionais redundantes e aninhadas desnecessariamente.

A ferramenta foi essencial para confirmar quantitativamente os problemas de design percebidos manualmente. Enquanto a leitura do código já mostrava que o método era extenso, o eslint-plugin-sonarjs ofereceu uma medição objetiva da complexidade cognitiva, algo que nem sempre é evidente apenas pela inspeção visual. Esse tipo de feedback ajuda a priorizar quais trechos do código mais exigem refatoração.

Processo de Refatoração

O smell mais crítico identificado foi a alta complexidade do método principal, que concentrava toda a lógica do relatório. Para resolver esse problema, foi aplicada a técnica de Extract Method, quebrando o método generateReportem várias funções menores e mais específicas. Além disso, foi usada a técnica de Decompose Conditional para separar a lógica das verificações de tipo de usuário e de formato de relatório em funções próprias.

Antes:

```
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;"' : '';

      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') {

    if (item.value <= 500) {

      …  }}}
```

Depois:

```
generateRows(reportType, user, items) {

  return items.map(item =>

    reportType === 'CSV'

      ? this.formatCsvRow(item, user)

      : this.formatHtmlRow(item)

  );}
```

A lógica de repetição e formatação foi extraída para funções menores como formatCsvRow, formatHtmlRow, filterItemsByUser, markHighValueItems e filterUserItems. Essa separação reduz drasticamente a complexidade cognitiva e permite testar cada parte isoladamente. O código também ficou mais legível e aderente ao princípio da responsabilidade única (Single Responsibility Principle).

Após as mudanças, o ESLint deixou de apontar os erros de complexidade, e todos os testes unitários continuaram passando, confirmando que a refatoração manteve o comportamento original.

Conclusão

O uso de testes automatizados foi fundamental como rede de segurança durante a refatoração. A execução contínua dos testes garantiu que nenhuma funcionalidade fosse perdida enquanto o código era reestruturado. Esse processo reforça a importância de se trabalhar com cobertura de testes antes de qualquer refatoração, já que mudanças estruturais sem testes confiáveis tendem a introduzir novos defeitos.

A redução dos Bad Smells trouxe ganhos claros na legibilidade, manutenibilidade e extensibilidade do código. A separação de responsabilidades facilita futuras evoluções, como a inclusão de novos formatos de relatório, e reduz o risco de regressões. Além disso, o código refatorado torna-se mais simples de revisar, de explicar a outros desenvolvedores e de integrar em projetos maiores.

Em resumo, a combinação de análise manual, ferramentas automáticas e testes unitários demonstra uma prática essencial de engenharia de software: melhorar continuamente a qualidade do código sem comprometer sua estabilidade.