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

MATHEUS NOLASCO MIRANDA SOARES

MATRÍCULA: 688826

Bad Smells

Testes de Software

Análise de Smells

Durante a análise manual do código original da classe ReportGenerator, foram identificados três Bad Smells principais que prejudicam a manutenção e a testabilidade do sistema.

O primeiro problema identificado foi o Método Longo. O método generateReport concentrava toda a lógica de geração do relatório em um único bloco extenso, responsável por montar o cabeçalho, processar as regras de negócio para diferentes tipos de usuários, aplicar formatação em CSV e HTML, e montar o rodapé. Isso viola o Princípio da Responsabilidade Única, pois o método faz múltiplas tarefas distintas. Esse problema é crítico para a manutenção porque qualquer alteração em uma parte específica exige navegar por todo o método, aumentando o risco de introduzir bugs. Para os testes, isso significa que não é possível testar partes isoladas da lógica, tornando os testes mais complexos e frágeis.

O segundo Bad Smell identificado foi a presença de Muitos If/Else Aninhados. O código original tinha condições aninhadas para verificar o tipo de relatório, o papel do usuário e regras específicas de valor. Isso aumentava drasticamente a complexidade ciclomática do método, tornando o fluxo difícil de seguir. Adicionar um novo formato de relatório ou um novo tipo de usuário exigiria mais ramificações espalhadas pelo método. Esse problema afeta a manutenção porque aumenta a chance de erros lógicos e torna o código menos flexível para mudanças. Para os testes, isso significa maior dificuldade em alcançar cobertura completa de todos os caminhos possíveis.

O terceiro Bad Smell foram os Números Mágicos. Os valores 1000 e 500 apareciam diretamente no código sem explicação clara do que representavam. Não havia indicação se 1000 era um limite de prioridade ou 500 era o valor máximo para usuários comuns. Isso reduz a legibilidade e dificulta a manutenção, pois se esses valores precisarem mudar, é necessário localizá-los manualmente em todos os pontos do código. Para os testes, isso significa que não fica claro quais valores são significativos para os cenários de teste, dificultando a criação de casos de teste adequados.

Relatório da Ferramenta

Ao executar o comando eslint src/ no código original, a ferramenta reportou dois problemas principais. O primeiro foi a alta complexidade cognitiva do método generateReport, que apresentava complexidade de 27 quando o limite permitido era 5. O segundo foi a presença de ifs colapsáveis que poderiam ser combinados para reduzir o aninhamento.

O eslint-plugin-sonarjs se mostrou extremamente útil ao automatizar a detecção de problemas que a análise manual poderia ter deixado passar. Enquanto a análise manual identifica problemas de forma subjetiva baseando-se na experiência do desenvolvedor, a ferramenta aplica métricas objetivas como a complexidade cognitiva. Essa métrica quantifica exatamente o quão difícil é entender e manter o código, fornecendo um número concreto que facilita priorizar quais métodos precisam de refatoração urgente. A ferramenta também detectou padrões específicos como ifs colapsáveis, que são fáceis de ignorar durante uma revisão manual focada em problemas mais evidentes. Isso demonstra que a análise automática complementa a análise manual, garantindo que nenhum problema passe despercebido.

Processo de Refatoração

O smell mais crítico corrigido foi o Método Longo combinado com Muitos If/Else Aninhados. O código original tinha toda a lógica concentrada em um único método de mais de 60 linhas com múltiplos níveis de aninhamento.

A refatoração aplicada utilizou três técnicas principais. Primeiro, foi aplicado Extract Method para quebrar o método longo em métodos menores e focados. Foram criados os métodos shouldIncludeItem para decidir se um item deve ser incluído no relatório baseado no papel do usuário, applyPriorityIfNeeded para aplicar a lógica de prioridade para administradores, e getFormatter para selecionar o formatador adequado.

A segunda técnica foi Replace Conditional with Polymorphism. Em vez de ter múltiplos if/else para verificar o tipo de relatório e gerar a saída adequada, foram criadas classes separadas CsvFormatter e HtmlFormatter que implementam a interface ReportFormatter. Cada formatador sabe como gerar seu próprio cabeçalho, formatar itens e criar rodapés. Isso eliminou completamente os ifs relacionados ao tipo de relatório do método principal.

A terceira técnica foi Replace Magic Numbers with Constants. Os valores 1000 e 500 foram extraídos para constantes VALUE_PRIORITY_ADMIN e VALUE_LIMIT_USER, documentando claramente seu significado e facilitando futuras alterações. Após essas refatorações, o método generateReport passou a apenas orquestrar a execução, delegando responsabilidades específicas para métodos e classes especializadas. A complexidade cognitiva foi reduzida de 27 para aproximadamente 5, e o código ficou muito mais legível e fácil de manter.

Conclusão

O processo de refatoração demonstrou a importância fundamental de ter uma suíte de testes robusta funcionando como rede de segurança. Durante todo o processo de refatoração, os testes foram executados repetidamente para garantir que o comportamento do sistema permanecesse inalterado. Isso deu confiança para fazer alterações profundas na estrutura do código sabendo que qualquer regressão seria imediatamente detectada. Sem essa rede de segurança, refatorar código complexo seria extremamente arriscado, pois não haveria garantia de que o sistema continuaria funcionando corretamente após as mudanças.

A redução de Bad Smells melhorou significativamente a qualidade geral do software em várias dimensões. O código ficou mais legível, facilitando que outros desenvolvedores compreendam rapidamente o que cada parte faz. A manutenibilidade aumentou drasticamente, pois agora alterações podem ser feitas em pontos específicos sem afetar o resto do sistema. A extensibilidade também melhorou, tornando trivial adicionar novos formatos de relatório ou tipos de usuário sem modificar código existente. Por fim, a testabilidade foi aprimorada, permitindo criar testes mais focados e específicos para cada componente isoladamente. Esses benefícios se acumulam ao longo do tempo, reduzindo o custo de manutenção e aumentando a velocidade de desenvolvimento de novas funcionalidades.