Refatoração

Thauany Moedano[1]

[1]Departamento de Ciência e Tecnologia – Universidade Federal de São Paulo (Unifesp) – São José dos Campos – SP – Brasil

t.moedano@unifesp.br

Abstract. *Refactoring is a continuous process that can be executed in any type of application that uses object-oriented approach. This work has the objective to present an overview of refactoring concepts and how to execute them in projects.*

Resumo. *A refatoração é um processo contínuo que pode ser executado em qualquer tipo de aplicação que utiliza orientação a objeto. Este trabalho tem como objetivo apresentar uma visão geral sobre conceitos da refatoração e como aplicá-la em projetos.*

1. Introdução

Quando se inicia o projeto de um software, não é possível prever o quanto o projeto irá crescer ao longo de anos, décadas ou quanto tempo irá existir. Projetos que crescem muito precisam de constantes atualizações e conforme as tecnologias se atualizam, é cada vez mais difícil dar manutenção a esse tipo de software.

Muitas empresas costumam utilizar a mesma tecnologia durante todo o tempo de existência de sua aplicação, mesmo que essa tecnologia esteja muito ultrapassada. Isso se dá pela grande dificuldade em atualizar o projeto e acoplar novos *frameworks*, sistemas de persistência e os mais diversos tipos de APIs.

A refatoração nesse caso surge como um processo de melhoria. O projeto precisa evoluir e para fazê-lo crescer é necessário reestruturar seus moldes de maneira que sempre seja mais simples o possível dar manutenção.

Este trabalho apresenta alguns conceitos relacionados a refatoração e algumas técnicas de como aplicá-la.

2. O que é refatoração

Refatoração é um processo de melhoria contínua no projeto de um sistema onde a ideia é alterar o código deixando-o mais claro, limpo e fácil de dar manutenção sem que isso afete a experiência do usuário final. [Mens and Tourwé 2004]

A refatoração consiste numa série de conjuntos de técnicas que podem ser aplicá-las para melhorar o código. Algumas delas são: [Fowler 2009]

- Renomear Método: Significa renomear o método para algo mais significativo. A ideia aqui é desviar de abreviações, pois fica difícil entender o que o método faz.
- Mover Método: Mover um método de uma classe para outra, onde faz mais sentido o seu uso. Isso evitar ter que ficar criando muitos objetos e poluindo o código.

- Extrair Método: Consiste em agrupar operações duplicadas no projeto em métodos. Deixa o código mais limpo e claro de entender. Além disso, métodos pequenos são bem mais simples de dar manutenção.
- Internalizar Método: É o processo inverso de Extrair Método. Quando o método é tão pequeno (uma, duas linhas) que não faz sentido existir um método para aquela operação.
- Subir Método: Essa técnica remove métodos de subclasses e as agrega na superclasse. É utilizada quando as subclasses empregam métodos iguais e quebra a regra de generalização-especialização.
- Adicionar Método: Pode estar diretamente relacionado com a técnica de Adicionar Classe. Significa criar novos métodos sobre uma classe afim de melhorar seu funcionamento.
- Extrair Classe: Significa agrupar métodos e atributos de uma classe para criar uma nova classe. Essa abordagem ajuda a deixar as classes mais claras do que fazem. Por exemplo, em uma aplicação escolar, ao invés de uma classe Aluno ter um monte de métodos para tratar de enviar e receber emails, a classe Aluno poderia simplesmente comunicar com a classe Email.
- Adicionar Classe: Essa técnica consiste em criar uma nova classe com o propósito de melhor comunicar as classes existentes. Geralmente é empregada quando deseja-se aplicar padrões de projeto sobre o código.
- Mover Atributo: Assim como Mover Método, realoca um atributo de uma classe para outra.
- Encapsular Atributo: Essa técnica encapsula os acessos a atributos. Geralmente torna-se o acesso aos atributos privados e cria métodos (relacionado a técnica de Adicionar Método) de *getters* e *setters* para acessar um atributo. É uma técnica relacionada a segurança e integridade dos dados.
- Substituir Variável Temporária Por Consulta: Essa técnica refere-se quando em um método você faz operações de expressão e as amarzena e uma variável temporária para guardar esse valor, é possível simplesmente extrair as operações para um método (relacionado ao Extrair Método).

Outras técnicas de refatoração incluem refatorar um conjunto de classes para um padrão de projeto. A seguir algumas técnicas que refatoram para padrões: [Kerievsky 2009].

- Substituir Envio Condicional Por Command: Essa técnica retira do código um conjunto de chamadas condicionais (*ifs..else / switch..case*) pela implementação do padrão de projeto Command.
- Introduzir Objeto Nulo: Retira atribuições de objetos à nulo e cria uma representação nula daquele objeto com o padrão *Null Object*.
- Introduzir Criação Polimórfica com Factory Method: Identifica os polimorfismos existentes no código e os extrai para o padrão Factory Method de criação de objetos.

3. Identificando onde utilizar a refatoração

A refatoração é um processo de melhoria contínua do código mas não é preciso refatorar todo o código sempre que quiser aplicar esse conceito. O processo de identificação de

onde é necessário refatorar é um pouco subjetivo, envolve o conhecimento do próprio programador e das próprias pendências do projeto.

Existem alguns tipos de implementações que podem dar indicativos que o código necessita ser refatorado. Esses conjuntos de códigos são chamados de *Bad Smells*. Identificar *Bad Smells* de forma rápida e correta agilizam o processo de refatoração.

[Beck et al. 1999] elenca 22 tipos de *bad smells* onde [Mantyla et al. 2003] divide esses 22 tipos em 7 classes distintas: *bloaters, abusers, preventers, dispensables, encapsulators e couplers*. A seguir uma breve resumo sobre as classes e os *bad smells* que a compõe.

3.1. Abusers

Os *abusers* são partes do código que não aproveitam da melhor maneira o paradigma de orientação a objeto. A essência de OO são classes e objetos e poluir as classes com um monte de métodos, código duplicado, composições que poderiam ser outras classes indicam um mal uso dos recursos de OO.

Uso demasiado de condicionais switch, subclasses que nada tem a ver com a superclasse, atributos que armazenam variáveis temporárias, classes que fazem exatamente a mesma coisa e alta dependência de heranças são tipos de *bad smells* que se enquadram nessa classe.

3.2. Bloaters

A classe de *bloaters* representa conjuntos de código que cresceram sem seguir nenhum padrão de projeto e sem nenhum tipo de refatoração. Isso gera partes de código que não são possíveis de dar manuntenção e necessitam ser refatoradas no sentido de desacoplar informação. Classes e métodos muito grandes, métodos com uma lista de parâmetros gigante, usar um monte de atributos primitivos quando poderiam ser encapsulados em objetos e código duplicado em várias partes do projeto são exemplos de *bloaters*.

3.3. Couplers

É uma classe de *bad smell* que representa uma alta dependência entre duas classes de tal forma que uma classe viola a estrutura interna e encapsulamento de outra classe. Exemplos de *couplers* são quando uma classe acessa dados de outra classe em demasiado. Ou seja, a classe dona desses atributos e métodos os acessa menos que as outras classes. Outro exemplo são chamadas encadeadas muito longas (por exemplo o aluno chama o método que retorna um email que chama um método que retorna uma mensagem que chama um método que retorna um selo e assim por diante). Isso está diretamente relacionado com classes que apenas delegam operações a outras classes, não fazendo sentido existirem.

3.4. Dispensables

Essa classe engloba partes dos códigos que podem ser removidas por não serem úteis ou que podem ser encapsuladas/englobadas em outras parte do código. Classes sem muita utilidade, atributos, métodos que não são mais utilizados e tudo aquilo que pode ser removido do projeto sem afetar a experiência final do usuário.

Comentários também são indicativos desse tipo de *bad smell* [sou] uma vez que se você tem que fazer um comentário muito grande para explicar o que um método faz ou o que uma classe é, então isso é um indicativo de que o método ou classe podem ser quebrados (removidos e redirecionados) para métodos e classes menores.

3.5. Preventers

Essa última classe engloba conjuntos de códigos onde, se é necessário alterar uma parte do código, também se faz necessário alterar muitas outras partes do código. Isso é ruim em termos de manutenção. Segundo [Beck et al. 1999], mudanças nos código precisam ter o menor encadeamento possível.

Exemplos de *bad smell* desse tipo são quando você altera uma classe e implica em alterar classes que não tem nada a ver com a classe que você alterou.

4. Conclusão

Refatoração é um processo de melhoria e sempre bem vindo. Existem diversas técnicas que podem ser empregadas e que também contemplam o uso de padrões de projeto. Portanto, utilizar padrões de projeto na construção de uma aplicação significa ter que utilizar menos técnicas de refatoração futuramente e dar manutenção ao código se torna bem mais fácil.

Para refatorar partes do código existem diversos tipos de implementações que podem nos dar indicativo de que aquela área do código necessita de refatoração.

Entretanto a refatoração não é milagrosa. Ás vezes o código está tão ruim que é melhor começar do zero.

References

Source making. `https://sourcemaking.com`. acesso em 16/06/2017.

Beck, K., Fowler, M., and Beck, G. (1999). Bad smells in code. *Refactoring: Improving the design of existing code*, pages 75–88.

Fowler, M. (2009). *Refatoração: Aperfeiçoamento e Projeto*. Bookman Editora.

Kerievsky, J. (2009). *Refatoração para padrões*. Bookman Editora.

Mantyla, M., Vanhanen, J., and Lassenius, C. (2003). A taxonomy and an initial empirical study of bad smells in code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 381–384. IEEE.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139.