PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Disciplina: Projeto de Software
Professor: João Paulo Carneiro Aramuni
Aluno: João Pedro Santana Marques

Capítulo 9: Refactoring

No desenvolvimento de software, a manutenção é essencial, assim como o teste, para garantir a qualidade e evolução do sistema. Existem três principais tipos de manutenção:

Corretiva: quando corrigimos bugs.
Evolutiva: quando adicionamos novas funcionalidades.
Adaptativa: quando ajustamos o software para mudanças externas, como novas regras de negócio ou tecnologias.

Softwares envelhecem, aumentando sua complexidade ao longo do tempo. Meir Lehman formulou as **Leis da Evolução do Software**: O software precisa ser continuamente mantido e, em certo ponto, pode ser mais vantajoso substituí-lo. Cada manutenção tende a aumentar a complexidade interna do sistema, a menos que haja esforços para controlar essa deterioração. Para evitar essa degradação, uma técnica chamada refactoring (refatoração) é aplicada. Refactoring consiste em modificar o código para torná-lo mais legível, modular e testável, sem alterar seu comportamento funcional. Exemplos incluem renomeação de variáveis, extração de métodos e reorganização da arquitetura.

A **Extração de Método** é um refactoring fundamental que melhora a organização do código ao separar trechos específicos de um método original (f) e movê-los para um novo método (g). O método original passa a chamar o novo método extraído, tornando o código mais modular e legível. Um estudo realizado com desenvolvedores do GitHub identificou 11 principais motivações para o uso da Extração de Método, sendo as mais frequentes: reutilização de código, criação de uma assinatura alternativa, divisão para melhor compreensão, remoção de duplicação e facilitação de novas funcionalidades ou correções.

O **Inline de Métodos** é um refactoring que atua no sentido oposto à Extração de Método. Ele é utilizado quando um método pequeno, com uma ou duas linhas de código, oferece poucos benefícios em termos de reutilização e legibilidade. Nesse caso, o método pode ser removido e seu conteúdo incorporado diretamente nos pontos onde era chamado.

A **Movimentação de Métodos** é um refactoring usado quando um método está na classe errada. Se um método f depende mais de uma classe B do que da sua classe original A, pode ser melhor movê-lo para B. Isso melhora a cohesão da classe A, reduz o acoplamento entre A e B e facilita a manutenção do sistema.

Pull Up Method: Mover um método comum a várias subclasses para uma superclasse, eliminando duplicação. **Push Down Method**: Mover um método de uma superclasse para uma subclasse, se ele for relevante apenas para essa subclasse.

A **Extração de Classes** é um refactoring recomendado quando uma classe A possui muitas responsabilidades e atributos que podem ser agrupados em uma nova classe B. Isso melhora a

organização e a coesão do código. Outro refactoring semelhante é a **Extração de Interfaces**, usada para criar uma interface a partir de métodos comuns em várias classes.

Renomeação é um dos refactorings mais comuns e visa dar nomes mais claros e significativos a elementos do código, como variáveis, métodos e classes. Isso é necessário quando um nome original não é adequado ou quando a responsabilidade do elemento muda ao longo do tempo.

Os refactorings apresentados têm diferentes níveis de impacto no código. Alguns possuem escopo global, como Movimentação de Métodos e Extração de Classes, enquanto outros são de escopo local, melhorando a implementação interna de métodos específicos:

Extração de Variáveis: Simplifica expressões complexas tornando-as mais legíveis e fáceis de entender.
Remoção de Flags: Elimina variáveis de controle desnecessárias, substituindo-as por comandos como return ou break, resultando em um código mais conciso.
Substituição de Condicional por Polimorfismo: Evita comandos switch ao utilizar métodos específicos em subclasses, tornando o código mais modular e fácil de manter.
Remoção de Código Morto: Elimina métodos, classes ou atributos que não são mais utilizados, reduzindo a complexidade do sistema.

A prática de refactoring em projetos de software exige boas suítes de testes, especialmente de unidade. Testes garantem que mudanças estruturais no código, como refactorings, não introduzam erros indesejados. Sem testes, fica mais difícil detectar problemas até que o código entre em produção, quando a correção é mais cara. Existem duas abordagens para realizar refactorings:

Refactorings oportunistas acontecem durante a programação, ao identificar trechos de código que podem ser melhorados enquanto se implementa novas funcionalidades ou se corrigem bugs. Esses refactorings são mais frequentes e devem ser feitos rapidamente, melhorando a estrutura do código à medida que a mudança é realizada.
Refactorings planejados envolvem mudanças mais profundas e complexas, que não devem ser feitas durante outras tarefas. Esses refactorings exigem planejamento, pois podem afetar várias partes do sistema, como a divisão de pacotes ou a correção de uma série de problemas acumulados. Embora importantes, esses refactorings devem ser raros e realizados em períodos específicos.

Code Smells são sinais de que o código tem problemas de qualidade, tornando-o difícil de entender, manter e testar. Eles não devem ser refatorados imediatamente, mas indicam que algo pode ser melhorado dependendo da situação:

Código Duplicado: A duplicação aumenta a complexidade e o esforço de manutenção. Pode ser resolvido com refactorings como Extração de Método, Extração de Classe ou Pull Up Method. Existem quatro tipos de clones de código duplicado, variando de simples diferenças em espaços até alterações em algoritmos.
Métodos Longos: Métodos grandes são difíceis de entender e manter. A prática é escrever métodos curtos (idealmente abaixo de 20 linhas) e, quando necessário, refatorar com Extração de Método.
Classes Grandes: Classes com muitas responsabilidades ou atributos são difíceis de entender e reutilizar. A solução é usar Extração de Classe para dividir classes grandes em menores e mais coesas.

Feature Envy: Ocorre quando um método de uma classe acessa dados ou métodos de outra classe de forma excessiva. A solução é mover o método para a classe que contém os dados desejados.
Métodos com Muitos Parâmetros: Métodos com muitos parâmetros são difíceis de entender. Para melhorar, podemos eliminar parâmetros desnecessários ou agrupar parâmetros em uma nova classe.
Variáveis Globais: Variáveis globais dificultam o entendimento do código, pois seu valor pode ser alterado em qualquer parte do sistema. Elas devem ser evitadas sempre que possível.
Obsessão por Tipos Primitivos: O uso excessivo de tipos primitivos em vez de classes dedicadas pode levar a um código menos seguro e difícil de manter. É preferível criar classes para encapsular valores primitivos.
Objetos Mutáveis: Objetos mutáveis, cujos estados podem ser alterados após sua criação, são problemáticos. A prática é preferir objetos imutáveis, que são mais seguros e fáceis de gerenciar, especialmente em sistemas concorrentes.
Classes de Dados: Classes que só contêm atributos e métodos simples (getters/setters) devem ser repensadas. O comportamento deve ser movido para essas classes, quando necessário, para melhorar a coesão.
Comentários: Comentários não devem ser usados para explicar código ruim. O código deve ser refatorado para melhorar sua clareza. Métodos longos e complexos devem ser quebrados em métodos menores com nomes autoexplicativos, tornando os comentários desnecessários.

Aplicação no Mercado: Remoção de Código Morto em Sistema Legado em Uma Plataforma de E-commerce: Muitas vezes, sistemas antigos acumulam funcionalidades que não são mais usadas, mas permanecem no código. Esse código morto pode incluir métodos, classes ou até mesmo atributos não utilizados.