# Um estudo sobre o efeito de Mover Classe na Arquitetura de Sistemas

**Vagner Clementino**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)

vagnercs@dcc.ufmg.br

***Resumo.*** *Recentemente estudos vêm focando em entender as motivações que levam os desenvolvedores a realizem refatoração do código. Existe um consenso que a causa original é remover porções de código com baixa qualidade conhecidos como Bad Smells. Todavia, estudos demonstraram que o desenvolvedores refatoram para outros fins. Neste sentido, este trabalho se propõe em analisar a relação entre mudanças na arquitetura de um sistema e a refatoração Mover Classe. Após a análise de 52 mil commits de 04 sistemas de código aberto hospedados no GitHub verificou-se em todos o fenômeno de* `batching moving`*, que é movimentação substancial de classes dentro do sistema. Este evento tem impactos na arquitetura do software.*

## 1. Introdução

A atividade de refatoração tem por objetivo alterar o código fonte de um software sem modificar o seu comportamento. Em última instância, refatorar visa melhorar a qualidade interna do sistema [Fowler 1999, Opdyke 1992]. Sua importância é reconhecida tanto na literatura quanto na indústria, no qual, nesta última, é possível verificar a existência de processos de desenvolvimento que incorporam a refatoração como atividade rotineira [Beck and Fowler 2000]. No últimos anos, pesquisas foram realizadas com o objetivo de entender com qual frequência os desenvolvedores aplicam diferentes tipos de refatoração [Murphy-Hill et al. 2009]; a relação entre a atividade de refatorar e a correção de *bugs*[Kim et al. 2011], bem como nos resultados de testes de regressão [Kim and Rachatasumrit 2012]. No trabalho de kim et.al discute a percepção dos desenvolvedores sobre a refatoração [Kim et al. 2012].

Recentemente estudos vêm focando em entender as motivações que levam os desenvolvedores a realizem refatoração. Existe um consenso que a causa original é remover porções de código com baixa qualidade conhecidos como *Bad Smells* [Fowler 1999]. Todavia, estudos demonstraram que o desenvolvedores refatoram para outros fins. Por exemplo, a refatoração *Extrair Método*, pode ser utilizada para fins de extensão do sistema ou mesmo possibilitar compatibilidade com versões anteriores do código [Tsantalis et al. 2013a]. É possível verificar o emprego de *Extrair Método* com intuito de favorecer a reutilização de código no longo prazo, mesmo que inicialmente não seja esta a intenção [Silva et al. 2015].

Apesar da existência de estudos relativos à motivação da refatoração, ao bem do nosso conhecimento, não existem trabalhos que relacionem a refatoração *Mover Classe* com mudanças na arquitetura do sistema. Suspeitamos que o fato de um desenvolvedor

mover classes em um número maior que padrão do sistema tem por objetivo alterar a arquitetura do sistema. Este tipo de ação é conhecida como `batching moving`. Uma alteração na arquitetura pode ter como objetivo, por exemplo, reorganizar o código existente em uma nova camada lógica ou ainda tratar o problema da *Erosão Arquitetural*. O processo de Erosão arquitetural é conhecido como os desvios ocorridos no código de um sistema que causam violação de alguma regra arquitetural previamente estabelecida [Perry and Wolf 1992].

Neste sentido, este trabalho se propõe em analisar a relação entre mudanças na arquitetura de um sistema e a refatoração *Mover Classe*. A fim de investigar tal relação analisamos o histórico de versões de 04 sistemas de código aberto, desenvolvidos em Java e hospedado no `GitHub`. Este estudo visa responder a seguinte questão de pesquisa:

**RQ** Com qual frequência a refatoração Mover Classe tem por objetivo alterar a arquitetura do sistema (`batching moving`)?

Ao responder a questão proposta entendemos que iremos contribuir no aumento do entendimento das razões que levam os desenvolvedores a realizar refatorações. Esta informação poderá ser utilizada posteriormente na construção de ferramentas que ajudem os times de desenvolvimento em tarefe relativas à mudança da arquitetura do software. Além disso será possível verificar a relação entre a atividade de refatoração e a mudança de arquitetura de um sistema.

O restante deste trabalho está organizado da seguinte forma: a Seção 2 descreve a metodologia utilizada neste estudo; a Seção 3 apresenta os resultados e responde a questão de pesquisa; na Seção 4 realiza-se uma discussão sobre as ameaças à validade do trabalho; a Seção 5 apresenta os trabalhos relacionados à análise da atividade e detecção de refatoração; a Seção 6 sumariza o artigo e discute suas principais contribuições.

## 2. Metodologia

### 2.1. Seleção dos Sistemas

Com o objetivo de responder as questões de pesquisa propostas neste trabalho foram coletados sistemas de código aberto hospedados no *GitHub*. Com cerca de 38 milhões de repositórios[1], GitHub é atualmente o maior repositório de código na Internet. Sua popularidade e a disponibilidade de metadados, acessíveis através de uma API, tem tornando GitHub bastante atrativo para a realização de pesquisas na área de Engenharia de Software.

Para escolha dos projetos foi definido inicialmente um conjunto de critérios baseados em boas práticas recomendadas na literatura [Bird et al. 2009]. Em síntese, um projeto para ser escolhido deve atender aos simultaneamente seguintes requisitos:

- Os projetos devem ter Java como a linguagem principal, por limitações da ferramenta de analise utilizada.

- Os projetos devem ter no mínimo seis meses de desenvolvimento, para evitar projetos que não tenham passado por um tempo de manutenção relevante.

[1] `https://github.com/features`. Acesso em junho/2016.

- Os projetos devem ter no mínimo 200 revisões pelos mesmos motivos da restrição anterior.

- Os projetos escolhidos não devem ser ramificações (*forks*) um do outro projeto, para evitar dados duplicados.
- Os projetos obtidos devem ser os 120 mais populares que atendem aos demais critérios, utilizando como métrica o campo `most stars`

Os projetos foram recuperados através a busca avançada do GitHub[2] utilizados os critérios: *created:*$< 2016 - 01 - 01$ & *language:Java*. O resultado foi ordenado pelo parâmetro *"Most stars"* até o limite de 120 projetos. Ao conjunto recuperado foi aplicado os critérios de escolha definidos previamente. Ao final do processo foram escolhidos os projetos descritos na Tabela 1.

*JUnit* é um *framework* para escrever testes. O *Spring Framework* fornece um modelo de programação para aplicações empresariais baseadas em Java. *Clojure* é uma linguagem de programação de propósito geral com tipagem dinâmica. *Gradle* é uma ferramenta de `build` com foco na automação e suporte a multilinguagem. O conjunto selecionado abrange diversos tipos de sistema, desde de linguagens de programação, passando por *frameworks* de teste e desenvolvimento até chegar em ferramenta de `build`. Trata-se de sistema bem conceituados dentro do seu campo de atuação.

| Projeto | Commits | Branches | Releases | Contribuidores |
|---|---|---|---|---|
| *junit4* | 2105 | 5 | 20 | 123 |
| *spring-framework* | 12178 | 10 | 89 | 160 |
| *clojure* | 2953 | 22 | 100 | 124 |
| *gradle* | 36947 | 50 | 829 | 227 |

**Tabela 1. Projetos Analisados. Os dados apresentados tem como referência 22/06/2016.**

### 2.2. Detecção das Refatorações

Para que fosse possível detectar a presença de `batching moving`, ou seja, de um conjunto de movimentações de classes em um número maior do que o padrão do sistema, se fez necessário detectar a refatoração *Mover Classe*. Com este objetivo, cada alteração de código existente no histórico de versões do sistema foi analisada.

Um problema que surge quando se analisa revisões consecutivas de um sistema, em especial para aqueles que utilizam um sistema de controle de versão distribuído, como é o caso do `git`, é o *implicit branches*. Quando dois desenvolvedores que colaboram no mesmo projeto realizam alterações locais pode ocorrer que ambos repositórios divirjam. No caso de um futuro *merge* que contenha pelo menos dois pais pode resultar duplicação das refatorações detectadas, caso a atividade de refatoração tenha ocorrido em pelo menos um dos *branches*. A Figura 1 exibe uma sequência de *commits* ilustrando o problema da da duplicação da contagem de refatorações. Suponhamos que exista um *refactoring* em $c_4$. Ele será detectado quando compararmos as versões $c_4$ e $c_2$ e também na comparação

[2] `https://github.com/search/advanced`

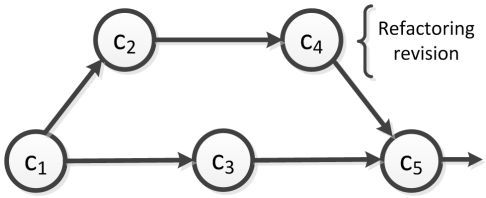


**Figura 1. Exemplo de implicit branches [Tsantalis et al. 2013a].**

das versões $c_5$ e $c_3$. Para evitar este tipo problema foram excluídas das análises as versões que representem uma operação de `merge`, ou seja, que possuam dois ou mais pais.

As refatorações foram detectadas utilizando a técnica proposta Tsantalis et al. [Tsantalis et al. 2013a] que é baseada no algoritmo `UMLDiff` [Xing and Stroulia 2005]. Tendo em vista a ausência de código compilado foi utilizado uma implementação da técnica com algumas adaptações [Silva et al. 2015]. A ferramenta proposta Tsantalis et al. é capaz de detectar 11 tipos diferentes de refatoração. Para a refatoração *Mover Classe* a detecção ocorre com base nas seguintes regras definidas por [Biegel et al. 2011]:

- Seja $(p, n, m)$ uma tupla tal que $p$ é o pacote que uma classe pertence, $n$ é o nome da classe e $m$ é o conjunto de métodos e campos que compõe a classe.
- Seja $t$ uma transação que corresponda uma comparação entre duas revisões sucessivas de um sistema
- Seja $C_t^+$ o conjunto de classes adicionadas na transação $t$
- Seja $C_t^-$ o conjunto de classe removidas na transação $t$
- A refatoração é detectado quando as duas condições a seguir são verdadeiras:
  - $\exists(p, n, m) \in C_t^-$
  - $\exists(p^{'}, n, m) \in C_t^+$

Em resumo a detecção do refatoração *Mover Classes* consiste em verificar se uma classe que previamente existia um em pacote $p$ na versão $v_1$ está no pacote $p^{'}$ na versão subsequente $v_2$. No estudo empírico de [Tsantalis et al. 2013a] foi verificado um número baixo de falsos positivos e por consequência uma alta precisão quando utilizado este conjunto de regras na detecção de Mover Classe.

## 3. Resultados

Neste seção apresentados os resultados para a questão de pesquisas propostas. Em resumo, este estudo avaliou um total de *52001* diferentes versões de sistemas, do qual foi possível detectar o montante de *35402* refatorações diferente. A Tabela 2 exibe os valores da média, mediana e desvio padrão da distribuição de Mover Classe entre as diferente versões dos sistemas.

| Projeto | Total | Média | Mediana | Desvio Padrão | Valor do batching moving |
|---|---|---|---|---|---|
| clojure | 64 | 15,5 | 2 | 23,08 | 23 |
| gradle | 4253 | 31,3 | 2 | 12,9 | 46 |
| junit4 | 279 | 12 | 2 | 7,9 | 18 |
| spring-framework | 1427 | 26 | 2 | 12,5 | 39 |

**Tabela 2. Media, Mediana, Desvio Padrão e Valor do batching moving**

Do total de refatorações encontrados 17% era do tipo Mover Classe. A coluna *Mover Classe* exibe o número da refatoração em estudo, bem como o seu percentual, para cada sistema. A Tabela 3 detalha estes resultado para cada projeto.

| Projeto | #Commits | #Refactorings | #Mover Classe |
|---|---|---|---|
| clojure | 2923 | 522 | 64 (12,3%) |
| gradle | 35593 | 20258 | 4253 (21%) |
| junit4 | 1721 | 1070 | 279 (26,1%) |
| spring-framework | 11764 | 13552 | 1427 (10,5) |
| Total | 52001 | 35402 | 6023 (17%) |

**Tabela 3. Estatísticas dos Sistemas Analisados**

### 3.1. Com qual frequência a refatoração Mover Classe tem por objetivo alterar a arquitetura do sistema (`batching moving`)?

As Figuras 2, 3, 4, 5 exibem a frequência da refatoração Mover Classe nos sistemas estudados. Por exemplo, na Figura 2 é possível verificar a ocorrência de uma (01) refatoração do tipo Mover Classe onde um total de 54 itens foram movidos. Da mesma forma, a Figura 3 demonstra os resultados do sistema `gradle` em que na maior parte das revisões o máximo de classe movidas chegam à 40 itens.

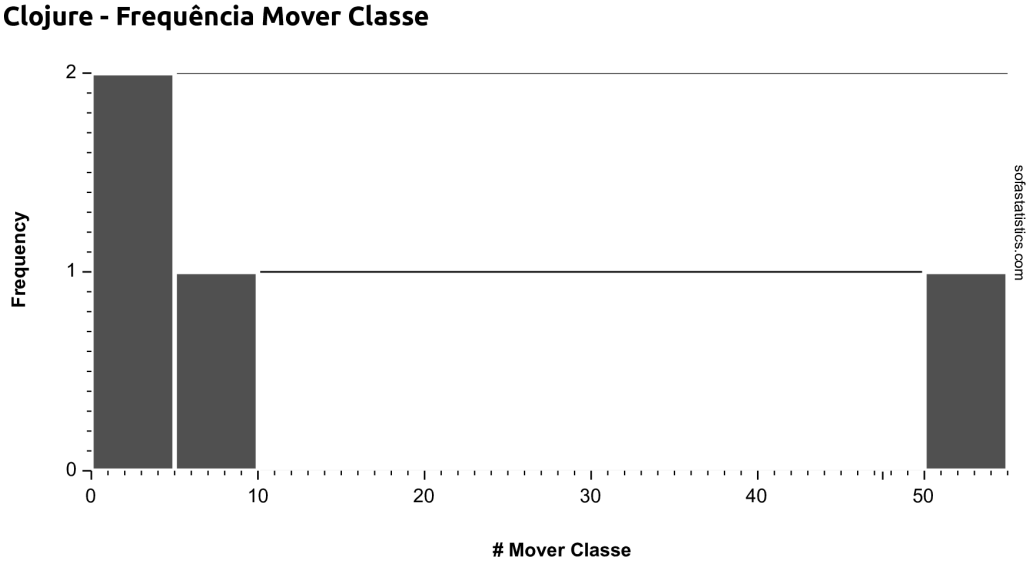


**Figura 2. Clojure - Frequência de Mover Classe**

É possível observar uma tendência de um menor número de classe sendo movimentadas, contudo, em uma maior frequência. Não obstante, ocorre alguns eventos em que um grande número de classe são movidas, como por exemplo, no sistema `gradle` onde foi detectado a refatoração Mover Classe para mais de 200 classes. Este tipo de ação possivelmente causa alteração na arquitetura dos sistema.

A fim de responder a questão de pesquisa proposta é exibida a Tabela 4 a frequência de ocorrência de movimentação de classes em um número maior ou igual a coluna *Classe Movidas*. Por exemplo, no projeto `junit4` cerca de 24% do Mover Classe

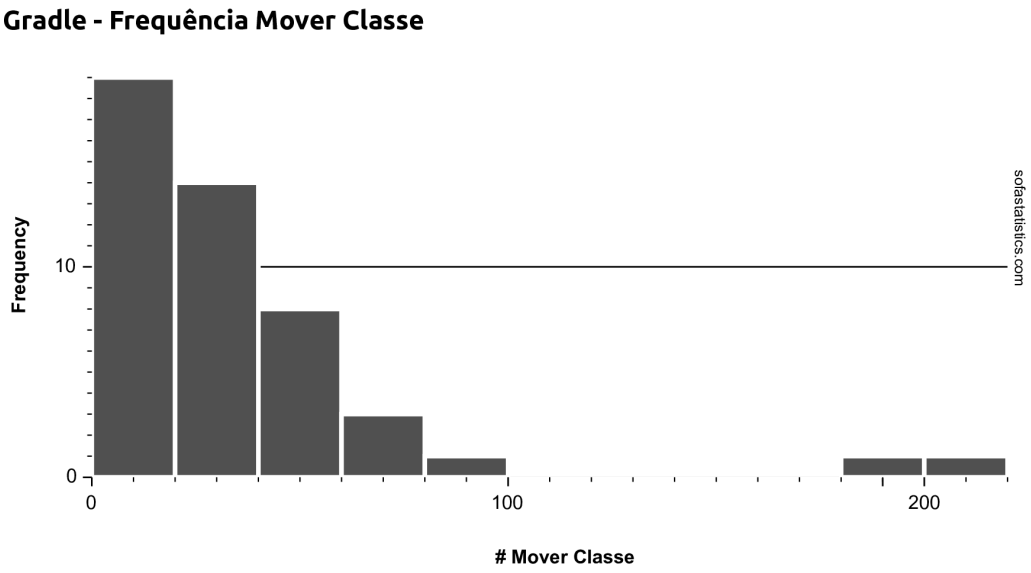


**Figura 3. Gradle - Frequência de Mover Classe**

ocorre com 5 ou mais classes. Em média, cerca de 27% das movimentações de classe foram com 5 ou mais classe.

Os resultados da Tabela 4 demonstram que a frequência de `batching moving` é relativamente baixa nos sistemas analisados. Se consideramos um limiar (threshold) de *40* classe movidas por revisão, *a ocorrência média é de 6%*. Todavia, da baixa frequência foi possível detectar este tipo padrão em todo os sistemas analisados.


**Figura 4. JUnit - Frequência de Mover Classe**

## 4. Ameaças à Validade

As ameaças à validade de qualquer estudo podem surgir de diversas formas e de diferentes direções. A *Ameaça Interna* avalia se o tratamento utilizado produziu algum efeito, ou

| Classe Movidas | clojure | gradle | junit4 | spring-framework | Média |
|---|---|---|---|---|---|
| *5* | 0,40 | 0,208 | 0,237 | 0,271 | 0,279 |
| *10* | 0,20 | 0,063 | 0,136 | 0,136 | 0,13375 |
| *15* | 0,20 | 0,047 | 0,085 | 0,104 | 0,109 |
| *20* | 0,20 | 0,027 | 0,068 | 0,086 | 0,09525 |
| *30* | 0,20 | 0,027 | 0,017 | 0,054 | 0,0745 |
| *40* | 0,20 | 0,021 | 0,017 | 0,036 | 0,0685 |
| *50* | 0,20 | 0,010 | - | 0,018 | - |
| *60* | - | 0,008 | - | 0,009 | - |
| *70* | - | 0,007 | - | 0,009 | - |
| *80* | - | 0,005 | - | 0,005 | - |
| *90* | - | 0,002 | - | 0,005 | - |
| *100* | - | 0,002 | - | 0,005 | - |
| *150* | - | 0,002 | - | - | - |
| *200* | - | 0,001 | - | - | - |

**Tabela 4. Frequência da Movimentação das Classes**

seja, se a variável independente realmente causa mudanças observáveis na variável dependente examinada [Kazman et al. 2016]. Neste estudo existem pelos menos as seguintes ameças internas:

- A precisão dos resultados obtidos dependem do quão precisa é a ferramenta de detecção utilizada. Desta forma, existe a possibilidade de ocorrência de falso negativos, pois não ha uma estimativa confiável da revocação da ferramenta.
- Não é possível definir um limiar (threshold) no qual podemos afirmar que a quantidade de classe movidas estão alterando a arquitetura do sistema. Assim a determinação `batching moving` dependerá da quem está analisando o sistema.

*Ameaça Externa* é uma análise crítica do quanto os resultados obtidos em um estudo podem ser generalizados. Neste trabalho, a análise recai sobre os critérios de seleção dos sistemas, em especial os que estão listados a seguir:

- O total de sistemas escolhidos não permitem a generalização dos resultados.
- Os sistema escolhidos pertencem a única linguagem de programação, desta forma, poderíamos esperar outros resultados quando utilizados software desenvolvidos em outras linguagens.
- Os sistemas analisados são todos de uma única fonte (`GitHub`). Os resultados podem ser diferentes para sistemas comerciais ou desenvolvidos em outras comunidades de desenvolvimento.

## 5. Trabalhos Relacionados

Apesar de sua reconhecida importância os padrões de uso do refactoring é ainda pouco conhecido. Além disso, a sua aplicação de forma manual pelos desenvolvedores é bastante custosa. Afim de preencher estas lacunas diversos trabalhos vêm sendo propostos.

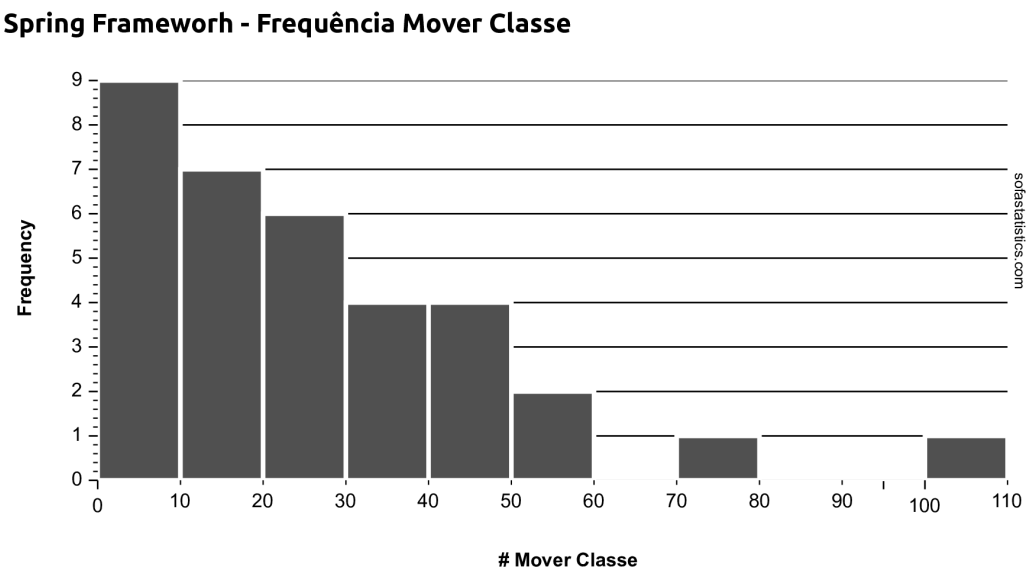


**Figura 5. Spring Framework - Frequência de Mover Classe**

No trabalho de [Dig et al. 2009] é apresentado uma ferramenta visa facilitar o desenvolvimento de aplicações multi-threads em java por meio de refactorings automatizados na biblioteca *java.util.concurrent*. Os resultados demostraram que a ferramenta conseguiu detectar oportunidades de refatoração que os desenvolvedores negligenciaram, ao mesmo tempo que conseguir uma boa taxa de conversão de código.

Em [Silva et al. 2014], Silva et. al propõe-se uma ferramenta visando automatizar a recomendação refactorings. O estudo apresenta uma nova abordagem para identificar e ranquear oportunidades de *Extract Methods* de modo a ser automatizado nas IDE's. Por meio de um estudo exploratório foi possível verificar que a ferramenta atingiu um percentual de 48% de precisão ao avaliar uma amostra de 81 extracts methods artificialmente gerados.

Seguindo esta mesma linha, o trabalho de [Sales et al. 2013] propõe uma ferramenta de recomendação de *Move Method* refactorings utilizando uma técnica denominada *Dependency Sets*. A partir da premissa que "métodos em classes bem desenhadas geralmente estabelecem dependências entre tipos semelhantes"os autores avaliam a similaridade das dependências estabelecidas por um método com os demais em sua classe com os demais métodos das outras classes no sistema. Neste sentido é possível ranquear as classe e por conseguinte recomendar a mudança do método para uma das classes que possui maior similaridade que sua classe atual. Nos testes efetuados esta técnica se mostrou superior àqueles apresentadas pelo *JDeodorant*[3], um sistema de recomendações de move method bem conhecido. Na mesma linha de pesquisa, contudo com o foco bem distinto, em [Feldthaus et al. 2011] temos uma proposta de refatoração automatizada de uma linguagem de tipagem dinâmica. Percebe-se na literatura uma concentração de pesquisas sobre refactoring em linguagens de tipagem estática, tais como Java, para as quais é possível tirar vantagem de informações por meio da análise estática do código. Em contrapartida, refatoração para linguagens dinâmicas como JavaScript é complicado devido aos identificadores da linguagem serem resolvidas em tempo de execução. A fim

[3]http://www.jdeodorant.com/

de preencher esta lacunas os autores apresentam uma ferramenta para recomendar refactorings em códigos escritos em JavaScript. Apesar do limitado conjunto de refatorações suportados pela ferramenta, ela abre o caminho para o desenvolvimento de técnicas de recomendação de refactorings em linguagem de tipagem dinâmica.

Um segundo grupo de trabalho na sobre refactorings visa avaliar e entender os padrões de uso desta técnica. Podemos dividir estes estudos em duas abordagens principais: aqueles que coletam os dados diretamente da IDE, através de um plugin por exemplo; outra que analisam diferentes versões de um código, por meio de um Sistema de Controle de Versão, a fim de detectar padrões. Em [Robbes 2007] é proposto um repositório alternativo de informações ao qual coleta mudanças incrementais do sistema em estudo diretamente da IDE do programado. Com base nos dados coletados os autores avaliam os padrões de refactoring em dois casos de estudos, além de comparar o resultados com clássica abordagem da coleta de refactorings em CVS.

Na mesma linha [Negara et al. 2013] se propões a analisar as diferenças entre refactorings manual e automáticos, ou seja, entre aqueles realizados diretamente pelo desenvolvedor e aqueles produzidos por meio de uma funcionalidade da IDE. Para tanto, os autores desenvolveram um plugin que coletou um total de 1.520 de trabalho de 23 programadores. O trabalho indicou que em média 30% dos refactoring não chegam aos CVS's. A principal contribuição deste tipo de enfoque é permitir analisar separadamente os padrões de rafactoring manual e automático. Contudo, como os dados são coletados diretamente da IDE, o possível overhead gerado pode desmotivar a participação de um maior número de desenvolvedores. Outra limitador é a pouca abrangência deste tipo de abordagem, tendo em vista que o pesquisador necessitar criar um novo plugin para cada tipo de linguagem/paradigma estudado [Robbes 2007].

Em [Xing and Stroulia 2005] é apresentado o *UMLDiff*, um algoritmo capaz de detectar automaticamente alterações estruturais entre duas versões de um software orientado a objetos. Basicamente ele toma como entrada dois diagramas de classe de um sistema e produz uma "árvore de mudanças"que registra as alterações realizadas no software. Por conta de sua capacidade de exibir as alterações estruturais entre duas versões do sistema, o *UMLDiff* é utilizado com algoritmo base para detecção de refactorings [Tsantalis et al. 2013b]. Este trabalha utiliza este algoritmo como base para detecção de refactorings.

## 6. Conclusão

Este trabalho se propôs em analisar o fenômeno `batching moving` e seu efeito na arquitetura de software. Apesar da sua baixa frequência, chegando em alguns em cerca de 6% do total, este tipo de ação foi detectado em todos os sistemas analisados. Neste sentido, as ferramentas podem ser melhoradas para dar um maior suporte a este tipo de operação. Não obstante, os estudo que avaliam a evolução da arquitetura do software devem ter uma maior atenção a este fenômeno com objetivo de entender quais as motivações de um desenvolver mudar diversas classes do sistema, como por exemplo, mais de 200 classe em uma revisão em uma mesma revisão como demostrado neste estudo.

Como possíveis extensões deste trabalho pretende-se analisar o texto do *log de commit* dos possíveis `batching moving` com o objetivo de avaliar qual a frequência que o desenvolvedor informa que a refatoração teve por objetivo alterar a arquitetura do

sistema. Um novo ponto de ampliação do escopo deste estudo é verificar, mediante uma pesquisa com os desenvolvedores, se Mover Classe foi efetivamente utilizado visando alterar a arquitetura dos sistema.

## Referências


Beck, K. and Fowler, M. (2000). *Planning Extreme Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

Biegel, B., Soetens, Q. D., Hornig, W., Diehl, S., and Demeyer, S. (2011). Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 53–62, New York, NY, USA. ACM.

Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., and Devanbu, P. (2009). The promises and perils of mining git. *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, MSR 2009*, pages 1–10.

Dig, D., Marrero, J., and Ernst, M. D. (2009). Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 397–407, Washington, DC, USA. IEEE Computer Society.

Feldthaus, A., Millstein, T., Møller, A., Schäfer, M., and Tip, F. (2011). Tool-supported refactoring for javascript.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Kazman, R., Goldenson, D., Monarch, I., Nichols, W., and Valetto, G. (2016). Evaluating the effects of architectural documentation: A case study of a large scale open source project. *IEEE Transactions on Software Engineering*, 42(3):220–260.

Kim, M., Cai, D., and Kim, S. (2011). An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 151–160, New York, NY, USA. ACM.

Kim, M. and Rachatasumrit, N. (2012). An empirical investigation into the impact of refactoring on regression testing. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 357–366, Washington, DC, USA. IEEE Computer Society.

Kim, M., Zimmermann, T., and Nagappan, N. (2012). A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 50:1–50:11, New York, NY, USA. ACM.

Murphy-Hill, E., Parnin, C., and Black, A. P. (2009). How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA. IEEE Computer Society.

Negara, S., Chen, N., Vakilian, M., Johnson, R. E., and Dig, D. (2013). A comparative study of manual and automated refactorings. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 552–576.

Opdyke, W. F. (1992). *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA. UMI Order No. GAX93-05645.

Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52.

Robbes, R. (2007). Mining a change-based software repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 15–, Washington, DC, USA. IEEE Computer Society.

Sales, V., Terra, R., Miranda, L. F., and Valente, M. T. (2013). Recommending move method refactorings using dependency sets. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 232–241.

Silva, D., Terra, R., and Valente, M. T. (2014). Recommending automated extract method refactorings. In *22nd IEEE International Conference on Program Comprehension (ICPC)*, pages 146–156.

Silva, D., Valente, M. T., and Figueiredo, E. (2015). Um estudo sobre extração de métodos para reutilização de código.

Tsantalis, N., Guana, V., Stroulia, E., and Hindle, A. (2013a). A Multidimensional Empirical Study on Refactoring Activity. *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pages 132–146.

Tsantalis, N., Guana, V., Stroulia, E., and Hindle, A. (2013b). A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 132–146, Riverton, NJ, USA. IBM Corp.

Xing, Z. and Stroulia, E. (2005). Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA. ACM.