Universidade Católica de Brasília

Pró-Reitoria Acadêmica
Curso de Ciência da Computação

ANÁLISE ESTÁTICA E REFATORAÇÃO DE CÓDIGO EM UM PROJETO REAL
Qualidade de Software e Governança

Autor: Arthur Lemos Bendini
Orientador: William Roberto Malvezzi

Brasília – DF
2024

Arthur Lemos Bendini

ANÁLISE ESTÁTICA E REFATORAÇÃO DE CÓDIGO EM UM PROJETO REAL

Trabalho desenvolvido no curso de graduação em Ciência da Computação da Universidade Católica de Brasília, como requisito parcial para obtenção do título de (Bacharel/Licenciado) em Ciência da Computação.

Orientador: William Roberto Malvezzi

Brasília

2024

Resumo

Este trabalho refere-se à análise estática e refatoração de um projeto de código aberto incluindo complexidade ciclomática, cobertura de código, dívida técnica e vulnerabilidades de segurança. Com base na análise realizada, são identificados problemas notáveis, seguidos pela elaboração de um plano de refatoração para melhorar a legibilidade, manutenibilidade e desempenho do software, assegurando a preservação do comportamento funcional original. A atividade inclui a implementação das melhorias propostas e a apresentação dos resultados obtidos.

Palavras-chave: análise estática, refatoração, qualidade do código, complexidade ciclomática, desempenho.

Abstract

This paper concerns the static analysis and refactoring of an open-source project, including cyclomatic complexity, code coverage, technical debt, and security vulnerabilities. Based on the analysis, notable issues are identified, followed by the development of a refactoring plan to improve the software's readability, maintainability, and performance, ensuring the preservation of the original functional behavior. The activity includes the implementation of the proposed improvements, and the presentation of the results obtained.

Keywords: *static analysis, refactoring, code quality, cyclomatic complexity, performance.*

SUMÁRIO

1. INTRODUÇÃO

A refatoração de código e a análise estática são práticas essenciais para garantir a qualidade e a manutenibilidade de software. A análise estática permite identificar problemas como complexidade elevada, dívidas técnicas e vulnerabilidades de segurança, sem a necessidade de executar o código. Enquanto a refatoração aplica mudanças estruturais para melhorar a clareza e modularidade do código, reduzindo sua suscetibilidade a erros futuros. Neste trabalho, essas práticas foram aplicadas a um sistema em Java, previamente utilizado para uma apresentação em grupo para a matéria de Programação Orientada a Objetos, demonstrando como essas técnicas contribuem para a melhoria contínua de um software.

2. OBJETIVO

Melhorar a qualidade, a legibilidade e a manutenibilidade de um sistema em Java por meio de técnicas de análise estática e refatoração de código, sem alterar o comportamento externo do software.

2.1 OBJETIVO GERAL

Avaliar o código existente para identificar pontos de melhoria e garantir que o sistema mantenha seu comportamento original após as modificações, e se possível, aprimorar a sua eficiência após as alterações.

2.2 OBJETIVOS ESPECÍFICOS

Contribuir para a compreensão das técnicas de análise estática e refatoração de código para uso em projetos futuros, assim melhorando a legibilidade, manutenibilidade, segurança, eficiência e qualidade geral dos códigos a serem desenvolvidos durante atividades acadêmicas e no trabalho formal.

3. ESTRUTURA DO PROJETO

O projeto a ser analisado nesse relatório foi realizado como atividade avaliativa **N1 – AT2** da matéria **Programação Orientada a Objetos**. O código fonte do projeto, junto com sua refatoração completa e testada, está no repositório do *GitHub* abaixo:

https://github.com/leviint/Code-Refactor-UCB-QSG

Batizado de *TaskFlow* durante a produção, seu propósito consiste em ser um sistema de gerenciamento de projetos, onde é possível criar projetos com prazos específicos, além de adicionar, editar e remover funcionários associados a esses projetos.

O objetivo do projeto é facilitar a organização e o acompanhamento de tarefas e prazos, proporcionando uma interface via terminal simples e intuitiva.

A estrutura de arquivos do projeto é feita da seguinte forma:

```
.
└── pacote/
    ├── Funcionario.java
    ├── Principal.java
    └── Projeto.java
```

Cada arquivo dentro da pasta “pacote” é uma Classe em Java.

4. ANÁLISE ESTÁTICA DE CÓDIGO

A análise estática de código é um processo automatizado de revisão de código fonte que identifica problemas de qualidade, como complexidade excessiva, duplicação, dívidas técnicas e vulnerabilidades de segurança, sem executar o programa. Ela funciona examinando o código diretamente, avaliando sua estrutura e padrões em busca de inconsistências e possíveis erros.

Ferramentas de análise estática aplicam regras predefinidas e métricas para detectar más práticas, calcular a complexidade ciclomática e apontar trechos de código que podem ser otimizados ou que apresentam riscos, ajudando desenvolvedores a garantir uma base de código mais segura, eficiente e fácil de manter.

Também é possível que o próprio desenvolvedor do código faça essa análise sem a necessidade de uma ferramenta específica para esse fim. Isso muitas vezes é útil caso o desenvolvedor queira revisitar um projeto anterior para melhorá-lo.

4.1 COMPLEXIDADE CICLOMÁTICA

A complexidade ciclomática mede a quantidade de caminhos independentes em um código, calculando o número de ramificações e decisões, como estruturas de condição (*if, switch*) e laços de repetição (*for, while*). Pode ser calculada pela seguinte fórmula, introduzida por Thomas J. McCabe:

$$\mathrm{M} = E - N + 2P$$

Onde:

- E é o número de arestas (transições entre os nós) no gráfico de controle de fluxo do programa;
- N é o número de nós (pontos de decisão);
- P é o número de componentes conectados.

Quanto maior a complexidade, mais difícil é testar e manter o código. Valores altos sugerem a simplificação do fluxo com a refatoração, tornando o código mais fácil de entender e gerenciar.

4.1.2 Cálculo da Complexidade Ciclomática

Analisando as classes do projeto separadamente, conclui-se que a classe “Projeto” possui:

- **E:** 12;
- **N:** 9;
- **P:** 1.

Já a classe “Funcionario” possui:

- **E:** 6;
- **N:** 4;
- **P:** 1.

E por fim, a classe “Principal” possui:

- **E:** 6;
- **N:** 6;
- **P:** 1.

Assim, a fórmula para o cálculo de complexidade ciclomática será:

- Para a classe “Projeto”:

$$\mathrm{M} = E - N + 2P = 12 - 9 + 2 \times 1 = 5$$

- Para a classe “Funcionario”:

$$M = E - N + 2P = 6 - 4 + 2 \times 1 = 4$$

- Para a classe “Principal”:

$$M = E - N + 2P = 6 - 6 + 2 \times 1 = 2$$

Sabendo das informações acima, a complexidade ciclomática total do projeto equivale a 11.

4.2 COBERTURA DE CÓDIGO

A cobertura de código (*code coverage)* é uma métrica utilizada para avaliar o percentual do código total executado durante uma série de testes, o que ajuda a identificar áreas não testadas ou executadas adequadamente.

Por exemplo, pode existir uma função em um programa que está completa, mas ela não é executada no código por nunca ter sido chamada.

O código original do projeto carecia de modularização em alguns trechos, o que dificulta a realização de testes.

Visto isso, é necessário refatorar e dividir responsabilidades entre os métodos para facilitar testes unitários mais precisos e corrigir possíveis erros e *bugs*.

4.2.1 Cobertura do código do projeto

4.2.1.1 Classe Funcionario

Esta classe é simples e tem uma boa cobertura em termos de funcionalidade. Ela inclui Construtores Funcionario() e Funcionario(*String, String, double*) para inicializar objetos, além de métodos *getters* e *setters* para os atributos *Name*, cargo e Salario.

Potenciais coberturas de testes seriam a instanciação de objetos com o construtor padrão e o construtor parametrizado, e a verificação de *getters* e *setters* para cada atributo.

O código original não realiza verificações de integridade de dados, como assegurar que o salário não seja negativo. Porém, de modo geral, a cobertura dessa classe está completa em termos de funcionalidade.

4.2.1.2 Classe Projeto

Uma classe mais complexa, pois oferece diversas funcionalidades. Nela, é possível criar projetos com prazo de entrega, possui métodos para adicionar, remover, listar e editar detalhes de funcionários, e conta com a presença de um método para validar e verificar prazos.

Nessa classe, potenciais coberturas de testes incluem:

- a verificação de data de entrega inválida ao criar um projeto;
- testar o método adicionarFuncionario para garantir que funcionários sejam adicionados corretamente;
- testar os métodos removerFuncionarioPorNome e editarFuncionarioPorNome incluindo cenários onde o funcionário existe e não existe;
- Confirmar o método listarFuncionarios, tanto com lista vazia quanto com vários funcionários;
- Validar os métodos prazoEntregaValido e estaDentroDoPrazo, especialmente com diferentes datas (podendo ser inválidas, passadas e futuras);
- Checar a formatação de datas em exibirPrazo.

O código original possui tratamentos de exceções para entradas inválidas, mas seria importante garantir que todos os caminhos sejam testados. Pontos adicionais que poderiam ser verificados incluem a cobertura de cenários como funcionários duplicados ou alteração de dados após a adição inicial.

4.2.1.3 Classe Principal

Essa classe representa a interface do usuário do programa. Ela oferece menus para navegar pelo sistema e permitindo que o usuário gerencie os projetos e funcionários. O sistema inclui menus para criação, remoção e verificação de prazos de projetos, também possui funções para gerenciar funcionários e atribuí-los a projetos específicos.

Possíveis coberturas de testes incluem testar as opções de menu (menuPrincipal, GerenciarProjeto, GerenciarFuncionariosProjeto), verificando se a navegação ocorre da maneira correta, além de testar adicionarFuncionario, removerFuncionario e editarFuncionario com cenários variados. Também vale checar cenários onde não há projetos cadastrados, que deve redirecionar o usuário ao menu principal.

4.3 DÍVIDA TÉCNICA

Dívida técnica se refere ao cumulado de decisões de desenvolvimento de software que priorizam soluções rápidas e de curto prazo em detrimento de uma arquitetura mais robusta e sustentável. Decisões assim são motivadas pela necessidade de entregar código rapidamente, por falta de tempo para realizar um trabalho melhor, ou por impaciência e preguiça do desenvolvedor, o que resulta em um código de difícil manutenção.

No projeto analisado, são observadas diversas inconsistências relacionadas à conformidade de nomenclatura de variáveis, classes, métodos e atributos. Alguns não obedecem aos padrões *camelCase*, e possuem inconsistência na língua, alguns elementos estando em língua portuguesa, e outros na língua inglesa.

Não só isso, como alguns métodos estão localizados na classe errada e acabam acrescentando volume de código desnecessário à classe Principal.

4.4 CÓDIGO DUPLICADO

O método GerenciarProjeto encontrado na classe “Principal” possui uma repetição lógica na criação e na verificação de prazos. É recomendável refatorar essa lógica em métodos separados em virtude da reutilização e clareza.

4.5 MÁS PRÁTICAS

Más práticas de programação referem-se a abordagens que dificultam a manutenibilidade, segurança e eficiência do código. No projeto analisado, foram identificados diversos exemplos, incluindo nomenclatura inconsistente entre classes, métodos e variáveis, como citado anteriormente.

Além disso, Métodos extensos e não modulados, como GerenciarProjeto - que por si já representa uma divergência do padrão camelCase - possuem múltiplas responsabilidades, o que vai contra o princípio SRP (*Single Responsibility Principle*), tornando o código menos modular e difícil de testar.

Por outro lado, em alguns casos, como na manipulação de prazos de entrega, o código poderia se beneficiar de um tratamento mais robusto de exceções para evitar erros inesperados em execução.

Essas práticas foram revistas e modificadas na refatoração, visando padronizar o código e torná-lo mais legível e seguro.

4.6 VULNERABILIDADES DE SEGURANÇA

Na análise também foram identificadas áreas de vulnerabilidade. Alguns atributos e métodos não possuem proteção contra acessos indevidos de dados no código, o que poderia causar erros e falhas de segurança se não tratados de forma correta.

Vale constar que outro ponto relevante é a falta de validação rigorosa de entrada do usuário, especialmente nas funções de adição e edição de dados. Isso pode levar a comportamentos inesperados e riscos de segurança, como injeções de dados maliciosos.

5. REFATORAÇÃO DE CÓDIGO

A refatoração de código é o passo final do processo de métrica de software. Ela deve ser realizada após fazer toda a identificação do problema e o planejamento prévio de refatoração, apontando todos os problemas que devem ser corrigidos.

A refatoração apropriada do projeto, levando em consideração todos os pontos comentados neste relatório, está disponível no repositório do *GitHub* citado no começo do documento.

5.1 IDENTIFICAÇÃO DE PROBLEMAS

Os problemas de complexidade, modularidade, nomeação inconsistente, código duplicado e más práticas foram identificados após a leitura, análise do código e documentação dos problemas nesse relatório, visando corrigir esses problemas de forma eficaz e simples.

5.2 PLANO DE REFATORAÇÃO

1. Modularizar métodos complexos;
    - Quebrar métodos extensos em sub-métodos para simplificar a lógica do código.
2. Padronizar nomes;
    - Ajustar a nomenclatura inconsistente. Corrigir o padrão camelCase, manter a uniformidade de idioma para a língua portuguesa.
3. Tratar exceções de forma mais robusta;
    - Adicionar tratamento para dados inválidos, reforçar a integridade nas entradas do usuário.
4. Remover/Readaptar código duplicado.
    - Reestruturar o método GerenciarProjeto (além de corrigir sua nomenclatura para gerenciarProjeto) em sub-funções que podem ser reutilizadas.

5.3 IMPLEMENTAÇÃO

A implementação do plano de refatoração resultou em um código mais limpo e fácil de manter. A modularização facilitou a execução de testes unitários, enquanto a padronização da nomenclatura melhorou a legibilidade e padronização do código, e o tratamento de exceções ajudaram a reduzir possíveis erros e vulnerabilidades.

6. CONCLUSÃO

Este trabalho ilustrou a importância da análise estática e da refatoração para melhorar a qualidade de um código revisitado. Com as métricas obtidas e o plano de refatoração executado, o projeto agora apresenta uma estrutura mais modular, segura, legível e fácil de manter. O uso dessas práticas proporcionou uma visão clara dos benefícios da refatoração para a vida útil de um sistema. Além disso, forneceu novos aprendizados e conhecimentos para serem levados ao longo de uma carreira como desenvolvedor.

7. REFERÊNCIAS BIBLIOGRÁFICAS

1 GOVERNO DO ESTADO DE PERNAMBUCO. **Métricas de software: definições.** Disponível em: http://www.portaisgoverno.pe.gov.br/web/metricas-de-software/definicoes. Acesso em: 05 nov. 2024.

2 Tetila, E., Costa, I., de Mesquita Spínola, M., & Queiroz da Silva Tetila, J. (2011). **Processo de estimativa de software com a métrica use case points,** PMBOK e RUP. *Iberoamerican journal of industrial engineering*, *3*(5), 249–264. Disponível em: https://incubadora.periodicos.ufsc.br/index.php/IJIE/article/view/666. Acesso em: 05 nov. 2024.