Lucas Felipe Barbosa – Resenha para a disciplina de projeto de software do capítulo 9 do livro “Engenharia de Software Moderna” do autor Marco Tulio Valente

Refactorings são uma série de melhorias realizadas no código afim de melhorar seu entendimento e facilitar futuras manutenções. Existem 3 tipos de mannunteções: a corretiva, que é quando surge um bug, a evolutiva que é quando se é solicitado uma nova funcionalidade e a adaptativa, que é quando alguma regra de negócio é mudada e portanto o código também deve ser mudado.

O primeiro tipo de refactoring é a extração de método. Quando temos algumas linhas de código que estão sendo usadas em mais de um lugar ao mesmo tempo ou até mesmo essas linhas podem ter uma função bem específica, podemos extrair essas partes do método e criar um novo método com essas linhas. Dessa forma teremos várias vantagens, incluindo eliminação da duplicação e melhoria na legibilidade do código, devido ao fato que os métodos menores serão mais específicos e terão nomes que explicam mais precisamente o que fazem.

Indo no caminho contrário da extração de método, as vezes temos uma função que possui apenas 1 ou 2 linhas e que pode, ao invés de estar em um método a parte, ela pode ser incorporada no método em que ela é chamada. Não é porque o método é pequeno que devemos removê-lo e incorporá-lo ao método pai, porém é uma alternativa a se pensar caso.

Movimentação de método é quando um método está em uma classe A, porém ela faz mais sentido que esteja em outra, seja por usar recursos ou até mesmo realizar uma maior quantidade de ações com a classe B do que com a classe A, a qual ela de fato está. Ao ocorrer esse tipo de mudança entre classes de diferentes hierarquias dizemos que está ocorrendo um refactoring do tipo pull up method ou push down method. O primeiro diz respeito a subir um método que é usado por várias classes filhas para uma classe pai, afim de evitar duplicação de código. Quando temos um método na classe pai, mas ele só é de interesse de alguma das filhas, descemos ele para a classe que ele de fato será utilizado.

Se temos em uma classe vários atributos que se juntados podem formar um novo objeto podemos fazer um refactoring de extração de classe, extraindo os atributos dessa classe A para criar uma classe B e depois referenciar não os atributos, mas sim a classe B em A.

Indiscutivelmente uma das partes mais difíceis da programação é nomear classes, atributos e principalmente métodos. Não apenas por querermos que seus nomes condizam bem com o que elas representarão, mas também porque não podemos prever possíveis adições de responsabilidades que tornariam o nome do componente que demos o nome desatualizado e incoerente com o que de fato ele fará. Para tentar contornar isso temos o famoso refactoring de renomeação, que consiste em mudar o nome do método para um que faça mais sentido. A depender de alguns fatores, nem sempre é possível alterar o nome dele e alterar todos os pontos em que ele é chamado,

por isso podemos criar um novo método com o nome refatorado e apenas chamar o método antigo.

Além dos refactoring de escopo global mencionados anteriormente, temos também os de escopo local. O primeiro a ser discutido é a extração de variáveis, que é utilizado para “encaixotar” uma parte do código, uma operação ou até mesmo uma palavra para uma variável e usarmos essa variável no nosso método. Outro tipo é a remoção de flags, que consiste em usar comandos como break ou return ao invés de variáveis de controle para ficar com uma lógica mais clara e dar um fim mais rápido à chamada de função caso seja necessário. Substituição de condicional por polimorfismo é quando temos uma situação em que podemos ao invés de usar uma estrutura condicional como um switch, poderemos criar objetos que contenham a informação que queremos obter de forma mais rápida e sem necessitar de uma estrutura condicional. O último tipo de refactoring mencionado é a remoção de código morto, que apesar de não fazer diferença no sentido de funcionalidade, está lá para ocupar as classes e tornar confusos coisas que não precisariam de serem confusas.

Antes de fazermos o refactoring é necessário analisar com extremo cuidado se as alterações que iremos fazer não irão comprometer a estrutura e as funcionalidades que já existem e já estão funcionando perfeitamente. É necessário analisar com cuidado para evitar situações e possíveis problemas.

No contexto de boas práticas de programação existe um termo chamado “bad smell” que reoresenta possíveis indicadores de um código de baixa qualidade ou de pontos aos quais é necessário se prestar atenção e analisar a viabilidade de refactorings e mudanças.

O primeiro dos bad smells é o código duplicado, que como já mencionado, é crucial evitar. Método e classes longas também podem representar grandes problemas, já que em geral podem assumir muitas responsabilidades e possuir informações que não dizem respeito diretamente ao que fazem de fato.

Feature envy é um método que acessa mais atributos e métodos de outra classe do que a que ele mesmo está. Nesse caso é necessário analisar a viabilidade de realizar a extração desse método para a classe em que ele está se referenciando mais. Métodos com muitos parâmetros também podem indicar problemas já que podem possuir informações duplicadas.

Variáveis globais devem ser evitadas, já que seus valores podem ser alterados em outros lugares no código e influenciar negativamente no resultado de méotdos. Usar tipos primitivos ao invés de classes é um indicativo de possível problema também e deve ser evitado. Sempre que possível é recomendado utilizar objetos imutáveis como strings, já que devido a essa característica eles poderão ser compartilhados de forma livre e segura entre os métodos.

Classes de dados são classes que possuem apenas atributos e nenhum método. Por si só elas não representam uma má prática, porém é necessário analisar onde os comportamentos que dizem respeito a essa classe foram criados e caso seja necessário,

realolocar esses métodos na classe devida. Por fim, temos que comentários podem ser um bad smell. Nem sempre eles indicam que o código é ruim, mas se é necessário que exista um comentário para que um método seja entendido, isso é um grande indicativo que talvez seja necessário refatorá-lo.

Um termos bastante interessante trazido pelo autor do livro e que foi cunhado por Ward Cunningham é “Dívida técnica”. Em um primeiro momento, não parece interessante refatorar um código considerando que iremos perder tempo que poderíamos aproveitar parar implementar novas funcionalidades e corrigir coisas que estão pendentes, porém o que esse termo significa é que os problemas técnicos existentes devido a falta do refactoring levam a uma dívida que é representada pela quantidade de tempo e esforço necessários que se levam para alterar um código quando ele não é refatorado. Por isso, quando os problemas não são resolvidos vai ficando cada vez mais maior essa dívida e enquanto ela não é “paga” os juros (complexidade de manutenibilidade) só aumentam, até que se torne inviável continuar com o atual sistema e seja necessário adotar um sistema completamente novo.