Universidade de Brasília

Faculdade do Gama (FGA)

TP3 - Grupo 2

Técnicas de Programação em Plataformas Emergentes

Visão geral

O objetivo deste documento é resolver as questões propostas no enunciado, são essas as questões:

1 - 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.
2 - 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.

O material de consulta para a formulação das respostas é o livro Refactoring: Improving the design of Existing Code. Martin Fowler

Questões

1. **Explicar os princípios de bom projeto:** Nessa primeira parte, explicaremos os princípios de bom projeto e faremos a relação deles com *maus-cheiros de código (code smells)* apresentados no livro

2. **Identificar os maus-cheiros do TP2:** Na última parte identificaremos os maus-cheiros de código a partir dos códigos do TP2 e quais operações de refatoração serão aplicáveis para ficar de acordo com os princípios de bom projeto

Questão 1 - Princípios de Fowler

1. Simplicidade

   O princípio da simplicidade se refere à facilidade com que se entende de um código. Um código que compreende esse princípio é direto, sem complexidade desnecessária. A simplicidade no código facilita a manutenção de código e o desenvolvimento do projeto. A simplicidade está relacionada com maus-cheiros como “Complexidade Artificial” e “Código Duplicado”. A Complexidade Artificial surge ao tentar antecipar mudanças futuras, que leva a um código mais complexo do que o necessário, já o Código Duplicado diz respeito a duplicação real do código, repetições de métodos, classes, dados, módulos que são desnecessárias, gerando ainda mais complexidade

2. Elegância

   O princípio da Elegância se refere a forma de resolver problemas de maneira clara, concisa, precisa, usando abordagens que aumentam a eficiência e legibilidade do código ao máximo. Pode-se relacionar com o mau-cheiro “Comentários em Excesso”, onde um código com comentários em excesso pode não ser claro o suficiente para a compreensão sem uma ajuda externa.

3. Modularidade

   O princípio da Modularidade se refere como dividir o código em módulos ou componentes distintos de forma efetiva, cada um com uma função coesa e independente, permitindo, assim, que os módulos sejam desenvolvidos, testados e mantidos separadamente. Pode-se relacionar com os maus-cheiros “Objeto Deus” e “Métodos Longos”. Objeto Deus (God Object) ocorre quando um único objeto ou módulo, acumula muitas responsabilidades e funções dentro do escopo do código. Ja Métodos Longos são métodos que realizam muitas tarefas diferentes, ao qual, se seguisse o princípio da Modularidade, deveriam estar em módulos separados.

4. Boas Interfaces

O princípio das Boas Interfaces diz que tais interfaces expressam claramente como os como os componentes ou módulos do sistema interagem, garantindo, certamente, uma excelente e eficiente comunicação, evitando dependências desnecessárias. Códigos que não seguem esse princípio, geram maus-cheiros como “Mudanças Divergentes” e “Shotgun Surgery”. Mudanças Divergentes (Divergent Change) ocorre quando uma mudança de um componente ou módulo gera várias alterações em diversas partes do sistema. Já a Shotgun Surgery ocorre de maneira similar, em que pequenas mudanças necessitam de modificações em diversos locais do código.

5. Extensibilidade

O princípio da Extensibilidade diz que é a capacidade do código ser, de maneira fácil, ampliado ou modificado para que sejam incluídas novas funcionalidades sem grandes mudanças na estrutura existente. Quando esse princípio não é obedecido, podem aparecer maus-cheiros como “Rigidez” e “Fragilidade”. A Rigidez ocorre quando há uma dificuldade de alterar o código devido a sua estrutura. Já a Fragilidade ocorrer quando pequenas mudanças feitas geram bugs em outras áreas do código que não estão diretamente relacionadas

6. Evitar Duplicação

O princípio de Evitar Duplicação quer dizer reduzir a repetição de código, mantendo a DRY (Don't Repeat Yourself) como uma prática chave. Quando esse princípio não é obedecido pode surgir o mau-cheiro “Código Duplicado” descrito por Fowler, em que a presença de código repetido em vários lugares pode levar a dificuldades de manutenção e evolução do sistema.

7. Portabilidade

   O princípio da Portabilidade afirma que é a onde a presença de código repetido em vários lugares pode levar a dificuldades de manutenção e evolução do sistema. Não seguir esse princípio pode resultar no mau-cheiro “Inveja Característica”, ao qual um módulo depende muito de detalhes específicos de outro, tornando a portabilidade difícil.

8. Código Idiomático e Bem documentado

   Por fim temos o princípio do Código Idiomático e Bem documentado ao qual é aquele que segue as convenções e práticas recomendadas da linguagem de programação utilizada, enquanto a boa documentação assegura que outros desenvolvedores possam entender e colaborar no projeto, sem necessariamente a ajuda de algum terceiro para explicar. A não utilização desse princípio pode gerar maus-cheiros como “Nomes Ruins” e “Comentários Supérfluos”.

Questão 2 - Maus-Cheiros do TP2

Para o código NotaFiscal.java:

Maus-cheiros

1. God Class (Classe Deus): A classe “NotaFiscal” possui muitos métodos estáticos que lidam com diferentes responsabilidades, como gerar a nota fiscal, imprimir o cabeçalho, imprimir itens, calcular totais, e imprimir o total a pagar. Isso indica que a classe está assumindo muitas responsabilidades, o que pode torná-la difícil de manter e estender.
2. Long Method (Método Longo): O método “gerarNotaFiscal” realiza várias operações diferentes, como impressão de cabeçalho, cálculo de valores, e impressão de totais. Deixando o código mais complexo e atribuindo muitas funções para um só módulo.

Princípios Violados

Os princípios descritos por Martin Fowler que estão sendo violados neste código podem ser: Modularidade, Simplicidade e Elegância.

Possíveis Alterações

1. **Extrair Classe:** Separar as responsabilidades em diferentes classes, por exemplo, uma classe “ImpressoraNotaFiscal” que lida apenas com a impressão, e uma classe “CalculadoraNotaFiscal” para os cálculos de valores, como “ICMS”, “frete” e “desconto”.
2. **Extrair Método:** Dividir o método “gerarNotaFiscal” em métodos menores e mais focados, como “calcularTotais”, “imprimirDetalhesCliente”.
3. **Mover Método:** Considerar mover os métodos de cálculo (calcularValorTotalItens, calcularICMS, calcularImpostoMunicipal, calcularFrete, calcularDesconto) para a classe “Venda” ou para uma classe de cálculo específica, caso isso se encaixe melhor na modelagem do domínio.

Para o código Venda.java:

Maus-cheiros

1. Feature Envy (Inveja de Característica): O método “calcularFrete” demonstra uma forte dependência da classe Cliente para acessar o endereço, o que pode indicar que essa responsabilidade poderia estar melhor localizada em uma classe relacionada ao Cliente, como “CalculadoraFrete”.
2. Long Method (Método Longo): O método “calcularVendasUltimoMes” é relativamente longo e realiza múltiplas operações, como filtragem de vendas por cliente e data. Isso poderia ser dividido em métodos menores.
3. Divergent Change (Mudança Divergente): A classe “Venda” ainda possui métodos de cálculo que, ao serem modificados (por exemplo, mudanças nas regras de negócios), exigem alterações tanto na classe Venda quanto na “CalculadoraVenda”. Isso pode indicar uma responsabilidade cruzada que ainda persiste.

Princípios Violados

Os princípios descritos por Martin Fowler que estão sendo violados neste código podem ser: Modularidade, Simplicidade e Extensibilidade.

Possíveis Alterações

1. **Mover Método :** O método “calcularFrete” poderia ser movido para a classe “CalculadoraFrete” ou para uma classe relacionada ao Cliente. Resolveria a "inveja de característica" e a fortalecer o encapsulamento.
2. **Extrair Método :** O método “calcularVendasUltimoMes” poderia ser dividido em métodos menores, como “filtrarVendasPorData” e “somarValoresVendas”, para melhorar a legibilidade e manutenção.
3. **Substituir Tipo Primitivo por Objeto :** O uso de métodos que retornam valores numéricos diretamente, como “calcularICMS” e “calcularDesconto”, poderia ser melhorado com a introdução de classes específicas que encapsulam esses cálculos e os dados associados.
4. **Substituir Método por Objeto-Método:** O método “calcularVendasUltimoMes” poderia ser transformado em um objeto-método, encapsulando a lógica complexa e promovendo maior coesão e modularidade.

Para o código CalculadoraVenda.Java:

Maus-cheiros

1. Feature Envy (Inveja de Característica): Os métodos “calcularDesconto”, “calcularICMS”, e “calcularImpostoMunicipal” demonstram uma forte dependência de informações contidas na classe “Cliente”. Isso pode indicar que parte dessa lógica deveria estar na classe “Cliente” ou em subclasses especializadas (ClienteEspecial, ClientePrime).
2. Conditional Complexity (Complexidade Condicional): O método “calcularDesconto” possui diversas condições (if-else) que aumentam a complexidade do código. Isso também é visível em “calcularICMS” e “calcularImpostoMunicipal”, onde o cálculo do imposto é baseado em condições específicas de regiões.

3. Duplicated Code (Código Duplicado): A lógica de cálculo em “calcularICMS” e “calcularImpostoMunicipal” segue um padrão repetido, onde a taxa é multiplicada pelo valor total dos itens. Essa lógica poderia ser encapsulada em um método reutilizável para evitar duplicação.

Princípios Violados

Os princípios descritos por Martin Fowler que estão sendo violados neste código podem ser: Modularidade, Extensibilidade e Elegância.

Possíveis Alterações

1. **Extrair Classe:** Extrair a lógica de cálculo de descontos e impostos para classes específicas, como “CalculadoraDesconto”, “CalculadoraICMS”, e “CalculadoraImpostoMunicipal”, para melhorar a modularidade e facilitar a extensão.
2. **Mover Método:** Mover a lógica condicional baseada no tipo de cliente para as respectivas classes de cliente (ClienteEspecial, ClientePrime). Por exemplo, “ClienteEspecial” poderia ter seu próprio método “calcularDesconto”, eliminando a necessidade de verificações condicionais.
3. **Substituir Condicional com Polimorfismo:** Ao invés de usar condicionais para determinar o tipo de cliente ou região, utilizar polimorfismo para permitir que cada tipo de cliente ou região implemente sua própria lógica de cálculo.
4. **Eliminar Código Duplicado :** Criar um método auxiliar para centralizar o cálculo de taxas multiplicadas pelo valor total dos itens, e reutilizá-lo em “calcularICMS” e “calcularImpostoMunicipal”.
5. **Estratégia de Cálculo:** Implementar o padrão Strategy para encapsular os diferentes métodos de cálculo, permitindo que cada tipo de cálculo (desconto, ICMS, imposto municipal) seja intercambiável e adicionado ou modificado sem impactar outras partes do código.

Para o código CalculadoraFrete.Java:

Maus-cheiros

1. Long Method (Método Longo): O método “calcular” é relativamente longo, com muitas verificações condicionais (if-else), o que torna o código mais difícil de manter e entender.
2. Conditional Complexity (Complexidade Condicional): O uso excessivo de condicionais baseadas em combinações de “localidade” e “regiao” indica que há uma alta complexidade condicional. Esse tipo de código é propenso a erros e dificilmente extensível.
3. Duplicated Code (Código Duplicado): As estruturas if-else repetem a mesma lógica básica para diferentes regiões e localidades, resultando em código duplicado.

Princípios Violados

Os princípios descritos por Martin Fowler que estão sendo violados neste código podem ser: Modularidade, Simplicidade e Extensibilidade.

Possíveis Alterações

1. **Substituir Condicional com Polimorfismo:** Criar classes separadas para cada combinação de “localidade” e “regiao”, cada uma implementando uma interface ou classe abstrata “CalculadoraFrete”. Isso eliminaria a necessidade de condicionais no método calcular.
2. **Tabela de Valores:** Utilizar uma tabela de valores (por exemplo, um Map ou um banco de dados) para armazenar os valores de frete associados a cada combinação de “localidade” e “regiao”. Isso facilitaria a adição ou alteração de valores sem modificar o código.
3. **Estratégia de Cálculo:** Implementar o padrão Strategy para que a lógica de cálculo do frete possa ser facilmente alterada ou estendida sem modificar a classe principal. Isso também pode ajudar a eliminar a complexidade condicional e melhorar a modularidade.

4. **Eliminar Código Duplicado:** Ao mover a lógica de cálculo para classes específicas ou uma tabela de valores, o código duplicado pode ser eliminado, resultando em um código mais limpo e fácil de manter.

Considerações Finais

Esse relatório foi feito pelo grupo 2, composto por:

- Breno Henrique 20/2015984
- Daniel Rocha Oliveira 19/0104821
- Wengel Rodrigues 19/0118555

Seguindo o que foi pedido no enunciado e usando como referência bibliográfica:

- Martin Fowler. Refactoring: Improving the design of Existing Code. Addison-Wesley Professional, 1999.
- Pete Goodliffe. Code Craft: The practice of Writing Excellent Code. No Starch Press, 2006.