A Lightweight Remodularization Process based on Structural Similarity

Ricardo Terra
Department of Computer Science
Federal University of Lavras
UFLA, Brazil
Email: terra@dcc.ufla.br

Marco Tulio Valente
Department of Computer Science
Federal University of Minas Gerais
UFMG, Brazil
Email: mtov@dcc.ufmg.br

Nicolas Anquetil
RMoD Team
Institut National de Recherche en Informatique et Automatique
INRIA, France
Email: nicolas.anquetil@inria.fr

***Abstract*—Erosion process, when neglected over long periods, may reduce the concrete architecture to a small set of strongly-coupled and weakly-cohesive components. This nullifies the benefits provided by an architectural design, such as maintainability, scalability, portability, etc. In this scenario, the solution might be only achieved through a complete remodularization process. However, most remodularization approaches are heavyweight and lack tool support. This paper, therefore, proposes a lightweight and tool supported remodularization process based on structural similarity. The process is centered on the observation that the remodularization of a software system can be achieved by successive applications of Move Class, Move Method, and Extract Method refactorings. We evaluate the process in a modified version of an open-source software achieving a recall of 100% and a precision of 50%. The design decisions of the proposed process—such as the use of structural similarity, non-implementation of preconditions, and identification of the target entity on demand—were also evaluated obtaining outcome results.**

***Index Terms*—remodularization, structural similarity, refactoring.**

I. Introdução

Erosão arquitetural é uma das mais evidentes manifestações de *software aging* [23]. Na prática, quando o processo de erosão arquitetural é negligenciado por longos períodos, ele pode reduzir a arquitetura a um conjunto de componentes com baixa coesão e alto acoplamento [4]. Consequentemente, isso anula os benefícios proporcionados por um projeto arquitetural, tais como manutenibilidade, escalabilidade, portabilidade, etc. Nesse cenário, a solução normalmente empregada consiste na remodularização do sistema [15], [5]. O *problema*, entretanto, consiste no fato de que a maioria dos processos de remodularização são pesados e sem apoio ferramental [15], [16], [5], [28].

Remodularização, por definição, é um processo que altera o projeto modular de um sistema de software para propósitos de adaptação, evolução ou correção. No entanto, esse processo não requer que o comportamento do sistema deva ser alterado. Em várias situações, de fato, é conveniente alterar o projeto modular de um sistema sem alterar seu comportamento externo. Nesse sentido, remodularização é bem similar a uma refatoração de larga escala; conceito esse adotado neste artigo.

Na área de refatoração, existem diversos estudos que identificam oportunidades de refatoração [7], [26], [20], [3], [8], [21], [13]. Marinescu [8] identifica *bad smells* utilizando estratégias de detecção baseadas em métricas, Tsantalis e Chatzigeorgiou [26] identificam refatorações *Move Method* baseadas em *Feature Envy*, um *bad smell* que ocorre quando um método acessa mais dados de um outro objeto do que do próprio. No entanto, uma nova gama de estudos identificam oportunidades de refatoração com base em similaridade estrutural [20], [21], [22]. Em linhas gerais, métodos são movidos para classes mais estruturalmente similares. Por exemplo, assuma o método `Bar::foo` que estabelece dependência estrutural com os tipos $\{\texttt{X}, \texttt{Y}, \texttt{Z}\}$. Se os demais métodos da classe `Bar` não dependerem de tais tipos, recomendar-se-á mover o método `foo` para uma classe mais similar.

Diante desse cenário, este artigo – centrado na observação de que a remodularização de um sistema de software pode ser realizada por sucessivas aplicações de refatorações *Move Class*, *Move Method* e *Extract Method* – propõe um processo de remodularização leve, com apoio ferramental e baseado em similaridade estrutural. O processo é composto de cinco etapas automatizadas: (i) importação do sistema para a plataforma Moose; (ii) identificação de oportunidades de refatoração *Move Class*, (iii) identificação de *Move Method*, (iv) identificação de *Extract Method*, e (v) análise do resultado, podendo retornar às etapas anteriores ou concluir o processo de remodularização.

O processo é dito leve por (i) não requerer a verificação de pré-condições, (ii) a extração das dependências estruturais ser compartilhada na identificação de oportunidades das três refatorações, e (iii) a similaridade estrutural ser calculada apenas entre as entidades e os escopos em que estão contidas, e.g., uma classe com o pacote em que está contida, um método com a classe em que está contido, etc. Ainda, todo o processo possui apoio ferramental, seja na importação para a plataforma de análise ou nas visualizações de identificação de oportunidades de refatoração que exibem os pares classe/-pacote, método/classe e bloco/método com valores baixos de similaridade estrutural considerados *outliers*.

O restante deste artigo está organizado conforme a seguir. A Seção II introduz conceitos básicos ao estudo. A Seção III descreve o processo de remodularização proposto. A Seção IV avalia o processo proposto na remodularização de um sistema de código aberto. A Seção V justifica as principais decisões que tornam o processo de remodularização leve, tais como

978-1-5090-5086-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SBCARS.2016.24

o uso de similaridade estrutural, não implementação de pré-condições, etc. Por fim, a Seção VI descreve trabalhos relacionados e Seção VII conclui.

II. Background

Esta seção introduz conceitos fundamentais ao entendimento do estudo.

Similaridade Estrutural: Coeficiente de similaridade mede o grau de correspondência entre duas entidades conforme um critério estabelecido [19]. Neste artigo, assume-se que entidades do código fonte – classes, métodos e blocos — são representados pelos conjuntos de tipos com os quais estabelecem dependência. Assim, o cálculo da similaridade entre duas entidades (i.e., $sim(E_1, E_2)$ considera: $\boldsymbol{a}$ = número de tipos que ambos as entidades dependem, $\boldsymbol{b}$ = número de tipos que apenas E_1 depende, $\boldsymbol{c}$ = número de tipos que apenas E_2 depende e $\boldsymbol{d}$ = número de tipos que nenhuma depende; essa última variável não é utilizada em diversos coeficientes, o que deixa o cálculo mais leve [19]. *Jaccard* – um dos coeficientes mais utilizados em estudos relacionados [26], [6], [22] – é definido pela Equação 1.

$$sim(E_1, E_2) = \frac{a}{a + b + c} \quad (1)$$

Por exemplo, considere dois métodos em que $Deps(m_1) = \{A, B, C\}$ e $Deps(m_2) = \{B, C\}$. Como $a = 2$, $b = 1$ e $c = 0$, $sim(m_1, m_2) = 2/3 = 0.67$. Caso seja adicionada uma dependência com o tipo D no método m_2 o que implica em $c = 1$, a similaridade cairia para $2/4 = 0.50$.

Move Class: A rudimentar organização de classes em sistemas de software é o *bad smell* mais evidente de sistemas monolíticos, i.e., sistemas sem a definição de uma arquitetura de referência. Por exemplo, assuma uma classe de persistência de dados (*Data Access Object*, DAO) localizada no mesmo pacote de entidades (*Data Transfer Object*, DTO). Nesse cenário, é esperado que no pacote de entidades tenham somente classes representativas de entidades de domínio, como empregados, departamentos, etc. Desenvolvedores, ao observarem tal classe DAO, normalmente aplicam a refatoração *Move Class* que contribui diretamente para organização e a modularidade de sistemas de software [7], [14]. O processo proposto neste artigo recomenda refatorações *Move Class* quando uma particular classe possui baixa similaridade estrutural com as demais classes do mesmo pacote. No cenário do exemplo, enquanto a classe DAO depende de tipos que representam conexão, instruções, conjunto de resultados, etc., classes de entidade dependem de tipos básicos (inteiros, *strings*, datas, etc.) e de outras entidades.

Move Method: Métodos implementados em classes incorretas – o que implica em baixa coesão e alto acoplamento no nível de classes – são *bad smells* comumente encontrados em sistemas de software, especialmente em sistemas mantidos e evoluídos por longos períodos. Por exemplo, assuma um método de persistência que esteja implementado em uma classe na camada de controle. Nesse cenário, é importante que a persistência ocorra em uma classe de modelo. Desenvolvedores, em situações análogas, normalmente aplicam a refatoração *Move Method* que contribui diretamente para a modularidade de sistemas de software [7], [14]. O processo proposto neste artigo recomenda refatorações *Move Method* quando um particular método possui baixa similaridade estrutural com os demais métodos da mesma classe. No cenário do exemplo, enquanto o método de persistência depende de tipos que representam conexão, instruções, conjunto de resultados, etc., os demais métodos da classe de controle dependem puramente de tipos que manipulam requisição e resposta.

Extract Method: Métodos longos que acumulam diversas responsabilidades – o que implica em baixa coesão e alto acoplamento no nível de métodos – também são *bad smells* comumente encontrados em sistemas de software. Por exemplo, assuma um método da camada de controle que manipule os dados recebidos da camada de visão e realize a geração de um relatório. Nesse cenário, é importante, pelo conceito de divisão de responsabilidades, que a configuração do relatório ocorra em um outro método. Desenvolvedores, em situações análogas, normalmente aplicam a refatoração *Extract Method* que, além de contribuir diretamente para a modularidade [7], [14], promove reúso e reduz duplicação de código. O processo proposto neste artigo recomenda refatorações *Extract Method* quando um bloco específico de um método (i.e., um bloco estrutural delimitado por chaves) possui baixa similaridade estrutural com os demais blocos do mesmo método. No cenário do exemplo, enquanto os métodos da classe de controle dependem puramente de tipos que manipulam requisição e resposta, o bloco responsável pela geração do relatório depende de tipos que representam rótulos, campos, *layouts*, etc.

Moose[2]: Uma plataforma independente de linguagem para engenharia reversa e reengenharia de sistemas de software complexos que 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 [12]. O processo proposto neste artigo foi implementado na plataforma Moose, dependendo das bibliotecas `FAMIX`, `Famix-Blocks` e `FASTCore` para a manipulação das dependências e `Roassal2` para a elaboração das visualizações. `VerveineJ`[3] foi a ferramenta utilizada para importar sistemas Java para o meta-modelo adotado no Moose.

III. Um Processo Leve de Remodularização

Esta seção descreve um processo de remodularização proposto na orquestração de três técnicas de identificação de oportunidades de refatoração baseadas em similaridade estrutural. O objetivo é prover à comunidade de arquitetos um processo de remodularização para reverter o processo de

[2]http://www.moosetechnology.org
[3]http://gforge.inria.fr/projects/verveinej

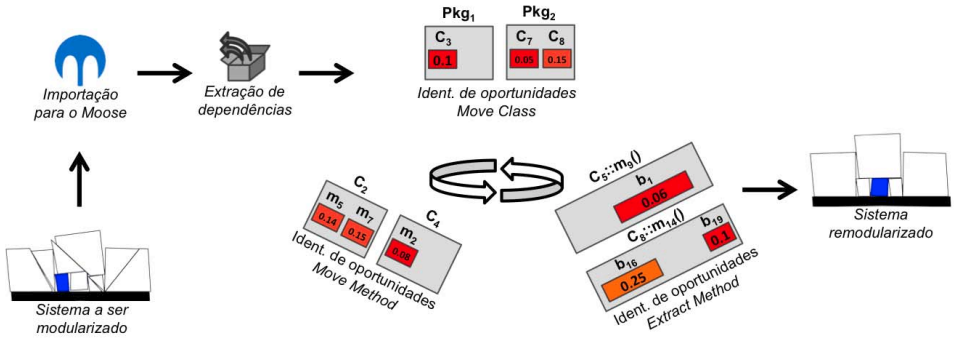

Figura 1. Processo de remodularização proposto[1]

erosão acumulado ao longo dos anos. Como diferencial, o processo proposto é centrado em similaridade estrutural, e projetado para ser o mais leve e automatizado possível.

Conforme ilustrado na Figura 1, o processo é composto de cinco etapas automatizadas: (i) importação do sistema para a plataforma Moose; (ii) identificação de oportunidades de refatoração *Move Class*, (iii) identificação de *Move Method*, (iv) identificação de *Extract Method*, e (v) análise do resultado, podendo retornar às etapas anteriores ou concluir o processo de remodularização ao se atingir a arquitetura desejada. Em seguida, detalha-se cada uma das etapas.

A. Plataforma Moose

Conforme descrito na Seção II, a plataforma Moose requer que o sistema em análise esteja representado em um meta-modelo independente de linguagem de programação. Essa tarefa é trivial e direta, uma vez que a comunidade provê diversas ferramentas que geram esse meta-modelo a partir de código Java, C++, etc. VerneineJ, por exemplo, importa sistemas Java para o meta-modelo adotado no Moose.

A informação dos tipos com os quais uma entidade de código fonte (classe, método ou bloco) estabelece dependência fica disponível na plataforma Moose. Uma vez que essa informação é o núcleo do processo de remodularização proposto, estabeleceu-se duas decisões de projeto:

(i) desconsiderou-se entidades de código fonte que dependam de menos de três tipos. Essas entidades contêm pouca informação para se realizar qualquer inferência baseada em suas dependências estruturais; e

(ii) filtrou-se dependências com tipos comuns (e.g., primitivos, *strings*, etc.) por não contribuírem significativamente na medida de similaridade (analogia com *stop words* de sistemas de recuperação de informação [2]).

B. Recomendações Move Class

Essa etapa recomenda que uma classe C de um pacote Pkg seja movida para um outro pacote quando existir uma baixa similaridade entre C e as demais classes em Pkg. Assuma que o pacote $Pkg1$ possua as classes C_1, C_2 e C_3. Assuma hipoteticamente que a classe C_1 dependa de T_a e T_b, a classe C_2 dependa de T_a, T_b, T_c e T_d, e a classe C_3 dependa de T_d e T_e. Dessa forma, calculada como descrita na Seção II, a similaridade estrutural entre essas classes seria:

$$sim(C_1, C_2) = 2/(2+0+2) = 0.5 \quad (2)$$
$$sim(C_1, C_3) = 0/(0+2+2) = 0.0 \quad (3)$$
$$sim(C_2, C_3) = 1/(1+3+1) = 0.2 \quad (4)$$

Com base na similaridade entre as classes, a similaridade entre uma classe C e um pacote Pkg é calculada pela média aritmética simples entre C e as demais classes de Pkg:

$$sim(C_1, Pkg1) = (0.5+0.0)/2 = 0.25 \quad (5)$$
$$sim(C_2, Pkg1) = (0.5+0.2)/2 = 0.35 \quad (6)$$
$$sim(C_3, Pkg1) = (0.0+0.2)/2 = 0.10 \quad (7)$$

A estabelecimento de um *threshold* que defina uma similaridade como "baixa" é uma tarefa crítica. Como o cálculo da similaridade estrutural é realizado para todo par classe/pacote do sistema, é direta a obtenção da distribuição dos valores de similaridade estrutural. O princípio é que a maioria das classes esteja no local correto, logo o objetivo é encontrar valores discrepantes (*outliers*) na parte baixa da distribuição. Este estudo, portanto, adota um método estatístico para identificação de *outliers* baseado em barreiras internas da distribuição [9].

Assuma uma distribuição hipotética dos valores de similaridade onde o valor do 1^o-quantil ($Q1$) é 0.42 e do 3^o-quantil ($Q3$) é 0.66 e, consequentemente, a distância interquartílica (IQR) é 0.24 ($Q3 - Q1$). Nesse cenário, o *threshold* adotado – i.e., a barreira interna inferior (*lower fence*) – é o resultado da seguinte fórmula:

$$(1^o\text{-}quantil) - IQR = 0.42 - 0.24 = 0.18 \quad (8)$$

Dessa forma, retomando nosso exemplo, a visualização das oportunidades de refatoração *Move Class* exibiria apenas os pares classe/pacote com valores de similaridade estrutural inferiores a 0.18, como é o caso da classe C_3 do pacote $Pkg1$ ilustrado na Figura 1.

É importante mencionar que, nos casos em que a barreira interna inferior (*lower fence*) resultar em um valor inferior a

[3]Para o propósito de facilitar a visualização, as figuras deste artigo estão publicamente disponíveis em: `http://github.com/rterrabh/2016_sbcars`

zero, adotar-se-á o valor do 1^o-quantil ($Q1$) como *threshold*. Isso normalmente ocorrerá quando o sistema já estiver bem modularizado.

Propriedades: A identificação das oportunidades, que é baseada em similaridade estrutural, requer a informação dos tipos, os quais já foram computados e estão disponíveis na plataforma Moose. O processo de recomendação, desde o cálculo de *threshold* e similaridades até a visualização, é completamente automatizado. Essas propriedades também se satisfazem para as demais recomendações do processo (*Move Method* e *Extract Method*).

Mais importante, o custo computacional é baixo, já que não existe verificações de pré-condições e a similaridade é calculada apenas entre as classes de um mesmo pacote, o que resulta em uma complexidade assintótica superior $\mathcal{O}((\frac{|classes|}{|pacotes|})^2)$. A identificação do pacote mais apropriado para uma classe, o que encarece o custo computacional, é realizada apenas sob demanda. Essa identificação ocorre por meio do cálculo da similaridade estrutural da classe em questão com todas as demais classes do sistema. Assim, são retornados os pacotes que possuem maior similaridade estrutural (i.e., média das similaridades das suas classes com a classe em questão).

C. Recomendações Move Method

Essa etapa recomenda que um método m de uma classe C seja movido para uma outra classe quando existir uma baixa similaridade entre m e os demais métodos em C. Assumindo uma classe C_4 com os métodos m_1, m_2, m_3 e m_4, de forma análoga às recomendações *Move Class*, são calculadas as similaridades estruturais entre esses métodos. Em seguida, a similaridade entre, por exemplo, o método m_1 e a classe C_4 seria a média aritmética simples entre m_1 e os demais métodos de C_4 (m_2, m_3 e m_4).

O *threshold* que define uma similaridade como "baixa" é análoga ao adotado no *Move Class*, isto é, a barreira interna inferior (*lower fence*) da distribuição dos valores de similaridade estrutural de todos os pares método/classe do sistema. Dessa forma, assumindo um *threshold* igual a 0.15, a visualização das oportunidades de refatoração *Move Method* exibiria apenas os pares método/classe com valores de similaridade estrutural inferiores a 0.15, como é o caso do método m_2 da classe C_4 ilustrado na Figura 1.

Propriedades: Abordagens do estado-da-arte [20], [26], [3] normalmente possuem alto custo computacional devido às seguintes propriedades: (i) verificarem se é possível realizar as movimentações de métodos de forma automática, o que é desnecessário já que menos de 5% das refatorações são realizadas de forma automática [11]; (ii) verificarem se as movimentações de método trazem melhorias em determinadas métricas, o que pode não ser verdade já que processos de remodularização de sistemas não necessariamente melhoram valores de métricas [1]; ou (iii) verificarem qual a classe de destino mais apropriada, o que é altamente relevante, mas apenas no momento em que o desenvolvedor estiver convencido que a classe deve ser movida; não a priori.

A abordagem do processo proposto foca no baixo custo computacional – i.e., não verifica pré-condições, não se orienta por valores de métricas e calcula a similaridade apenas entre os métodos de uma mesma classe – o que resulta em uma complexidade $\mathcal{O}((\frac{|métodos|}{|classes|})^2)$. Ainda, a identificação da classe de destino mais apropriada, o que encarece o custo computacional, é realizada apenas sob demanda. Essa identificação é realizada de forma similar a como é realizado para *Move Class*. É realizado o cálculo da similaridade estrutural do método em questão com todos os demais métodos do sistema. Assim, são retornadas as classes que possuem maior similaridade estrutural (i.e., média das similaridades dos seus métodos com o método em questão).

D. Recomendações Extract Method

Essa etapa recomenda que um bloco b dentro de método m seja extraído para um novo método quando existir uma baixa similaridade entre b e os demais blocos em m. Blocos, mais especificamente, são blocos estruturais de uma linguagem de programação. Em Java, por exemplo, blocos seriam trechos de código entre chaves. Assumindo um método m_9 com os blocos internos b_1, b_2 e b_3, de forma análoga às recomendações *Move Class* e *Move Method*, são calculadas as similaridades estruturais entre esses blocos. Em seguida, a similaridade entre, por exemplo, o bloco b_1 e o método m_9 seria a média aritmética simples entre b_1 e os demais blocos de m_9 (b_2 e b_3).

O *threshold* que define uma similaridade é novamente a barreira interna inferior (*lower fence*) da distribuição dos valores de similaridade estrutural de todos os pares bloco/método do sistema. Assumindo um *threshold* igual a 0.34, a visualização das oportunidades de refatoração *Extract Method* exibiria apenas os pares bloco/método com valores de similaridade estrutural inferiores a 0.34, como é o caso do bloco b_1 do método m_9 ilustrado na Figura 1.

Propriedades: Abordagens do estado-da-arte [27], [21] normalmente possuem alto custo computacional devido às seguintes propriedades: (i) identificarem o fragmento a ser extraído por meio do cálculo dos *slices* ou por busca exaustiva; ou (ii) verificarem se é possível realizar as movimentações de métodos de forma automática. Este estudo questiona se é praticável investir em verificação de pré-condições já que menos de 9% das refatorações *Extract Method* são realizadas de forma automática [11]. Em consequência, como desenvolvedores realizam refatorações de forma manual, este estudo questiona se a identificação exata do fragmento a ser extraído é de fato necessária ou se bastaria apenas uma identificação aproximada da área que requer refatoração. Nesse contexto, o uso de blocos estruturais parece ser factível para a identificação aproximada do problema, embora se reconheça a necessidade de validação dessa hipótese (trabalho futuro).

A abordagem do processo proposto foca no baixo custo computacional – i.e., não verifica pré-condições e calcula a similaridade apenas entre os blocos de um mesmo método – o que resulta em uma complexidade $\mathcal{O}((\frac{|blocos|}{|métodos|})^2)$.

E. Análise do Resultado

Essa etapa requer a expertise do arquiteto na tomada de decisão se o sistema já atingiu a arquitetura desejada ou se o processo deve retornar às etapas anteriores para iniciar um novo ciclo de remodularização.

IV. Avaliação do Processo Proposto[4]

Esta seção avalia o processo proposto em um sistema de código aberto, o qual foi propositalmente modificado para verificar se o processo é capaz de recomendar refatorações de forma que seja restabelecida sua arquitetura original. Esta seção é organizada como a seguir. A Seção IV-A descreve o sistema alvo, a Seção IV-B executa cada etapa do processo, a Seção IV-C discute os resultados.

A. Sistema Alvo

Para ilustrar a execução do processo proposto, foi utilizado o MyWebMarket, um sistema web de comércio eletrônico previamente utilizado em estudos de remodularização [25], [24]. O sistema gerencia clientes e produtos, realiza vendas, gera relatórios, etc. Mais importante, o sistema foi cuidadosamente projetado e implementado para assemelhar-se, em uma escala menor, a arquitetura de um sistema de gestão de recursos humanos com mais de 200 KLOC [23].

Arquitetura: Figura 2 ilustra a arquitetura do MyWebMarket. A arquitetura segue o bem conhecido padrão arquitetural Modelo-Visão-Controle (MVC) e utiliza *frameworks* largamente utilizados no desenvolvimento de sistemas web (e.g., Hibernate, Struts, JSP, DWR, Quartz, etc.).

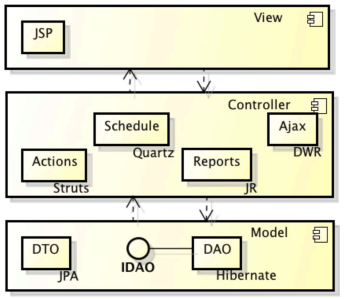

Figura 2. Arquitetura do MyWebMarket

Particularmente neste estudo, é importante destacar os seguintes componentes arquiteturais: (i) *DTOs* que representam entidades de domínio, tais como cliente, pedido, etc.; (ii) *DAOs* que representam implementações de persistência do *framework* Hibernate, como inserir um cliente, cancelar um pedido, etc.; (iii) *Actions* que representam objetos controladores de requisições e respostas do *framework* Struts; e (iv) *Reports* que contêm objetos de configuração e geração de relatórios usando o *framework* JasperReports.

[4] Para fins de replicação, os códigos fontes, modelos, *scripts*, etc. estão publicamente disponíveis em: `http://github.com/rterrabh/2016_sbcars`

B. Processo

Esta seção descreve cada etapa do processo de remodularização proposto. Como MyWebMarket é um artefato resultante de um projeto arquitetural rígido, é esperado que esteja bem modularizado. Diante disso, para verificar a precisão e o revocação do processo proposto, as seguintes modificações foram propositalmente realizadas no sistema antes do início do processo de remodularização: uma classe DAO foi movida para o pacote de entidades (mod. #1); um método de persistência foi movido para uma classe da camada de controle (mod. #2); e um método de geração de relatório foi transferido para dentro de um método da camada de controle (mod. #3).

1) Plataforma Moose: VerneineJ importou o MyWebMarket para o meta-modelo adotado no Moose, além de extrair a informação dos tipos com os quais as entidades de código fonte (classe, método ou bloco) estabelecem dependência.

2) Recomendações Move Class: Essa etapa recomenda que uma classe C de um pacote Pkg seja movida para um outro pacote quando existir uma baixa similaridade entre C e as demais classes em Pkg. A distribuição dos valores de similaridade classe/pacote do sistema resultaram em um 1^o-quantil ($Q1$) de 0.04 e em um 3^o-quantil ($Q3$) de 0.31. Logo, o *threshold* adotado foi 0.04, uma vez que a barreira interna inferior (*lower fence*) resulta em um valor negativo. A Figura 3 ilustra o resultado da execução dessa etapa no MyWebMarket.

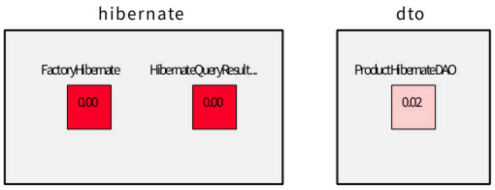

Figura 3. Identificação de Oportunidades *Move Class*

Como esperado, uma das recomendações sugeriu mover a classe `ProductHibernateDAO` localizada no pacote de entidades de volta ao pacote de DAOs (mod. #1). Primeiramente, foram calculadas as similaridades entre `ProductHibernateDAO` e as demais classes do pacote de entidades:

$$sim(ProductHibernateDAO, Customer) = 0.00 \quad (9)$$

$$sim(ProductHibernateDAO, Product) = 0.00 \quad (10)$$

$$sim(ProductHibernateDAO, PurchaseOrder) = 0.00 \quad (11)$$

$$sim(ProductHibernateDAO, PurchaseOrderItem) = 0.04 \quad (12)$$

$$sim(ProductHibernateDAO, User) = 0.06 \quad (13)$$

Por um lado, as classes de entidade normalmente estabelecem dependência com anotações JPA (Java Persistence API), tais como `Entity`, `Id`, `Column`, etc., e com outras entidades. Por outro lado, classes DAO estabelecem dependência com tipos de persistência do *framework* Hibernate, tais como

`Session`, `Transaction`, `Criteria`, etc., e também com entidades. Dessa forma, `ProductHibernateDAO` teve interseção apenas com `PurchaseOrderItem` e `Product` devido a todos estabelecerem dependência com o tipo `Product`. Por fim, a similaridade entre `ProductHibernateDAO` e o pacote `dto` é calculada como a seguir:

$$sim(_{ProductHibernateDAO,dto}) = {}_{(0.0+0.0+0.04+0.0+0.06)/5} = 0.02 \quad (14)$$

É importante notar que essa etapa recomendou a movimentação de duas outras classes – `FactoryHibernate` e `HibernateQueryResultDataSource` – as quais, por meio de uma análise do primeiro autor deste artigo, não deveriam estar localizadas nesse pacote. Essas classes, além de não terem relação estrutural, possuem propósitos e objetivos distintos das demais classes do pacote. Isso indica, portanto, que deveriam ser movidas para um pacote mais apropriado.

3) Recomendações Move Method: Essa etapa recomenda que um método m de uma classe C seja movido para uma outra classe quando existir uma baixa similaridade entre m e os demais métodos em C. A distribuição dos valores de similaridade método/classe do sistema resultaram em um 1^o-quantil ($Q1$) de 0.50 e em um 3^o-quantil ($Q3$) de 0.86. Nesse cenário, o *threshold* adotado foi 0.14, o qual representa a barreira interna inferior (*lower fence*). A Figura 4 ilustra o resultado da execução dessa etapa no MyWebMarket.

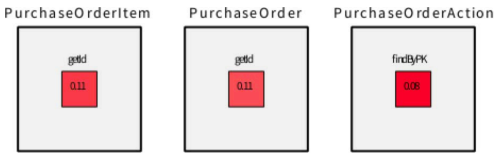

Figura 4. Identificação de Oportunidades *Move Method*

Como esperado, uma das recomendações sugeriu mover o método `findByPk` localizada em uma classe controladora de volta à classe de DAOs (mod. #2). Primeiramente, foram calculadas as similaridades entre `findByPk` e os demais métodos da classe controladora:

$$sim(_{findByPk,saveAction}) = 0.07 \quad (15)$$

$$sim(_{findByPk,validateForm}) = 0.09 \quad (16)$$

$$sim(_{findByPk,updateAction}) = 0.07 \quad (17)$$

$$sim(_{findByPk,inputAction}) = 0.06 \quad (18)$$

$$sim(_{findByPk,deleteAction}) = 0.08 \quad (19)$$

$$sim(_{findByPk,editAction}) = 0.09 \quad (20)$$

$$sim(_{findByPk,findAction}) = 0.08 \quad (21)$$

Por um lado, métodos controladores normalmente estabelecem dependência com classes de auditoria, constantes, entidades e interfaces de persistência. Por outro lado, métodos DAO estabelecem dependência com tipos de persistência do *framework* Hibernate, tais como `Session`, `Transaction`, etc., e também com entidades. Dessa forma, as únicas interseções entre `findByPk` e os demais métodos ocorreram devido a dependência com a classe de entidade `PurchaseOrder`.

Por fim, a similaridade entre `findByPk` e a classe `PurchaseOrderAction` é calculada como a seguir:

$$sim(_{findByPk,PurchaseOrderAction}) = {}_{(0.07+0.09+0.07+0.06+0.08+0.09+0.08)/7} = 0.08 \quad (22)$$

É importante notar que essa etapa recomendou a movimentação de dois métodos `getId`. Embora esses métodos estão corretamente localizados, eles possuem diversas dependências com tipos JPA específicos para chaves primárias – e.g., `Id`, `GenericGenerator`, `GeneratedValue` – o que implica em uma baixa similaridade com os demais métodos. Esse cenário fomenta o estudo de heurística para evitar falsos positivos (trabalho futuro).

4) Recomendações Extract Method*:* Essa etapa recomenda que um bloco b dentro de método m seja extraído para um novo método quando existir uma baixa similaridade entre b e os demais blocos em m. A distribuição dos valores de similaridade bloco/método do sistema resultaram em um 1^o-quantil ($Q1$) de 0.00 e em um 3^o-quantil ($Q3$) de 0.02. Logo, o *threshold* adotado foi 0.00, uma vez que a barreira interna inferior (*lower fence*) resulta em um valor negativo. Como MyWebMarket é um sistema de pequeno porte, apenas três métodos tiveram blocos suficientemente grandes (com mais de três dependências), o que justifica os baixos valores de similaridade. A Figura 5 ilustra o resultado da execução dessa etapa no MyWebMarket.

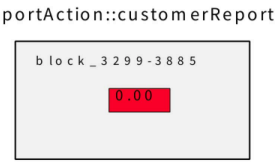

Figura 5. Identificação de Oportunidades *Extract Method*

Como esperado, foi recomendado extrair o bloco responsável pela geração de um relatório para um novo método (mod. #3). Primeiramente, foram calculadas as similaridades entre esse bloco e os demais blocos do método (que é apenas um neste caso):

$$sim(_{block\_3299-3885,main\_block\_customerReport}) = 0.00 \quad (23)$$

Por um lado, blocos de métodos controladores normalmente estabelecem dependência com classes de auditoria, constantes, entidades e interfaces de persistência. Por outro lado, blocos de métodos de geração de relatório estabelecem dependência com tipos do *framework* JasperReport, tais como `JasperReport`, `JasperFillManager`, etc. Dessa forma, como não houve interseção entre os blocos, a similaridade entre a similaridade entre esse bloco e o método `customerReport` é calculada como a seguir:

$$sim(_{block\_3299-3885,customerReport}) = {}_{(0.00)/1} = 0.00 \quad (24)$$

5) Análise do Resultado: Todas as modificações que foram intencionalmente incorporadas foram recomendadas pelo processo proposto. No entanto, a modificação #3 foi parcialmente solucionada. O método de geração de relatório foi de fato extraído para um novo método, porém esse método não pertencia a uma classe de controle. Dessa forma, a etapa (iii) foi novamente realizada na versão do MyWebMarket já com as refatorações recomendadas aplicadas, cujo resultado é ilustrado na Figura 6.

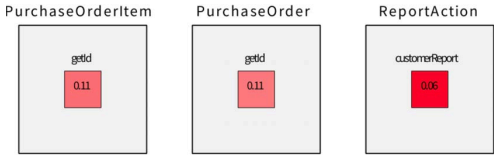

Figura 6. Identificação de Oportunidades *Move Method* - 2^o ciclo

Como esperado, uma das recomendações sugeriu mover o método `customerReport` – extraído no ciclo anterior do processo – localizada em uma classe controladora para uma classe do componente *Reports*. Observe que os métodos `getId` continuam sendo recomendados. Isso implica que um mecanismo deve ser desenvolvido para evitar que o processo seja redundante em suas recomendações (trabalho futuro).

C. Discussão

O processo proposto foi avaliado em uma versão modificada de um sistema de código aberto onde três modificações foram propositalmente realizadas. Esta seção reporta discussões relacionadas à revocação e à precisão do processo.

Revocação: A execução de um ciclo completo complementado por uma etapa adicional foram necessárias para reestabelecer a arquitetura original do sistema, o que ressalta uma revocação de 100%. Além disso, a abordagem também identificou corretamente a classe de destino – solicitada sob demanda – para todas as recomendações *Move Method* e *Move Class*.

Precisão: De um total de 10 recomendações, 4 estiveram corretas, 2 potencialmente corretas (mas não presentes no oráculo) e 2 erradas (que se repetiram em duas etapas). Dessa forma, ignorando duplicatas, o processo proposto alcançou uma precisão de 50% (4/8) com base no oráculo, mas que, considerando a prévia análise realizada pelo primeiro autor deste artigo, seria de 75% (6/8).

V. Investigação Empírica[4]

Por meio da análise de instâncias reais de refatoração, esta investigação tem o intuito de prover argumentos que validem que *a remodularização de um sistema de software pode ser sim guiada por dependências estruturais*, além de avaliar as decisões de projeto do processo proposto, como considerar apenas métodos com pelo menos três dependências, não verificar pré-condições, etc.

A. Dataset

Um *dataset* de 36 instâncias reais de refatorações *Move Method*, classificadas por outros pesquisadores, foi utilizada [11], [17].[5] A Tabela I reporta os sistemas, as respectivas versões ou número do *commit*, e o número de instâncias.

Tabela I
Dataset DE INSTÂNCIAS REAIS DE *Move Method*

Fonte	Sistema	Versão	# de instâncias
[17]	Ant	1.2 / 1.3	2
		1.3 / 1.4	2
	JMeter	1.7.3 / 1.8	2
		1.8 / 1.8.1	1
		1.8.1 / 1.9_RC1	7
		1.9_RC1 / 1.9_RC2	1
	XMLSec [17]	1.0.4 / 1.0.5D2	12
[11]	Eclipse	1.856	4
		2.444	1
	Mylyn	5.587	3
		6.267	1
Total			36

B. Metodologia

O estudo empírico foi conduzido por meio de análises quantitativas e qualitativas:

Análise Quantitativa: Para cada instância do *dataset*, é calculada a similaridade estrutural de um método m na classe de origem C' e na classe de destino C'' a fim de se aplicar métodos estatísticos para se obter maior confiabilidade nos resultados.

Análise Qualitativa: É realizada uma análise qualitativa nas instâncias em que não houve melhorias na similaridade estrutural a fim de se levantar potenciais melhorias no processo.

C. Análise Quantitativa

A Tabela II reporta os resultados de cada instância do *dataset*. Cada linha possui o sistema, o método, a classe origem (versão anterior) e a classe destino (versão posterior). Mais importante, a tabela reporta a diferença do valor da similaridade estrutural do método antes e depois da refatoração. Por exemplo, a instância #1, método `getOriginId` teve uma variação de 0.1605 (melhoria) no valor da sua similaridade estrutural após a refatoração. Como um outro exemplo, a instância #31, método `isValidRmiRemote` teve uma variação de -0.1288 (piora) no valor da sua similaridade estrutural após a refatoração.

Como informação complementar, a tabela também reporta a diferença do valor da similaridade estrutural da classe de onde o método foi movido (espera-se que melhore após a sua remoção) e para onde ele foi movido (também espera-se que melhore após a sua inserção). Por exemplo, na instância #28, após mover o método `sendMail`, a classe `MailerResultCollector` (onde o método estava)

[5]Uma análise manual removeu 28 refatorações do *dataset* original por métodos acessores comuns (24 instâncias), má interpretação de refatoração (2), propagação de alteração de código (1) e remoção de código duplicado (1).

Tabela II
RESULTADO DA INVESTIGAÇÃO EMPÍRICA

#	Sistema	Método	Classe de Origem	Classe de Destino	Diferença Similaridade			Ref. Automática?	Modificações?
					Método	Classe Origem	Classe Destino		
1	Mylyn	getOriginId	ActivityContextManager	AbstractUserActivityMonitor	0.1605	0.1129	nova classe	sim	não
2	Mylyn	getStructureKind	ActivityContextManager	AbstractUserActivityMonitor	0.1605	0.1129	nova classe	sim	não
3	JMeter	createTestElement	ModifyControllerGui	ModuleControllerGui	0.1462	classe removida	nova classe	não	sim, poucas
4	Mylyn	getStructureHandle	ActivityContextManager	AbstractUserActivityMonitor	0.1268	0.1129	nova classe	sim	não
5	Ant	isValidRmiRemote	Rmic#RmicFileNameMapper	Rmic	0.0711	classe removida	0.0112	não	sim, várias
6	XMLSec	canonicalizeSubTree	Canonicalizer20010315	CanonicalizerBase	0.0710	−0.0217	nova classe	não	sim, várias
7	Eclipse	hasSuppressWarningsProposal	ModifierCorrectionSubProcessor	SuppressWarningsSubProcessor	0.0677	0.0729	nova classe	não	não
8	XMLSec	outputTextToWriter	Canonicalizer20010315	CanonicalizerBase	0.0513	−0.0217	nova classe	não	sim, poucas
9	XMLSec	outputAttrToWriter	Canonicalizer20010315	CanonicalizerBase	0.0513	−0.0217	nova classe	não	não
10	Eclipse	isParameter	RenameTypeProcessor	JavaModelUtil	0.0505	0.0002	−0.0008	não	sim, poucas
11	Eclipse	addSuppressWarningsProposals	ModifierCorrectionSubProcessor	SuppressWarningsSubProcessor	0.0480	0.0729	nova classe	não	não
12	XMLSec	outputPItoWriter	Canonicalizer20010315	CanonicalizerBase	0.0477	−0.0217	nova classe	não	sim, poucas
13	XMLSec	outputCommentToWriter	Canonicalizer20010315	CanonicalizerBase	0.0477	−0.0217	nova classe	não	sim, poucas
14	XMLSec	getPositionRelativeToDocumentElement	Canonicalizer20010315	CanonicalizerBase	0.0435	−0.0217	nova classe	não	não
15	Mylyn	isValidURL	BugzillaRepositorySettingsPage	BugzillaClient	0.0394	0.0036	−0.0076	não	sim, poucas
16	XMLSec	canonicalizeXPathnodeSet	Canonicalizer20010315	CanonicalizerBase	0.0389	−0.0217	nova classe	não	sim, várias
17	XMLSec	getPositionRelativeToDocumentElement	Canonicalizer20010315Excl	CanonicalizerBase	0.0359	0.0374	nova classe	não	não
18	XMLSec	outputTextToWriter	Canonicalizer20010315Excl	CanonicalizerBase	0.0342	0.0374	nova classe	não	sim, poucas
19	XMLSec	outputAttrToWriter	Canonicalizer20010315Excl	CanonicalizerBase	0.0342	0.0374	nova classe	não	não
20	XMLSec	outputCommentToWriter	Canonicalizer20010315Excl	CanonicalizerBase	0.0307	0.0374	nova classe	não	sim, poucas
21	XMLSec	outputPItoWriter	Canonicalizer20010315Excl	CanonicalizerBase	0.0307	0.0374	nova classe	não	sim, poucas
22	JMeter	sendMail	MailerVisualizer	MailerModel	0.0255	classe removida	nova classe	não	sim, poucas
23	Eclipse	getFirstFragmentName	ModifierCorrectionSubProcessor	SuppressWarningsSubProcessor	0.0248	0.0729	nova classe	não	não
24	Ant	javaToClass	BatchTest#FileList	optional.junit.BatchTest	0.0229	classe removida	0.1579	não	sim, poucas
25	JMeter	focusLost	TextAreaTableCellEditor	TextAreaTableCellEditor#EditorDeleg	0.0192	0.1592	nova classe	não	não
26	JMeter	keyReleased	ConstantTimerGui	ConstantThroughputTimerGui	0.0029	0.0133	nova classe	não	sim, poucas
27	Ant	createTargetfile	Transform	ExecuteOn	0.0013	classe removida	−0.0155	não	não
28	JMeter	sendMail	MailerResultCollector	MailerModel	−0.0023	0.3641	−0.0062	sim	sim, poucas
29	Eclipse	addSuppressWarningsProposal	ModifierCorrectionSubProcessor	SuppressWarningsSubProcessor	−0.0133	0.0729	nova classe	não	não
30	JMeter	getStaticLabel	UrlConfigGui	HttpDefaultsGui	−0.0306	0.0514	nova classe	não	não
31	Ant	isValidRmiRemote	Rmic	Rmic#RmicFileNameMapper	−0.1288	−0.0419	nova classe	não	não

teve uma melhora de 0.3641 no valor da similaridade média de seus métodos. Ainda, a tabela reporta se a instância era passível de refatoração automática e se houve modificações no código fonte durante a refatoração.

No intuito de se comprovar que os valores das similaridades estruturais dos métodos antes e depois das refatorações são estatisticamente diferentes, o teste-t foi aplicado sobre o valor da similaridade estrutural do método antes e depois da refatoração (coluna 6 da Tabela II). Conforme reportado na Tabela III, pode-se garantir com 99.9% de confiança a melhoria estatisticamente relevante na similaridade estrutural dos métodos após as refatorações.

Tabela III
TESTE-t SOBRE OS VALORES DAS SIMILARIDADES ESTRUTURAIS DOS MÉTODOS ANTES E DEPOIS DAS REFATORAÇÕES

Média Aritmética ($\overline{x}$) = 0.0422
Desvio Padrão (σ) = 0.0558
Intervalo de Confiança = $\overline{x} \pm t\sqrt{(\sigma^2/n)} = 0.0422 \pm t\sqrt{(0.0031/31)} = 0.0422 \pm t(0.01)$
O 0.9995-quantil da variável t com 30 graus de liberdade é 3.646.
Intervalo com confiança de 99.9% = $0.0422 \pm (3.646)(0.01) = (0.0057, 0.0787)$

É importante mencionar que cinco métodos foram desconsiderados por serem *outliers*. Esses métodos foram movidos (i) de uma classe interna que continha apenas tal método ou (ii) para uma classe interna que passou a conter apenas tal método.

D. Análise Qualitativa

A Tabela IV reporta o resultado da classificação dos métodos em relação a variação da similaridade estrutural. Os métodos que tiveram diferenças na similaridade estrutural superiores a +2% foram classificados como *Condizentes à hipótese*, inferiores a -2% como *Relutantes à hipótese*, e entre -2% e +2% foram considerados como *Indiferentes à hipótese*.

Tabela IV
POSICIONAMENTO DOS RESULTADOS FRENTE À HIPÓTESE

Classificação		**# de instâncias**
Condizentes à hipótese	($\geq$0.02)	24
Indiferentes à hipótese	(entre -0.02 e 0.02)	5
Relutantes à hipótese	(≤ -0.02)	2
Total		**31**

Métodos relutantes à hipótese: Apenas dois métodos foram relutantes a hipótese. O método `isValidRMIRemote` foi movido da classe `Rmic` para a classe interna `RmicFileNameMapper` entre as versões 1.2 e 1.3 do sistema Ant. Antes da refatoração, a similaridade do método era de 0.2315, caindo para 0.1027 após a refatoração. No entanto, esse mesmo método foi movido de volta à classe `Rmic` na versão 1.4; fato que reforça a hipótese.

Já o método `getStaticLabel` foi movido da classe `UrlConfigGui` para `HttpDefaultsGui` entre as versões 1.7.3 e 1.8 do sistema JMeter. Antes da refatoração, a similaridade do método era de 0.0885, caindo para 0.0579 após a refatoração. Esse método é pequeno e possui apenas uma dependência (classe `URL`), o que reforça a decisão de projeto de o processo de remodularização proposto considerar apenas entidades de código fonte que dependam de pelo menos três tipos. Este estudo alega que, no cenário de similaridade estrutural, quanto maior o conjunto de dependências de uma entidade de código fonte, mais efetivo é o resultado.

Métodos indiferentes à hipótese: Os cinco métodos que foram indiferentes à hipótese foram analisados e foram vislumbradas as seguintes motivações para as refatorações: (i) decomposição de classe, onde um método não foi movido, mas sim, uma classe se dividiu; (ii) cenário do método comum, onde um método com um nome comum foi removido em uma classe e adicionado em outra; (iii) composição de classes, onde duas classes se tornaram apenas uma; (iv) código duplicado; e (v) orientação semântica, onde, embora sem similaridade estrutural, o método tem relação semântica com os demais. No entanto, este estudo não reivindica que similaridade estrutural é a única razão para a aplicação de refatorações *Move Method*.

E. Discussão

Esta seção reporta discussões relacionadas às evidências que suportem o uso de similaridade estrutural e que justifiquem as decisões de projeto tomadas para o processo de remodularização proposto.

Similaridade Estrutural: (i) 77% dos métodos apresentaram melhoria na similaridade estrutural após a refatoração, logo este estudo não descarta outros fatores que motivem refatoração, e.g., código duplicado; e (ii) 75% dos métodos foram movidos para novas classes, o que justifica a decisão de projeto de identificar a entidade de destino sob demanda.

Pré-condições: apenas quatro refatorações (11%) poderiam ser aplicadas de forma totalmente automática e pelo menos 47% dos métodos sofreram alguma forma de modificação, o que justifica a decisão de projeto de não realizar a refatoração automática, mas sim, *recomendar* refatorações.

VI. TRABALHOS RELACIONADOS

Esta seção descreve relevantes soluções de remodularização do estado da arte.

Rama e Patel analisaram vários esforços de remodularização para formalizaram seis operadores de modularização recorrentes [18] – união/decomposição de módulos, transferências de dado/função/arquivo e promoção de função para API. Os autores afirmam que operadores podem ser continuamente aplicados durante a evolução de um sistema de software, assim como a ideia do processo iterativo proposto neste artigo. Em contraste, tal solução não fornece apoio ferramental.

Clustering hierárquico é uma outra técnica para avaliar decomposições alternativas de sistemas de software [10]. No entanto, Anquetil et al. conduziram um estudo de caso no Eclipse que reporta que reestruturações não necessariamente melhoram a modularidade em termos de coesão/acoplamento [1]. O

processo proposto neste artigo, por sua vez, é centrado em similaridade estrutural e não por métricas de qualidade.

Moghadam et al. propuseram uma abordagem de remodularização que refatora automaticamente o código fonte de um sistema em direção a um diagrama de classes UML [13]. A abordagem compara o modelo atual com o desejado e expressa as diferenças como uma série de refatorações. O processo proposto neste artigo também se baseia na aplicação sucessiva de refatorações, embora não requer um modelo alvo por ser orientado pelas dependências estruturais intrinsecamente presentes no código fonte.

VII. Conclusão

Processos de remodularização de sistemas de software são normalmente conduzidos quando o processo de erosão arquitetural é negligenciado por longos períodos, anulando os benefícios proporcionados por um projeto arquitetural, tais como manutenibilidade, escalabilidade, portabilidade, etc. O *problema*, entretanto, consiste no fato de que a maioria dos processos de remodularização são pesados e sem apoio ferramental. Diante desse cenário, este artigo – centrado na observação de que a remodularização de um sistema de software pode ser realizada por sucessivas aplicações de refatorações *Move Class*, *Move Method* e *Extract Method* – propôs um processo de remodularização leve, com apoio ferramental e baseado em similaridade estrutural.

O processo proposto foi avaliado em uma versão modificada de um sistema de código aberto onde três modificações foram propositalmente realizadas. A execução de um ciclo completo complementado por uma etapa adicional foram necessárias para reestabelecer a arquitetura original do sistema, o que resultou em uma revocação de 100% e precisão de 50%.

Um estudo empírico apoiou as principais decisões de projeto do processo de remodularização proposto. A decisão do uso de similaridade estrutural foi apoiado pelo fato de que 77% dos métodos apresentaram melhoria na similaridade estrutural após a refatoração. A decisão da não verificação de pré-condições para refatoração automática é apoiada pelos fatos de que apenas quatro refatorações (11%) poderiam ser aplicadas de forma totalmente automática e 17 refatorações (47%) sofreram alguma forma de modificação. A decisão de identificar a entidade de destino sob demanda é apoiada pelo fato de que 75% dos métodos foram movidos para novas classes.

Como trabalhos futuros, pretende-se: (i) estender a avaliação do processo proposto em cenários reais de remodularização; (ii) propor heurísticas para evitar o reporte de falso positivos; (iii) investigar a relação entre refatorações *Extract Method* e blocos estruturais; e (iv) estender a investigação empírica para também contemplar instâncias reais de refatorações *Move Class* e *Extract Method*.

Agradecimentos

A CAPES apoiou este trabalho pelo Programa STIC-AmSud (processo n^o 054/2014) e pela bolsa de pós-doutorado (processo n^o 8028-14-1); além do apoio da FAPEMIG e CNPq.

Referências

[1] Nicolas Anquetil and Jannik Laval. Legacy software restructuring: Analyzing a concrete case. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 279–286, 2011.

[2] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Pearson, 2nd edition, 2011.

[3] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671–694, 2014.

[4] Jens Borchers. Invited talk: Reengineering from a practitioner's view – a personal lesson's learned assessment. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 1–2, 2011.

[5] Lakshitha de Silva and Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[6] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of Extract Class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, 2012.

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

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

[9] James N Miller. Tutorial review—outliers in experimental data and their treatment. *Analyst*, 118(5):455–461, 1993.

[10] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[11] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

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

[13] Mark Kent O'Keeffe and Mel Ó Cinnéide. Search-based software maintenance. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 249–260, 2006.

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

[15] David Lorge Parnas. Software aging. In *16th International Conference on Software Engineering (ICSE)*, pages 279–287, 1994.

[16] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, 1992.

[17] N. Rachatasumrit and Miryung Kim. An empirical investigation into the impact of refactoring on regression testing. In *28th International Conference on Software Maintenance (ICSM)*, pages 357–366, 2012.

[18] Girish Maskeri Rama and Naineet Patel. Software modularization operators. In *26th International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

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

[20] 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.

[21] 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.

[22] Ricardo Terra, J. Brunet, L. F. Miranda, Marco Tulio Valente, D. Serey, D. Castilho, and R. S. Bigonha. Measuring the structural similarity between source code entities. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 753–758, 2013.

[23] 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.

[24] Ricardo Terra, Marco Tulio Valente, and Roberto S. Bigonha. An approach for extracting modules from monolithic software architectures. In *IX Workshop de Manutenção de Software Moderna (WMSWM)*, pages 1–8, 2012.

[25] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. Recommending refactorings to reverse software architecture erosion. In *16th European Conference on Software Maintenance and Reengineering (CSMR), Early Research Achievements Track*, pages 335–340, 2012.

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

[27] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of Extract Method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[28] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61:105–119, 2002.