Análises Estruturais para Identificação de Falso-Positivos em Recomendações de Refatoração

Rafael S. Lima, Ricardo Terra

Departamento de Ciência da Computação,
Universidade Federal de Lavras (UFLA), Brasil

rafaelsplima@computacao.ufla.br,terra@dcc.ufla.br

Resumo. *Desenvolvedores – para atingir objetivos de curto prazo – realizam alterações de código que se opõem à organização estrutural existente. Nesse cenário, ferramentas de identificação de oportunidades de refatoração são normalmente aplicadas no intuito de reverter tais alterações. O problema, entretanto, consiste no fato de que tais ferramentas reportam uma parcela considerável de falsos positivos. Diante disso, este estudo tem o intuito de prover desenvolvedores com uma série de análises que visam explicar o porquê o método deve ser refatorado ou não. No estado atual da pesquisa, foram propostas três análises aplicáveis em refatorações Extract Method e Move Method cujas implementações estão sendo realizadas no Ambiente Moose.*

1. Introdução

Desenvolvedores – para atingir objetivos de curto prazo – realizam alterações de código que se opõem à organização estrutural existente [8]. Para reverter essas alterações, normalmente são aplicadas refatorações que consistem no processo de alteração de um sistema de software sem modificar seu comportamento no intuito de melhorar a estrutura e o entendimento do código [1, 4].

Diversas técnicas vêm sendo propostas na literatura para a identificação de oportunidades de refatoração, por exemplo, baseadas em *Feature Envy bad smells* [9], métricas [2], similaridade estrutural [6, 7], etc. O *problema*, entretanto, consiste no fato de que normalmente essas ferramentas reportam diversas recomendações e que uma parcela relevante consiste de falsos positivos, o que implica em uma baixa precisão.

Diante disso, este estudo – a partir de uma série de recomendações sugeridas por outras ferramentas – tem o intuito de prover desenvolvedores com uma série de análises estruturais para identificação de falsos positivos, visando explicar o porquê o método deve ser refatorado ou não. Por exemplo, assuma que o método m da classe C depende dos tipos $\{A, B\}$ enquanto que todos os outros métodos de C dependem de $\{X, Y\}$. Uma ferramenta baseada em similaridade estrutural recomendaria mover m para uma outra classe C' que seja mais similar.

No entanto, por mais coerente que se pareça, uma análise da dependência em nível de pacotes poderia indicar que A, B, X e Y pertencem ao mesmo pacote p, o que poderia levar a um questionamento por parte do desenvolvedor.

O restante deste artigo está organizado conforme a seguir. A Seção 2 introduz conceitos fundamentais ao estudo. A Seção 3 apresenta a proposta de pesquisa descrevendo um conjunto de análises estruturais no auxílio de refatorações. E, por fim, a Seção 4 conclui descrevendo as contribuições esperadas.

2. *Background*

Similaridade Estrutural: Coeficiente de similaridade mede o grau de correspondência entre duas entidades conforme um critério estabelecido [5]. Assume-se que uma entidade de código fonte (e.g., um método) é representada pelo conjunto de tipos com quais ela estabelece dependência. Desse modo, o cálculo de similaridade entre duas entidades pode ser realizado usando o coeficiente *Jaccard*, que considera a relação existente entre o número de tipos comuns e o número de tipos encontrados em cada entidade, como a seguir:

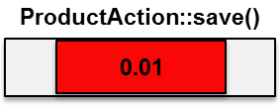
$$sim(E_1, E_2) = \frac{a}{a + b + c}$$

onde $\boldsymbol{a}$ = número de tipos comuns, $\boldsymbol{b}$ = números de tipos exclusivos à entidade E_1, $\boldsymbol{c}$ = números de tipos exclusivos à entidade E_2. Por exemplo, considere dois métodos $m_1 = \{B, C, D\}$ e $m_2 = \{C, D\}$. Como $\boldsymbol{a} = 2$, $\boldsymbol{b} = 1$ e $\boldsymbol{c} = 0$, $sim(m_1, m_2) = 0.666$.

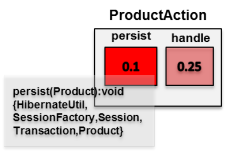
Extract Method: Métodos longos que acumulam diversas responsabilidades – o que implica em baixa coesão e alto acoplamento no nível de métodos – são *bad smells comumente* encontrados em sistemas de software. Por exemplo, assuma um método que realize duas responsabilidades. Nesse cenário, desenvolvedores normalmente aplicam a refatoração *Extract Method* que extrai um fragmento do método original para um novo método. Essa refatoração, além de contribuir diretamente para a modularidade, promove reúso e reduz duplicação de código [1, 4].

Move Method: Métodos implementados em classes incorretas – o que implica em baixa coesão e alto acoplamento no nível de classes – também são *bad smells* comumente encontrados em sistemas de software. Nesse cenário, desenvolvedores normalmente aplicam a refatoração *Move Method* que move o método para uma classe mais apropriada. Essa refatoração diminui o acoplamento e aumenta a coesão em nível de classe, além de promover a organização [1, 4].

Moose: É um ambiente independente de linguagem para engenharia reversa e reengenharia de sistemas de software.[1] Moose provê um conjunto de serviços que incluem um meta-modelo comum, visualização e avaliação de métricas, repositório de modelos e um apoio visual para consulta, navegação e agrupamento [3]. A Figura 1 ilustra um exemplo de duas visualizações do Moose baseadas em similaridade estrutural: (a) uma que reporta oportunidades de refatoração *Extract Method* que pôde identificar um bloco no método `save` que depende de tipos diferentes do restante do método; e (b) uma que reporta oportunidades de refatoração *Move Method* que pôde identificar que o método `persist` depende de tipos diferentes dos demais métodos da classe `ProductAction`.



(a) Rec. de *Extract Method*



(b) Rec. de *Move Method*

Figura 1. Exemplo de visualizações no Moose

[1] http://www.moosetechnology.org

3. Proposta de Pesquisa

O objetivo do presente estudo é prover análises *baseadas em similaridade estrutural* que validem ou invalidem recomendações providas por ferramentas de identificação de oportunidades de refatoração. Conforme ilustrado na Figura 2, logo que uma ferramenta apontar uma oportunidade de refatoração, disponibilizar-se-á uma série de análises que visam explicar o porquê o método deve ser de fato refatorado ou não.

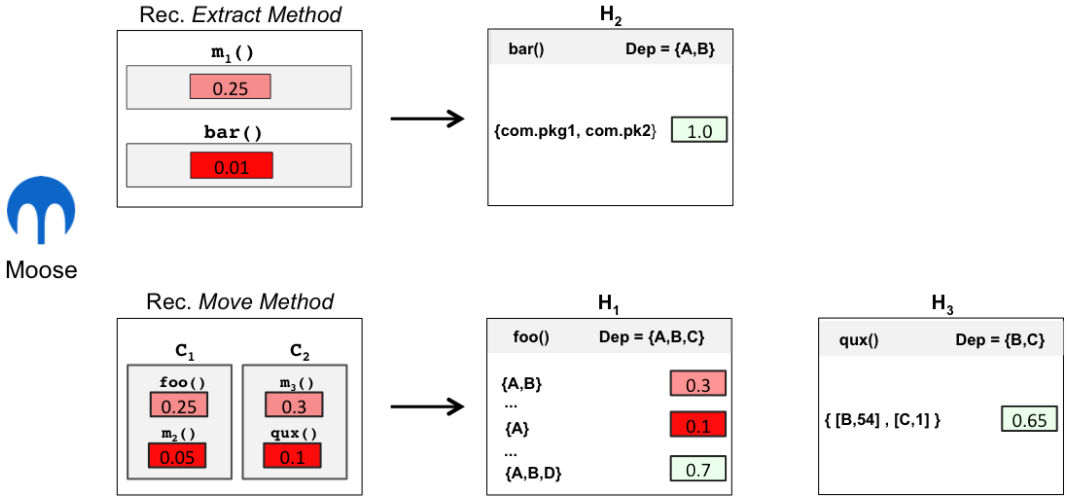


Figura 2. Proposta de Pesquisa

No estado atual desta pesquisa, foram propostas análises – as quais são ilustradas na Figura 2 – com o objetivo de verificar as seguintes hipóteses:

- H_1. *Apenas um ou mais tipos são responsáveis pela baixa similaridade.* Isso indicaria que uma sugestão de refatoração não seria recomendada se o método não dependesse de um certo tipo e/ou fosse adicionado um novo tipo. Por exemplo, o método `foo` depende de $\{A, B, C\}$ e tem similaridade de 0.25 com a classe atual. Se C fosse retirado e D adicionado, a similaridade subiria para 0.7.
- H_2. *Embora os tipos sejam diferentes, a maioria pertence ao mesmo pacote.* Isso indicaria que uma sugestão de refatoração não seria dada se a similaridade fosse calculada no nível de pacotes e não de tipos. Por exemplo, um bloco do método `bar` depende de $\{A, B\}$ e tem similaridade de 0.01 com o restante do método. Se o conjunto considerado fosse o do pacote dos tipos – $\{com.pkg1, com.pkg2\}$ – a similaridade subiria para 1.0.
- H_3. *Os tipos que a entidade depende são largamente (no fator quantitativo) utilizados*. Isso indicaria que uma sugestão de refatoração não seria dada se a similaridade fosse calculada considerando multiplicidade, i.e., o número de vezes que um tipo é acessado. Por exemplo, o método qux depende de $\{B, C\}$ e tem similaridade de 0.1 com a classe atual. Se o conjunto considerasse multiplicidade – $\{[B, 54], [C, 1]\}$ – a similaridade subiria para 0.65.

As análises supracitadas foram elaboradas a partir de justificativas de desenvolvedores para não acatarem certas recomendações. Novas análises serão propostas por investigações qualitativas em repositório de sistemas de código aberto. A avaliação, contudo,

será realizada por meio de questionários. Serão enviadas recomendações que casem com alguma análise proposta, indagando o desenvolvedor se o mesmo a aplicaria ou não com as devidas justificativas, as quais serão utilizadas para aperfeiçoar e corrigir as análises.

4. Considerações Finais

Desenvolvedores – para atingir objetivos de curto prazo – realizam alterações de código que se opõem à organização estrutural existente. Para reverter essas alterações, normalmente são aplicadas refatorações. Nesse cenário, existem diversas ferramentas que identificam oportunidades de refatoração baseadas em *Feature Envy bad smells*, métricas, similaridade estrutural, etc.

A maior contribuição desta pesquisa consiste em prover desenvolvedores com uma série de análises que visam explicar o porquê o método deve ser refatorado ou não. O estudo limita-se a refatorações *Extract Method* e *Move Method* e utiliza ferramentas centradas em similaridade estrutural adaptadas para o ambiente Moose [6, 7]. No entanto, é esperado que, com o decorrer da pesquisa, um maior número de refatorações e ferramentas sejam incorporadas ao estudo.

Agradecimentos

Este trabalho foi apoiado pela FAPEMIG, CAPES e CNPq.

Referências

[1] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, Boston, 1999.

[2] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th International Conference on Software Maintenance (ICSM)*, pages 350–359, 2004.

[3] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gĭrba. The story of Moose: An agile reengineering environment. In *10th European Software Engineering Conference (ESEC)*, pages 1–10, 2005.

[4] William Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[5] H. Charles Romesburg. *Cluster Analysis for Researchers*. Lulu Press, North Carolina, 2005.

[6] Vitor Sales, Ricardo Terra, Luis Fernanda Miranda, and Marco Tulio Valente. Recommending Move Method refactorings using dependency sets. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 232–241, 2013.

[7] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. Recommending automated Extract Method refactorings. In *22nd International Conference on Program Comprehension (ICPC)*, pages 146–156, 2014.

[8] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094, 2009.

[9] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of Move Method refactoring opportunities. *IEEE Transactions on Software Engineering*, 99:347–367, 2009.