TÉCNICAS DE PROGRAMAÇÃO EM PLATAFORMAS EMERGENTES - T01
Semestre: 2024/1
Professor: André Luiz Peron Martins Lanna
Alunos:

- Fellipe Pereira da Costa Silva - 200017772
- Gustavo Barbosa de Oliveira - 180042041
- Cristian Souza Assis Furtado - 170008291

TP3

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.

Simplicidade

Pete Goodliffe, em *Code Craft*, destaca que a simplicidade no código é essencial para clareza e manutenção. Métodos longos, como descritos por Martin Fowler em *Refactoring*, são exemplos de falta de simplicidade, pois acumulam muita lógica em um único lugar, tornando o código complexo e difícil de entender. Refatorar esses métodos em partes menores e focadas é uma forma prática de aplicar a simplicidade, alinhando-se com a visão de Goodliffe de que soluções diretas e claras são sempre preferíveis.

Elegância

A elegância no código, como discutida por Pete Goodliffe, refere-se à criação de soluções que são simples, claras e eficientes. Isso contrasta diretamente com os maus cheiros identificados por Martin Fowler, como "Classe Grande", "Métodos Longos" e "Longa Lista de Parâmetros".

- **Classe Grande**: Falta de elegância devido à acumulação de responsabilidades, tornando o código complexo e difícil de manter. Refatorar para dividir grandes classes em menores, com responsabilidades claras, promove uma estrutura mais elegante.
- **Métodos Longos**: Compromete a clareza e introduz complexidade. Métodos mais curtos e focados, conforme defendido por Goodliffe, resultam em um código mais elegante.
- **Longa Lista de Parâmetros**: Torna o método difícil de usar e entender, prejudicando a simplicidade. Simplificar interfaces e reduzir o número de parâmetros contribui para a elegância do código.

Essencialmente, refatorar esses maus cheiros ajuda a alcançar a elegância no código, que é fundamental para a manutenção e evolução eficiente do software.

Modularidade

A modularidade, conforme discutida por Pete Goodliffe em *Code Craft*, envolve organizar o código em módulos coesos com responsabilidades bem definidas. Isso se relaciona diretamente com os maus cheiros de código "Duplicated Code" e "Switch Statements" de Martin Fowler.

- **Duplicated Code**: A duplicação de código indica uma falta de modularidade, pois significa que partes do código que deveriam estar centralizadas em um único módulo estão espalhadas pelo sistema. Melhorar a modularidade envolve consolidar esse código duplicado em módulos reutilizáveis.
- **Switch Statements**: Declarações de switch frequentemente sugerem uma estrutura de código não modular, onde um bloco de código central decide o comportamento com base em tipos. A modularidade pode ser melhorada substituindo esses switch statements por classes polimórficas, onde cada módulo encapsula seu próprio comportamento.

Resumindo, tanto o "Duplicated Code" quanto os "Switch Statements" indicam áreas onde a modularidade pode ser aprimorada para criar um código mais organizado, coeso e fácil de manter.

Boas Interfaces

As "Boas Interfaces", conforme discutidas por Pete Goodliffe em *Code Craft*, são essenciais para garantir a clareza e a eficiência no design de software. Dois tópicos relacionados a boas interfaces são:

1. **Duplicated Code**: A duplicação de código indica que partes da lógica estão espalhadas pelo sistema, o que pode ser evitado por meio de interfaces bem projetadas que promovem a reutilização. Ao criar interfaces que encapsulam funcionalidades comuns, é possível centralizar a lógica, reduzindo a duplicação e facilitando a manutenção.
2. **Inappropriate Intimacy**: Esse conceito refere-se a uma dependência excessiva entre classes, onde uma classe conhece detalhes internos de outra. Isso sugere que as interfaces entre essas classes não estão bem definidas. Boas interfaces devem promover o encapsulamento e a separação de responsabilidades, garantindo que cada classe opere de forma independente, o que melhora a coesão e a modularidade do sistema.

Em resumo, boas interfaces desempenham um papel fundamental na eliminação de código duplicado e na minimização da intimidade inadequada entre classes, resultando em um design de software mais claro e manutenível.

Extensibilidade

A **extensibilidade** de um sistema refere-se à capacidade do mesmo de ser facilmente adaptado e expandido para novos requisitos que possam surgir ao longo da vida do software. Os code smells de Martin Fowler que afetam a extensibilidade incluem:

- *Rigidez*: Dificulta alterações sem impactar muitas partes do código. Um design flexível é primordial para a extensibilidade.
- *Código Duplicado*: A repetição de código aumenta a complexidade e a dificuldade de manutenção. Evitar duplicação facilita a adição de novas funcionalidades, além de manter organizado o código.
- *Classe Grande*: Classes com muitas responsabilidades são difíceis de modificar e estender. Dividir classes grandes em unidades menores melhora a extensibilidade e até a legibilidade do código.
- *Funções longa*s: Funções longas são complexas e difíceis de alterar. Funções menores e mais focadas são mais fáceis de adaptar, pois se consegue entender o seu fluxo mais facilmente.
- *Acoplamento Alto*: Dependência excessiva entre componentes limita a flexibilidade do sistema. Reduzir o acoplamento facilita a modificação e a extensão dele, pois em caso alguma mudança seja necessária, apenas a parte interessada (ou áreas relacionadas) serão impactadas.
- *Coesão Baixa*: Classes ou métodos com responsabilidades que não se relacionam são difíceis de modificar. Melhorar a coesão ajuda a manter o código organizado e extensível, além de centrar as responsabilidades onde é devido.
- *Dependência Cíclica*: Ciclos de dependência complicam a extensão do código. Resolver dependências cíclicas facilita a modificação e adição de funcionalidades.
- *Código Morto*: Código não utilizado adiciona complexidade desnecessária, sem falar na dificuldade que trechos “mortos” trazem na legibilidade do código. Remover código não utilizado mantém o sistema mais limpo e extensível.
- *Primitive Obsession (Obsessão ‘primitiva’)*: O uso excessivo de tipos primitivos pode limitar a flexibilidade. Usar classes específicas para representar conceitos melhora a extensibilidade, podendo prover métodos para manuseá-los de uma melhor forma.
- *Data Clump (Aglomeração de dados)*: Dados agrupados frequentemente devem ser encapsulados em classes. Isso facilita a extensão e a manutenção.
- *Switch Statements (Uso de switch-case)*: A repetição exagerada do uso de switch-case para diferentes casos pode tornar o código difícil de modificar. Usar padrões de design pode melhorar a extensibilidade.
- *Speculative Generality (Generalidade especulativa)*: Generalizar funções e métodos para requisitos futuros não confirmados adiciona complexidade, ou seja, achar que uma função precise fazer algo mas sem atualmente ser necessário. Manter o código específico e focado para os requisitos atuais facilita a extensão, legibilidade e manutenção do código.

Sem duplicação de código

Evitar duplicação é um princípio fundamental para manter o código limpo e gerenciável. Os *code smells* de Martin Fowler relacionados à duplicação incluem:

- *Código Duplicado*: Quando o mesmo trecho de código aparece em múltiplos lugares, isso pode levar a inconsistências e dificultar a manutenção. A duplicação deve ser evitada para facilitar a atualização e a correção do código.
- *Classe Grande*: Classes grandes podem conter código duplicado, especialmente quando muitas responsabilidades estão acumuladas em um único lugar. Dividir classes grandes e extrair ou apagar código repetitivo para métodos ou classes auxiliares ajuda a reduzir a duplicação.
- *Funções Longas*: Funções longas frequentemente contêm código repetitivo. Dividir funções longas em métodos menores e mais específicos ajuda a evitar duplicação e melhora a legibilidade.
- *Data Clump (Aglomeração de dados)*: Agrupamento de variáveis relacionadas que aparecem juntas em vários lugares pode indicar a necessidade de encapsulamento em uma classe. Criar classes para representar esses dados reduz a duplicação e melhora a organização.
- *Primitive Obsession (Obsessão 'primitiva')*: Uso excessivo de tipos primitivos para representar conceitos complexos pode levar a código duplicado e repetitivo. Substituir tipos primitivos por classes específicas ajuda a encapsular lógica e reduzir duplicação.

Portabilidade

A **portabilidade** refere-se à capacidade do código ser adaptado e executado em diferentes ambientes ou plataformas com facilidade. Embora o conceito de portabilidade não seja especificamente abordado por Martin Fowler, vários *code smells* podem impactar negativamente a portabilidade. Aqui estão alguns deles:

- *Acoplamento Alto*: Componentes fortemente acoplados são difíceis de isolar e adaptar para diferentes ambientes. Reduzir o acoplamento entre módulos facilita a portabilidade, permitindo que partes do sistema sejam movidas ou modificadas sem afetar outras.
- *Código Duplicado*: A duplicação pode criar variações do mesmo código para diferentes plataformas, o que aumenta o risco de inconsistências. Centralizar a lógica comum e evitar duplicação ajuda a manter o código mais portátil e fácil de adaptar a novos contextos.

- *Classe Grande*: Classes grandes podem conter lógica específica para uma plataforma ou ambiente, tornando a adaptação para outros contextos mais difícil. Dividir classes grandes em componentes menores e mais modulares facilita a portabilidade.
- *Data Clump (Aglomeração de dados)*: Dados agrupados frequentemente em locais diferentes podem criar problemas ao tentar adaptar o código para diferentes plataformas. Encapsular dados em classes específicas ajuda a gerenciar e adaptar esses dados de forma mais eficiente.
- *Dependência em Detalhes de Implementação*: Código que depende fortemente de detalhes específicos de uma plataforma ou ambiente pode ser difícil de portar. Criar abstrações e interfaces para isolar esses detalhes melhora a portabilidade do código.
- *Código Morto*: Código não utilizado ou obsoleto pode complicar o processo de adaptação para novos ambientes. Manter o código limpo e remover elementos não utilizados facilita a portabilidade.

Código Idiomático e Bem Documentado

Um **código idiomático e bem documentado** refere-se à prática de escrever código que segue as convenções e padrões da linguagem de programação, além de ser bem documentado para facilitar a compreensão e manutenção por outros desenvolvedores que participam (ou participarão) no desenvolvimento/manutenção do sistema. Relacionando isso com os *code smells* de Martin Fowler:

- *Nomes Ruins*: Nomes inadequados para variáveis, funções ou classes podem tornar o código difícil de entender. O uso de nomes claros e significativos é uma prática de código idiomático que melhora a legibilidade e a documentação implícita.
- *Código Morto*: Código não utilizado ou obsoleto não só contribui para a confusão, mas também torna a documentação mais difícil de manter. Remover código morto mantém o código limpo e a documentação relevante e atualizada.
- *Funções Longas*: Funções longas tendem a ser complexas e difíceis de entender. Dividir funções longas em métodos menores e mais focados, com nomes descritivos, ajuda a manter o código idiomático e melhora a clareza da documentação.
- *Comentários Excessivos*: Comentários excessivos podem ser um sinal de que o código não é claro o suficiente por si só. A documentação deve complementar o código, não substituí-lo. Código bem escrito e idiomático reduz a necessidade de comentários extensivos.
- *Primitive Obsession (Obsessão 'primitiva')*: Usar tipos primitivos para representar conceitos complexos pode resultar em código menos claro. Em vez disso, criar classes específicas para encapsular esses conceitos torna o código mais idiomático e autodescritivo.
- *Data Clump (Aglomeração de dados)*: Dados agrupados em vários locais podem indicar a necessidade de encapsulamento. Criar classes para gerenciar e lidar especificamente com esses dados melhora a clareza do código e facilita a documentação ao agrupar funcionalidades relacionadas.

- *Dependência em Detalhes de Implementação*: Dependência excessiva em detalhes específicos pode tornar o código menos claro e mais difícil de documentar. Usar abstrações e interfaces ajuda a isolar detalhes e melhora a clareza geral do código.

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.

Após a refatoração solicitada no TP2, nosso grupo ainda encontrou os seguintes *code smells* dentro do projeto:

1. **Código duplicado**
    - **Arquivos Afetados:** ClienteEspecial.java, ClientePrime.java
    - **Princípios Violados:** Simplicidade, Elegância
    - **Descrição:** O cálculo do desconto nas classes ClienteEspecial e ClientePrime é similar. Há duplicação na lógica de cálculo de desconto que poderia ser centralizada.
    - **Refatorações Aplicáveis:**
        - **Extração de Método:** Mover a lógica de cálculo de desconto comum para um método em uma classe base ou utilitário.

2. **Métodos longos**
    - **Arquivos Afetados:** ProcessamentoVenda.java (método processarVenda())
    - **Princípios Violados:** Modularidade, Simplicidade
    - **Descrição:** O método processarVenda() realiza várias operações, incluindo cadastro de produtos e clientes, criação de venda e cálculo de valores. Esse método é longo e realiza múltiplas tarefas.
    - **Refatorações Aplicáveis:**
        - **Extração de Método:** Dividir processarVenda() em métodos menores, cada um responsável por uma tarefa específica.

3. **Classe grande**
    - **Arquivos Afetados:** Venda.java
    - **Princípios Violados:** Modularidade
    - **Descrição:** A classe Venda possui muitas responsabilidades, incluindo cálculos complexos e lógica de criação de objetos.
    - **Refatorações Aplicáveis:**

        - **Extração de Classe:** Mover a lógica de cálculo de impostos e frete para classes auxiliares.

4. **Lista de parâmetros longa**
    - **Arquivos Afetados:** Venda.java (método calcularValorTotal())
    - **Princípios Violados:** Extensibilidade
    - **Descrição:** O método calcularValorTotal() tem muitos parâmetros, o que pode indicar que ele está realizando várias tarefas.
    - **Refatorações Aplicáveis:**
        - **Encapsulamento de Dados:** Criar uma classe que encapsule os parâmetros relacionados, como uma classe DetalhesVenda que inclui desconto, frete, ICMS e imposto municipal.

5. **'Intimidade inapropriada'**
    - **Arquivos Afetados:** Venda.java e ProcessamentoVenda.java
    - **Princípios Violados:** Modularidade
    - **Descrição:** A classe Venda possui lógica para cálculo de impostos e frete, que poderia ser movida para classes específicas de cálculo. Além disso, ProcessamentoVenda está muito acoplada aos detalhes de implementação do cadastro de produtos e clientes.
    - **Refatorações Aplicáveis:**
        - **Extração de Classe:** Mover a lógica de cálculos para classes especializadas e criar interfaces para abstrair dependências.

6. **'Inveja de funcionalidade'**
    - **Arquivos Afetados:** Venda.java (métodos de cálculo de desconto, ICMS, frete, etc.)
    - **Princípios Violados:** Boas Interfaces, Simplicidade
    - **Descrição:** A classe Venda tem muitos métodos que dependem de detalhes específicos de Cliente e Produto. Isso pode indicar que a responsabilidade desses cálculos não está bem alocada.
    - **Refatorações Aplicáveis:**
        - **Movimento de Método:** Mover métodos como calcularDesconto(), calcularICMS() e calcularFrete() para as classes Cliente e Produto se esses cálculos forem mais específicos a essas entidades.

7. **Classes de Dados**
    - **Arquivos Afetados:** Produto.java, Cliente.java
    - **Princípios Violados:** Boas Interfaces
    - **Descrição:** As classes Produto e Cliente são basicamente classes de dados com getters e setters, e não têm comportamento significativo além de armazenar dados.
    - **Refatorações Aplicáveis:**
        - **Adicionar Comportamento:** Incorporar comportamentos relevantes às classes para que elas não sejam apenas containers de dados.

Referências bibliográficas

- Martin Fowler. Refactoring: Improving the design of Existing Code. Addison-Wesley Professional, 1999.

Martin Fowler - Refactoring - Improving the Design of Existing-By www.LearnEngineering.in.pdf - Google Drive

- Pete Goodliffe. Code Craft: The practice of Writing Excellent Code. No Starch Press, 2006.

Code Craft The Practice Of Writing Excellent Code : Pete Goodliffe : Free Download, Borrow, and Streaming : Internet Archive