Capítulo 9 – Refactoring

Na primeira seção do capítulo(introdução), o autor nos introduz às Leis de Lehman. A primeira lei nos diz que um software deve ser constantemente mantido para funcionar dentro de seus conformes, até o ponto em que se torna mais vantajoso descontinuá-lo e criar outro. A segunda afirma que, à medida que um sistema sofre manutenções, sua complexidade interna aumenta e sua qualidade vai piorando, a menos que seja realizado um trabalho específico para evitar esse declínio. Tal trabalho se chama refactoring, como conhecemos nos dias atuais. De acordo com o que o autor apresentou, entendi o ato de refatorar um sistema como modificá-lo a fins de melhorar sua performance e manutenibilidade sem comprometer seu funcionamento.

Posteriormente, o autor nos apresenta a um catálogo de refactorings que estão de acordo com o livro de Martin Fowler, uma grande base para o assunto. O primeiro tipo de refactoring escolhido por Marco Tulio é a Extração de Método, que basicamente consiste em transportar um trecho de código de um método X para outro método Z, fazendo com que o primeiro agora chame o segundo durante sua execução. O intuito desta refatoração é eliminar código duplicado e permitir reuso. Ele ainda traz um estudo realizado por ele mesmo, comprovando estas motivações e trazendo mais outras.

Outro refactoring abordado pelo autor é Inline de Método, que consiste em incorporar métodos muito pequenos e que são pouco utilizados no corpo dos métodos que o chamam. Ele ressalta que é um caso mais raro e menos importante que a Extração.

Somos introduzidos também à Movimentação de Método, onde passamos um método presente numa classe X à classe Y. Isso se deve, por exemplo, pois o método, apesar de estar na classe X, utilizava muito mais elementos da classe Y, tornando assim mais coeso transferi-lo para tal classe. É um dos melhores refactoring para melhorar a modularização de um sistema.

Extração de Classes é outro método que ocorre quando, em uma classe específica, temos certos conjuntos de atributos que poderiam ser agrupados em outra classe e terem então vida própria. Exemplo fornecido é uma classe pessoa com atributos como codigoArea e número, que poderiam ser parte de uma classe Fone. Assim, pessoa poderia ter dois atributos de Fone, sendo foneFixo e foneCelular.

Ainda que pareça simples, o autor ainda comenta de outra refactoring denominada Renomeação, onde reatribuímos um nome a uma variável, método, etc. para um melhor significado daquele elemento.

Para concluir o tópico 9.2, somos apresentados a outros refactorings, porém com um escopo mais local, como extração de variáveis (simplifica expressões em variáveis), Remoção de Flags (usar comandos break ou return ao invés de variáveis de controle), Substituição de Condicional por Polimorfismo e a Remoção de Código Morto.

Prosseguindo para o tópico 9.3, o autor aborda como adotar a prática de refactoring. Foi bem frisado que a realização de refactorings depende de testes para garantirem sucesso. Pude absorver dois momentos principais/modos de refatorar, sendo eles uma refactoring oportunista ou planejada. O primeiro tipo se refere às mudanças feitas enquanto estamos delegando uma nova funcionalidade ou corrigindo um bug, por exemplo, e percebemos que "dar um passo para trás" pode resultar em "dois passos à frente". O segundo caso seria para momentos de mudanças mais profundas e que necessitam de mais atenção e tempo.

No próximo tópico somos apresentados a Refactorings Automatizados, como o autor denomina. Seriam refactorings utilizando a ajuda de IDE's e interfaces fornecidas por elas. Não considerei um tópico de alta importância, nos mostrando exemplos e casos específicos de uma IDE e outro exemplo onde o refactoring acaba se tornando uma mudança na funcionalidade de fato, se enquadrando como uma mudança de comportamento e não mais um refactoring.

No último tópico do texto, somos apresentados à parte que julgo a mais importante: casos de atenção, que ajudam o desenvolvedor a observar a necessidade de realizar refatoração: são chamados de "Bad Smells". É o tópico mais extenso do texto. O primeiro Bad Smell seria código duplicado, o com maior potencial para prejudicar a manutenção e evolução de um sistema. Em seguida, métodos longos. Métodos devem ser modulares, ou seja, pequenos e coesos. Ele ainda ressalta que, modernamente, um método deve ter menos de 20 linhas de código para se enquadrar como pequeno. Classes grandes também são um problema, pois muitas vezes acabam delegando funções que vão além de seu escopo inicial. Feature Envy, outro code smell, caracterizado por um método que acaba utilizando mais dados e métodos de outra classe do que daquela que está inserido. Uma solução nesse caso seria a Movimentação de Método.

Chegando à metade do último tópico, nos deparamos com métodos com muitos parâmetros: um smell que pode indicar alto acoplamento, e ainda uma função que pode ser quebrada em outras menores, indicando baixa coesão. Na medida do possível, outro smell a ser evitado, segundo o autor, é o uso de variáveis globais, que podem gerar um tipo de acoplamento ruim. O próximo code smell,

Obsessão por Tipos Primitivos, me fez refletir sobre como o uso excessivo de tipos simples pode tornar o código menos estruturado e mais propenso a erros. Em vez de depender apenas de int, String ou float, faz mais sentido encapsular esses valores em classes que incluam validações e regras de negócio, como um objeto CEP que já garanta a validade do dado. Essa abordagem não só melhora a organização do código, mas também facilita sua manutenção e evita problemas futuros. Além disso, o capítulo sugere que esses objetos sejam imutáveis, reforçando boas práticas de desenvolvimento. Somos introduzidos também a objetos mutáveis: podem causar bugs e problemas em concorrência, enquanto objetos imutáveis são mais seguros e previsíveis. Em Java, por exemplo, String é imutável, evitando alterações indesejadas. Embora em linguagens funcionais isso seja padrão, em linguagens imperativas devemos minimizar o uso de objetos mutáveis, especialmente para classes simples como Data ou Email, garantindo um código mais confiável e seguro. Classes de Dados, segundo o autor, também podem ser indicativos de um problema. Ao possuírem apenas getters e setters e alguns atributos, acabam delegando outros métodos às classes que a implementam. Nem sempre são um erro, mas podem indicar um projeto mal estruturado. O ideal é analisar se comportamentos podem ser incorporados a essas classes, evitando espalhar lógica em outras partes do código. Isso melhora a organização e a coesão, tornando o sistema mais fácil de manter e evoluir. Por fim, temos o último subtópico do capítulo, um de certo modo estranho: comentários. Comentários podem ser considerados um code smell quando usados para explicar código mal escrito. Em vez de depender de comentários, o ideal é refatorar o código para torná-lo mais claro e legível. Métodos longos, por exemplo, podem ser divididos em funções menores com nomes descritivos, eliminando a necessidade de explicações extras. Além disso, o conceito de dívida técnica reforça a importância de manter um código limpo, pois problemas acumulados tornam o sistema mais difícil e caro de manter no longo prazo.

Ao final da leitura de mais um capítulo do livro “Engenharia de Software Moderna”, pude novamente captar e trazer bastante informação útil para minha realidade. Percebi tamanha importância do refactoring, sendo muitas vezes mais necessários do que novas funcionalidades ao sistema. Um exemplo de uso seria ao iniciar um novo cargo como desenvolvedor em uma empresa: ao invés de começar criando novos métodos e funcionalidades, é um bom comportamento buscar compreender com o que irei trabalhar, acompanhar tais estruturas de perto e fazer refactorings que tornarão a minha contribuição futura ao código mais fáceis.