O capítulo 9, *Refactoring*, começa introduzindo as refactorings, que são responsáveis por uma série de transformações no código que visam melhorar sua manutenção sem alterar seu comportamento. Essas transformações incluem ações como renomear variáveis, dividir funções ou reorganizar classes, tornando o sistema mais legível, modular e fácil de modificar. A prática de refactoring é crucial para preservar a saúde e a capacidade de evolução de sistemas de software, complementando as manutenções corretivas, adaptativas e evolutivas. Ao seguir essas boas práticas, os desenvolvedores garantem a longevidade e a qualidade dos sistemas, mantendo-os eficientes e preparados para futuras mudanças.

Este capítulo deixa claro também que existe um catálogo de refactorings, apresentado por Fowler, são eles:

- **Extração de Método:** Tem como objetivo extrair um trecho de código de um método e levá-lo para um novo método. Para extrair um método, pode ser necessário fornecer parâmetros ao novo método, especialmente quando ele precisa acessar variáveis locais do método de origem. Nessa seção ainda se trata sobre as motivações para usar a extração de métodos, e chega-se à conclusão que a principal motivação para utilizar esse refactoring, é a reutilização do código.
- **Inline de Método:** Funciona no sentido contrário a uma extração de método. Um método pequeno, com uma ou duas linhas de código e poucas chamadas, oferece pouco benefício em termos de reutilização e legibilidade. Por isso, ele pode ser removido e seu conteúdo incorporado diretamente nos pontos onde é chamado.
- **Movimentação de Método:** É um dos refactorings com maior potencial para melhorar a modularização de um sistema. Por não se limitar a uma única classe, a Movimentação de Métodos pode beneficiar a arquitetura do sistema, assegurando que cada método esteja alocado na classe mais adequada, tanto em termos funcionais quanto arquiteturais.
- **Extração de Classes:** Esse refactoring é indicado quando uma classe A acumula muitas responsabilidades e atributos, alguns dos quais estão relacionados e poderiam formar uma unidade independente. Nesse caso, esses atributos podem ser extraídos para uma nova classe B, que passa a ser referenciada por um atributo na classe A.
- **Renomeação:** Visa tornar o nome do elemento mais claro e significativo. Como nomear elementos de código é uma tarefa difícil, muitas vezes é necessário renomeá-los — seja uma variável, função, método, parâmetro, atributo ou classe. Isso pode acontecer quando

o nome original não foi bem escolhido ou quando a responsabilidade do elemento mudou ao longo do tempo, tornando seu nome desatualizado. Ao realizar esse refactoring, a parte mais desafiadora não é a mudança do nome em si, mas a atualização de todas as referências ao elemento no código.

- **Outros:** Existem refactorings com escopo local, que melhoram, por exemplo, a implementação interna de um único método.
    - Extração de Variáveis: É usado para simplificar expressões e torná-las mais fáceis de ler e entender.
    - Remoção de Flags: É um refactoring que sugere usar comandos como *break* ou *return*, em vez de variáveis de controle, também chamadas de flags.
    - Substituição de Condicional por Polimorfismo: trata da simplificação de comandos condicionais.
    - Remoção de Código Morto: recomenda deletar métodos, classes, variáveis ou atributos que não estão sendo mais usados.

Após apresentar os diversos refactorings, o capítulo busca discutir como essa prática pode ser adotada em projetos reais. Para que os refactorings sejam bem-sucedidos, é fundamental contar com bons testes, especialmente testes de unidade. Sem essa cobertura, qualquer modificação no sistema se torna arriscada, ainda mais quando essas mudanças não adicionam novas funcionalidades nem corrigem bugs, como ocorre nos refactorings.

Outro ponto crucial é definir o momento adequado para realizar a refatoração do código, garantindo que essas melhorias sejam feitas de forma estratégica e segura. Existem dois modos principais de realizar refactorings:

- **Forma oportunista:** Ocorre durante a execução de uma tarefa de programação, ao perceber que um trecho de código está mal implementado e pode ser aprimorado. Isso costuma ocorrer tanto durante a correção de um bug quanto na implementação de uma nova funcionalidade.
- **Forma estratégica:** Geralmente, essas mudanças são mais profundas, demoradas e complexas, tornando impraticável realizá-las durante outra tarefa de desenvolvimento. Por isso, devem ser feitas em sessões planejadas e específicas.

O capítulo 9 também trata sobre refactorings automatizados que consistem em IDEs que oferecem suporte para automatizar a realização de refactorings, facilitando a modificação de

código de forma rápida e segura. Esse processo geralmente segue algumas etapas: o desenvolvedor seleciona o trecho de código a ser refatorado e escolhe a operação desejada. A IDE, então, executa essa operação automaticamente, mantendo a consistência do código.

Apesar de ser chamado de refactoring automatizado, a participação do desenvolvedor ainda é essencial. Ele precisa indicar o código a ser modificado, escolher o tipo de refactoring e fornecer informações necessárias — como o novo nome de um método ou classe. Além disso, o desenvolvedor é responsável por validar se a mudança atende aos requisitos do sistema.

Antes de aplicar o refactoring, a IDE também verifica se as pré-condições estão corretas, garantindo que a modificação não causará erros de compilação ou mudanças inesperadas no comportamento do programa.

Na seção seguinte, o capítulo aborda os Code Smells, sinais de código de baixa qualidade, ou seja, código difícil de manter, entender, modificar ou testar. Em outras palavras, é um código que apresenta indícios de problemas e, por isso, pode ser um candidato à refatoração. No entanto, é importante ressaltar que esses sinais são apenas indicadores, o que significa que nem todo code smell precisa ser imediatamente refatorado. Com isso, alguns dos code smells são:

- **Código Duplicado:** É o principal code smell e aquele com o maior potencial para prejudicar a evolução de um sistema. Responsável por aumentar o esforço de manutenção, pois alterações têm que ser replicadas em mais de uma parte do código. Consequentemente, corre-se o risco de alterar uma parte e esquecer outra.
- **Métodos Longos:** São considerados um code smell, pois eles tornam o código mais difícil de entender e manter. Quando se depara com um método longo, devemos considerar a possibilidade de usar uma Extração de Método para quebrá-lo em métodos menores.
- **Classes Grandes:** Considerado um code smell, pois, assim como métodos longos, elas tornam o código mais difícil de entender e manter. são caracterizadas por um grande número de atributos, com baixa coesão entre eles.
- **Feature Envy:** Esse smell designa um método que parece invejar os dados e métodos de uma outra classe, ele acessa mais atributos e métodos de uma classe do que de sua atual classe.

- **Métodos com Muitos Parâmetros:** É um smell, que pode ser eliminado de duas formas principais. Primeiro, deve-se verificar se um dos parâmetros pode ser obtido diretamente pelo método chamado, outra possibilidade é criar um tipo que agrupe alguns dos parâmetros de um método.
- **Variáveis Globais:** Dificultam o entendimento de um módulo de forma independente dos demais módulos de um sistema.
- **Obsessão por Tipos Primitivos:** Este code smell ocorre quando tipos primitivos (i*nt, float, String*) são usados no lugar de classes.
- **Objetos Mutáveis:** São aqueles cujo estado pode ser modificado após a criação. Por outro lado, objetos imutáveis têm seu estado fixo e não podem ser alterados depois de criados, o que facilita a manutenção e a previsibilidade do código.
- **Classes de Dados:** São classes que geralmente possuem apenas atributos e nenhum comportamento associado. Uma boa prática é analisar o código e avaliar a possibilidade de transferir funcionalidades para essas classes, criando métodos que realizem operações relacionadas aos seus próprios dados, em vez de deixar essas operações dispersas em outras classes.
- **Comentários:** Não devem ser usados como remendo para explicar código mal estruturado. A melhor abordagem é refatorar o código, tornando-o mais claro e legível. Muitas vezes, após essa melhoria, os comentários tornam-se desnecessários, já que a própria qualidade do código comunica sua intenção.