Universidade de Brasília
Faculdade do Gama
Engenharia de Software

Matheus Henrique Dos Santos – 211029666
Sidney Fernando Ferreira Lemes - 190037997

Relatório do Trabalho: Projeto de Software

Brasília
2024

1 INTRODUÇÃO

O desenvolvimento de software envolve a criação e evolução contínua de código, onde classes, objetos e estruturas de dados são gradualmente incorporados ao projeto. À medida que o sistema cresce, a qualidade do projeto se torna crucial para garantir a manutenibilidade, extensibilidade e eficiência do software. Um projeto de software bem estruturado é caracterizado por ser fácil de escrever, entender, manter e evoluir, além de ser resiliente a mudanças e menos propenso a erros.

Neste contexto, a refatoração se apresenta como uma prática essencial para a melhoria contínua do código, como Martin Fowler escreveu em seu livro sobre refatoração, a refatoração é uma alteração feita na estrutura interna do software para torná-lo mais fácil de entender e mais barato de modificar, sem alterar seu comportamento observável. A refatoração é fundamental para assegurar que os princípios de um bom projeto de código, como simplicidade, elegância, modularidade e extensibilidade, sejam mantidos ao longo do ciclo de vida do software.

2 PRINCÍPIOS DE BOM PROJETO DE CÓDIGO

Este relatório explora o uso de técnicas de refatoração em um projeto de software, abordando como essas técnicas podem ser aplicadas para corrigir "maus-cheiros" de código identificados. Serão discutidos os princípios de um bom projeto de código, conforme descrito por Fowler (1999) e Goodliffe (2006), e sua relação com os problemas de design observados. Além disso, serão indicadas as operações de refatoração que são aplicáveis e quais foram realizadas para melhorar a qualidade do código do trabalho.

2.1 SIMPLICIDADE

A simplicidade está relacionada com como o código é claro e direto, evitando complexidades desnecessárias. Ela facilita a compreensão e a manutenção do código, deixando-o mais fácil a identificação de problemas e a implementação de novas funcionalidades.

Os maus-cheiros apresentados por Fowler que podem violar esse princípio são: Código Duplicado, Método Longo, Classe Grande, Lista Longa de Parâmetros, Mudanças Divergentes, Aglomerados de Dados, Instruções Switch, Generalidade Especulativa, Campo Temporário, Cadeias de Mensagens, Classes Alternativas com Interfaces Diferentes e Comentários.

2.2 ELEGÂNCIA

A elegância combina clareza com eficiência, utiliza soluções que são funcionais e fáceis de entender. O objetivo é criar código que seja não apenas funcional, mas também esteticamente satisfatório, sem excesso. A falta de elegância pode resultar em código mal organizado ou confuso, o que torna o sistema difícil de compreender e modificar.

Os maus-cheiros apresentados por Fowler (1999, p. 63-72) que podem violar esse princípio são: Método Longo, Instruções Switch, Generalidade Especulativa, e Comentários.

2.3 MODULARIDADE

A modularidade propõe a divisão do software em partes menores, cada uma com uma responsabilidade específica e bem definida. Isso facilita a manutenção, testes e futuras expansões, pois mudanças e melhorias em um único módulo tendem a ser mais fáceis. Sem a modularidade, o código tende a se tornar monolítico e difícil de gerenciar.

Os maus-cheiros apresentados por Fowler (1999, p. 63-72) que podem violar esse princípio são: Código Duplicado, Método Longo, Classe Grande, Mudanças Divergentes, Cirurgia com “Rifle” (Shotgun Surgery), Inveja de Recursos, Aglomerados de Dados, Obsessão Primitiva, Hierarquias de Herança Paralelas, Classe Preguiçosa, Homem do Meio, Intimidade Inapropriada, Biblioteca de Classes Incompleta, Classe de Dados e Herança Negada.

2.4 BOAS INTERFACES

Interfaces bem projetadas facilitam a interação entre diferentes partes do sistema, sendo intuitivas e claras, enquanto escondem a complexidade interna. Elas devem ser fáceis de entender e usar, minimizando erros e confusões. Quando mal

definidas, elas podem dificultar a integração de diferentes componentes e levar a problemas de usabilidade.

Os maus-cheiros apresentados por Fowler (1999, p. 63-72) que podem violar esse princípio são: Lista Longa de Parâmetros, Cadeias de Mensagens, Intimidade Inapropriada, Classes Alternativas com Interfaces Diferentes, Classe de Dados e Herança Negada.

2.5 EXTENSIBILIDADE

Está ligada à capacidade do código de ser facilmente expandido com novas funcionalidades, sem exigir grandes modificações na estrutura existente. Isso permite que o software acompanhe mudanças e novos requisitos de forma eficiente.

Os maus-cheiros apresentados por Fowler (1999, p. 63-72) que podem violar esse princípio são: Classe Grande, Cirurgia com “Rifle” (Shotgun Surgery), Instruções Switch e Hierarquias de Herança Paralelas.

2.6 EVITAR DUPLICAÇÃO

Evitar duplicação pode ser definida com a garantir que um mesmo código não seja repetido em várias partes do sistema. Isso reduz a possibilidade de inconsistências e facilita a manutenção, pois uma mudança em um local não precisa ser replicada em vários outros. Esse é um dos problemas mais comuns e pode levar a dificuldades na manutenção e evolução do software.

Fowler dedica um mau-cheiro somente para lidar com a violação desse princípio, o Código Duplicado.

2.7 PORTABILIDADE

Está relacionada à facilidade de se adaptar para diferentes ambientes e plataformas. A portabilidade assegura que o software possa ser reutilizado ou transferido entre diferentes contextos com esforço mínimo. A ausência de portabilidade pode criar dependências que tornam o software difícil de mover para novos ambientes.

Os maus-cheiros apresentados por Fowler (1999, p. 63-72) que podem violar esse princípio são: Obsessão Primitiva, Instruções Switch e Biblioteca de Classes Incompleta.

2.8 CÓDIGO DEVE SER IDIOMÁTICO E BEM DOCUMENTADO

Pode ser definido como sendo orientar-se pelas melhores práticas e convenções da linguagem de programação usada, facilitando a leitura e a manutenção por outros desenvolvedores. Além disso, uma boa documentação é essencial para explicar claramente o propósito e o funcionamento do código. A violação desse princípio gera dificuldade no entendimento do código.

Os maus-cheiros apresentados por Fowler (1999, p. 63-72) que podem violar esse princípio são: Obsessão Primitiva, Cadeias de Mensagens, Intimidade Inapropriada e Classe de Dados.

3 MAUS-CHEIROS PERSISTENTES NO TRABALHO 2

Realizado o trabalho prático 2, foi identificado ainda alguns maus-cheiros a serem tratados.

3.1 CLASSE Cliente e SUBCLASSES

3.1.1 Mau-Cheiro: Classe de Dados

Descrição: As classes Cliente, ClienteEspecial e ClientePadrao funcionam principalmente como contêineres de dados, sem um comportamento mais significativo.

Princípios Afetados: Modularidade.

Operações de Refatoração Aplicáveis:

- **Adicionar Métodos**: realizar operações relevantes diretamente nas classes clientes, como cálculos ou validações específicas para cada tipo de cliente.

3.2 CLASSE Endereco

3.2.1 Mau-Cheiro: Instruções Switch

Descrição: O método *definirRegiao* usa um switch para mapear estados para regiões, o que pode ser difícil de manter.

Princípios Afetados: Extensibilidade e Modularidade.

Operações de Refatoração Aplicáveis:

- **Substituir por Estrutura de Dados:** utilizar um mapa ou uma estrutura de dados mais flexível para associar estados a regiões. Isso facilita a adição de novos estados ou regiões e melhora a manutenção.

3.3 CLASSE Venda

3.3.1 Mau-Cheiro: Classe Grande

Descrição: A classe Venda combina várias responsabilidades, como gerenciamento de itens e cálculos de impostos.

Princípios Afetados: Simplicidade e Modularidade.

Operações de Refatoração Aplicáveis:

- **Extrair Classe**: separar as responsabilidades em diferentes classes, uma classe para gerenciar itens e outra para cálculos de impostos. Isso torna o código mais modular.

3.3.2 Mau-Cheiro: Instruções Switch

Descrição: A lógica do método *valorFrete* pode ser melhorada.

Princípios Afetados: Extensibilidade e Modularidade.

Operações de Refatoração Aplicáveis:

- **Utilizar Padrões de Design**: usar padrão de design Strategy, para calcular o frete com base na região. Isso oferece uma maneira mais extensível e modular de lidar com diferentes regras de cálculo.

3.4 CLASSE RelatorioVenda

3.4.1 Mau-Cheiro: Inveja de Recursos

Descrição: O método *verificaClienteEspecial* realiza verificações com base em informações do cliente.

Princípios Afetados: Modularidade e Boas Interfaces.

Operações de Refatoração Aplicáveis:

- **Mover Método:** transferir a lógica de verificação para a classe Cliente ou criar um serviço específico para essa lógica.

3.5 CLASSE CalculadoraNota

3.5.1 Mau-Cheiro: Inveja de Recursos

Descrição: O método *aplicarCashBack* manipula o estado de *ClientePrime*.

Princípios Afetados: Modularidade e Boas Interfaces.

Operações de Refatoração Aplicáveis:

- **Mover Método:** transfirir a lógica de cashback para a classe ClientePrime ou criar uma classe específica para gerenciar as operações de cashback.

3.5.2 Mau-Cheiro: Classe de Dados

Descrição: A classe age como um contêiner de dados que utiliza valores diretamente fornecidos por Venda para calcular o total da nota. Em vez de encapsular lógica de cálculo de forma independente, ela depende fortemente da classe Venda.

Princípios Afetados: Modularidade e Boas Interfaces.

Operações de Refatoração Aplicáveis:

- **Extrair Classe:** utilizar a classe dedicada para o cálculo de notas para encapsular a lógica de forma apropriada.

3.5 CLASSE ItemVenda

3.5.1 Mau-Cheiro: Classe de Dados

Descrição: A classe somente armazena dados sobre um item de venda, sem adicionar comportamentos significativos.

Princípios Afetados: Simplicidade e Modularidade.

Operações de Refatoração Aplicáveis:

- **Adicionar Comportamento:** incluir métodos que realizem operações relevantes relacionadas ao item de venda, como cálculo de desconto.

4 CONSIDERAÇÕES FINAIS

Neste relatório, analisamos a aplicação de técnicas de refatoração em um projeto de software, abordando como essas técnicas podem melhorar a qualidade do código, corrigindo maus-cheiros identificados. As refatorações sugeridas foram orientadas pelos princípios de um bom projeto de código, incluindo simplicidade, elegância, modularidade, boas interfaces, e extensibilidade.

As refatorações sugeridas visam tornar o código mais claro e direto, eliminando complexidades desnecessárias e garantindo que ele seja fácil de manter e expandir. A modularidade foi aprimorada ao dividir o código em partes menores e bem definidas, o que facilita tanto a manutenção quanto a adição de novas funcionalidades. As operações de refatoração também abordaram a duplicação de código e a portabilidade, garantindo que o sistema seja mais eficiente e adaptável a diferentes contextos. Além disso, ao melhorar as interfaces, asseguramos que diferentes partes do sistema possam interagir de forma clara e intuitiva, minimizando erros e dificuldades de integração.

Em resumo, as refatorações propostas e realizadas alinham o código com os princípios de um bom projeto de código, resultando em um sistema mais robusto, flexível e preparado para evoluções futuras. Isso não apenas reduz o risco de introdução de novos erros, mas também facilita a continuidade do desenvolvimento, assegurando que o software possa crescer e se adaptar às necessidades em constante mudança com alta qualidade.

REFERÊNCIAS

MARTIN FOWLER. **Refactoring: Improving the design of Existing Code**. 1. ed. Addison-Wesley Professional, 1999.

PETE GOODLIFFE. **Code Craft: The practice of Writing Excellent Code.** No Starch Press, 2006.