CAPÍTULO 9 - REFACTORING

O nono capítulo, intitulado Refactoring, introduz o conceito de refatoração, que consiste em um conjunto de transformações no código com o objetivo de melhorar sua manutenção sem modificar seu comportamento. Essas transformações incluem ações como renomear variáveis, dividir funções e reorganizar classes, tornando o sistema mais compreensível, modular e de fácil adaptação. A refatoração é essencial para manter a qualidade e a evolução do software, complementando as manutenções corretivas, adaptativas e evolutivas, garantindo que o código permaneça eficiente para futuras modificações.

O capítulo também apresenta um catálogo de refatorações descrito por Fowler, incluindo:

- **Extração de Método**: Consiste em mover um trecho de código de um método para um novo método separado, facilitando a reutilização e melhorando a legibilidade.
- **Inline de Método**: Ocorre na direção oposta à extração, removendo métodos curtos que não trazem benefícios significativos e incorporando seu conteúdo diretamente onde são chamados.
- **Movimentação de Método**: Permite reorganizar métodos dentro do sistema para que fiquem na classe mais apropriada, contribuindo para uma melhor modularização e arquitetura.
- **Extração de Classes**: Quando uma classe assume muitas responsabilidades, parte de seus atributos e funcionalidades pode ser transferida para uma nova classe, melhorando a organização do código.
- **Renomeação**: Ajusta nomes de variáveis, métodos e classes para torná-los mais intuitivos e alinhados com suas funções, facilitando a compreensão e manutenção do código.
- **Outros refactorings**:
    - **Extração de Variáveis**: Simplifica expressões e melhora a legibilidade.
    - **Remoção de Flags**: Substitui variáveis de controle por comandos mais diretos, como `break` ou `return`.
    - **Substituição de Condicional por Polimorfismo**: Simplifica o uso de estruturas condicionais.
    - **Remoção de Código Morto**: Elimina elementos não utilizados do código, reduzindo sua complexidade.

Após apresentar essas técnicas, o capítulo discute sua aplicação em projetos reais, destacando a importância de testes, especialmente testes unitários, para garantir que as mudanças não introduzam erros. Como as refatorações não adicionam novas funcionalidades, a falta de cobertura de testes pode tornar o processo arriscado.

A refatoração pode ocorrer de duas formas:

- **Oportunista**: Realizada durante a implementação de uma funcionalidade ou correção de um bug, quando um trecho de código pode ser melhorado.
- **Estratégica**: Planejada previamente, abrangendo mudanças mais complexas que exigem um esforço dedicado.

O capítulo também aborda refatorações automatizadas, realizadas por ferramentas das IDEs, que tornam a aplicação de melhorias mais rápida e segura. Nesse processo, o desenvolvedor seleciona o código a ser refatorado e escolhe a transformação desejada, que é aplicada automaticamente pela ferramenta. No entanto, é essencial que o desenvolvedor valide se a alteração atende aos requisitos do sistema.

A seção seguinte trata dos **Code Smells**, que são indicadores de código com baixa qualidade, dificultando sua manutenção e compreensão. Nem todo code smell exige refatoração imediata, mas eles podem sinalizar oportunidades de melhoria. Alguns dos principais exemplos incluem:

- **Código Duplicado**: Aumenta o esforço de manutenção e pode levar a inconsistências.
- **Métodos Longos**: Reduzem a legibilidade e devem ser divididos em métodos menores.
- **Classes Grandes**: Possuem muitos atributos e baixa coesão, dificultando a organização.
- **Feature Envy**: Quando um método acessa mais atributos e métodos de outra classe do que da sua própria.
- **Métodos com Muitos Parâmetros**: Podem ser reduzidos através da criação de estruturas de dados específicas.
- **Variáveis Globais**: Complicam a compreensão do código, pois afetam múltiplas partes do sistema.
- **Obsessão por Tipos Primitivos**: Uso excessivo de tipos simples em vez de objetos mais estruturados.
- **Objetos Mutáveis**: Atribuições e modificações frequentes podem gerar inconsistências; objetos imutáveis são preferíveis.
- **Classes de Dados**: Contêm apenas atributos sem comportamento associado, o que pode indicar a necessidade de agregar funcionalidades a elas.

- **Comentários em Excesso**: Muitas vezes são um indicativo de código mal estruturado. Melhorar a clareza do código pode eliminar a necessidade de explicações extras.

O capítulo conclui ressaltando a importância da refatoração como uma prática contínua para manter o código limpo, eficiente e fácil de evoluir.