Universidade de Brasília

Faculdade do Gama

TP3 - Avaliação e Análise do Projeto Realizado para a Matéria de Técnicas de Programação em Plataformas Emergentes

Autor:

Levi Queiroz 17/0108341

Matéria:

FGA0242 - Técnicas de Programação em Plataformas Emergentes

Brasília
27 de agosto de 2024

Conteúdo

1 Resumo

No presente trabalho é apresentado "maus-cheiros", de acordo com Martin Fowler, presente no projeto de software criado pelo aluno durante o exercício do curso. Sendo apresentados também as operações de refatoração que podem ser aplicadas para mitigarem os "maus-cheiros"identificados.

Palavra-chave: "maus-cheiros", refatoração, software, Martin Fowler.

2 Introdução

Na página do github `https://github.com/andrelanna/fga0242/blob/master/tps/tp1/README.md` é apresentado o enunciado dos trabalhos 1 e 2, em que o 1 consistia em criar, a partir do cenário colocado, um programa utilizando as técnicas de Desenvolvimento Guiado por Testes (TDD) e o 2 consistia na refatoração de classes e métodos sugeridos pelo professor e as operações de refatoração a serem feitas. Assim, para este presente trabalho foi pedido a partir do enunciado apresento na página do github `https://github.com/andrelanna/fga0242/blob/master/tps/tp2/README.md` onde é apresentado a importância de realizar um bom projeto de software e é pedido ao aluno a responder esses dois itens na avaliação e análise do projeto:

- Para cada um dos princípios de bom projeto de código mencionados acima, apresente sua definição e relacione-o com os maus-cheiros de código apresentados por Fowler em sua obra.

- Identifique quais são os maus-cheiros que persistem no trabalho prático 2 do grupo, indicando quais os princípios de bom projeto ainda estão sendo violados e indique quais as operações de refatoração são aplicáveis. Atenção: não é necessário aplicar as operações de refatoração, apenas indicar os princípios violados e operações possíveis de serem aplicadas.

Sendo assim, o objetivo deste presente trabalho é apresentar as respostas desses itens a partir da avaliação do projeto criado.

3 Desenvolvimento

No desenvolvimento é apresentado o entendimento do aluno na correlação entre os princípios de um bom projeto de software e os "maus-cheiros"citado por Martin Fowler em seu trabalho ***Refactoring: Improving the Design***

of Existing Code e é apresentado as avaliações e análises do projeto apresentado os "maus-cheiros"identificado pelo aluno por cada classe criada no projeto.

3.1 Correlação dos Princípios e "Maus-Cheiros"

Para realizar um bom projeto de software tem-se como objetivo os principais princípios: Ser fácil de escrever, ser fácil de entender, ser fácil de manter, ter menor probabilidade de ter bugs e ser mais resiliente à mudanças. Satisfazendo esses princípios, o desenvolvedor ou a equipe de desenvolvimento tem um bom projeto de software e, de acordo com Martin Fowler, tem-se uma relação entre ter um bom projeto de software evitando os "maus-cheiros" no projeto.

Um código fácil de escrever é aquele que segue boas práticas de programação, como clareza, modularidade e uso adequado de abstrações. Envolve o uso de convenções de software. Tendo objetivo de que os desenvolvedores ou equipes de desenvolvimento possam implementar novas funcionalidades ou corrigir problemas rapidamente. Como também deve ser claro e legível, permitindo que outros desenvolvedores ou equipes compreendam rapidamente a funcionalidade do mesmo. Se adquire isso por meio de uma boa estrutura, comentários úteis, nomes de variáveis e funções descritivas e uma organização lógica do código. Um código fácil de entender reduz a curva de aprendizado para novos desenvolvedores e facilita a colaboração em equipe. Uma probabilidade de problema nesse ponto de escrita e entendimento é se o código tem métodos longos (Long Method), nomes enigmáticos (Inconsistent Naming), ou código duplicado (Duplicate Code). Uma boa prática de resolução desses "maus-cheiros" é através da aplicação de princípios de design como a decomposição em métodos menores, escolha de nomes significativos e reutilização de código por meio de abstrações apropriadas.

A facilidade de manter o projeto se dá na manutenção do software que envolve corrigir bugs, melhorar a funcionalidade existente e adaptar o software a novas necessidades. Um projeto de software bem organizado, com código modular e testes automatizados facilita a identificação de problemas e a implementação de melhorias. Além disso, o código deve ser flexível o suficiente para acomodar mudanças sem a necessidade de grandes reescritas. Assim, problemas como classes grandes (Large Class) ou dependências ocultas (Hidden Dependencies) acarretam dificuldades na manutenção do código, pois quaisquer alterações podem ter efeitos colaterais imprevisíveis. Sendo assim, um projeto de software que prioriza a manutenção é projetado para ser modular, coeso e pouco acoplado, evitando a proliferação de "maus-cheiros" que complicam a evolução do sistema.

Um software com menos bugs é resultado de práticas de codificação cuidadosas, como adoção de testes unitários, integração contínua e revisões de código. Além disso, seguir princípios como o desenvolvimento orientado a testes (TDD) e manter a simplicidade no design contribuem para reduzir a quantidade de erros no código. Assim, em casos como complexidade ciclomática alta (High Cyclomatic Complexity) ou divergência de funções (Divergent Change) podem indicar um código mais propenso a erros, já que complexidade excessiva ou responsabilidades mal distribuídas podem introduzir bugs. Para sanar quaisquer problemas de bugs, deve-se ter a preocupação em manter o código simples e bem estruturado, reduzindo a probabilidade de bugs através da refatoração constante para eliminar "maus-cheiros".

Mais resiliência às mudanças significa ter a capacidade de se adaptar de forma eficaz e positiva a novas situações, desafios ou adversidades. Para um software significa a capacidade do software de ser modificado sem introduzir novos bugs ou comprometer sua funcionalidade. Isso pode ser alcançado por meio de um design flexível, uso de padrões de projeto e arquitetura desacoplada, onde diferentes partes do sistema podem ser alteradas independentemente. O uso de boas práticas de versionamento e documentação também contribui para a resiliência. Um código que não é resiliente à mudanças é indicado por "maus-cheiros" como Shotgun Surgery (quando uma pequena mudança no código implica em modificações em várias partes do sistema) ou rigidez (Rigidity), que são solucionados por práticas de montar um projeto bem estruturado, pois garantem que mudanças possam ser feitas localmente e com o mínimo de impacto no restante do sistema, mitigando "maus-cheiros" por meio de um design desacoplado e aderente ao princípio da responsabilidade única.

3.2 Análise e Avaliação de "Maus-Cheiros"

Como nesse tópico serão abordados os "maus-cheiros" persistentes na entrega TP2 no projeto trabalhado pelo aluno durante o exercício do curso de Técnicas de Programação em Plataformas Emergentes, será dividido em subtópicos para cada classe, retirando as classes que foram utilizadas para enumerações, sendo elas FormaPagamento e TipoCliente.

3.2.1 Classe Venda

Iniciando-se pelo princípio de fácil mantenabilidade, a classe ficou extensa, existem métodos como 'calcularICMS()', 'calcularFrete()', 'calcularImpostoMunicipal()' e 'valorReal()' que poderiam ser utilizados como Objetos-Métodos das classes Cliente e Produto, os 3 primeiros para Cliente e o

último para Produto, para reduzir a quantidade de métodos existentes na classe Venda e assim melhorar a visualização desses métodos em utilização. No código, para a entrega do TP2, foram realizado operações de Extrair Método e Substituir Método por Objeto-Método para mitigar "maus-cheiros" de Método Longo e Divergência de Funções, que violam os princípios de fácil escrita e compreensão e menor probabilidade de bugs. Ainda assim um "mau-cheiro" que viola o princípio de menos probabilidade de ter bugs é que a classe Venda tem responsabilidades diversas, listar os produtos e somar seus preços, realizar o cálculo de icms, imposto municipal, frete, confirmar data e hora da venda, calcular cashback que o cliente vai receber e verificação de cartão da loja, algumas responsabilidades são realmente da classe venda e outras poderiam ser de responsabilidade de outras classes. Também há presente no código algumas Dependências Ocultas que dificultam a refatoração do projeto, como tive ao aplicar as operações de refatoração na classe, violando o princípio de manutenção. Para a classe, a operação de Substituição de Método para Objeto-Método, Extração de Classe e Introduzir Objeto Parâmetro reduziria a maior parte dos "maus-cheiros". Há problemas de encapsulamento, principalmente usando 'this.cliente', que deveria ser substituído por 'getCliente()'.

3.2.2 Classe UltimoMes

Um "mau-cheiro" detectado é da Inveja de Recurso que viola o princípio de segregação de interfaces, pois a classe necessita apenas de dados de outras classes, sendo elas Venda e Cliente, assim não é necessária sua existência. Poderia aplicar a operação de refatoração Incorporar Classe entre esta classe e a classe Cliente, que é o objeto que está sendo modificado pela classe. Isso facilitaria a leitura e diminuiria a quantidade de classes existentes, eliminando classes avulsas. Além disso, resolver problemas de encapsulamentos utilizando 'getCliente()' em vez de 'this.cliente'.

3.2.3 Classe Produto

O "mau-cheiro"detectado na classe foi a Classe de Dados, ou seja, não há métodos para serem utilizados pela classe, isso viola o princípio de encapsulamento, que indica uma falta de responsabilidade única, pois há métodos sendo utilizados na classe Venda que são de propriedade da classe produto. Assim a forma de resolver é por Substituição o Método por Objeto-Método trazendo o método para a classe Produto e o utilizando como objeto-método na classe Venda. O método referido é o 'valorReal()' na classe Venda.

3.2.4 Classe Endereco

Assim como a classe produto, o "mau-cheiro" detectado é a Classe de Dados, que viola o princípio de encapsulamento, que indica uma falta de responsabilidade única. A operação de refatoração para resolver esse "mau-cheiro" é Incorporar Classe na classe Cliente, pois diferentemente com a classe Produto, a classe Endereço não tem método espalhado em outras classe, assim podendo ser agrupada à classe cliente. Isso caso o método 'calcularFrete()' for inserido na classe Cliente e não na classe Endereço, o que seria outra alternativa de resolução da violação, repetindo a forma como seria resolvida na classe Produto.

3.2.5 Classe Cliente

Após a operação de Extração de Classe aplicado na classe Cliente, foi mitigado os "maus-cheiros" identificados pelo professor para a realização da entrega do TP2, assim o "mau-cheiro" encontrado que viola o princípio de encapsulamento é a utilização do 'this.calculoCliente' em vez do 'getCalculoCliente()', que seria resolvido modificando de um para o outro, respectivamente.

4 Conclusão

Por fim, no presente trabalho o aluno pode pesquisar e aprender mais sobre a importância de um bom projeto de software e como aplicar as verificações de Martin Fowler em um projeto de software.

Referências

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.