# 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. TODO***

## 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 os 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 motivação 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, como 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. 2013]. É possível verificar a utilização do *Extrair Método* visando 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*. A mudança 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 procedemos com um *survey* com os desenvolvedores visando responder as seguintes questões de pesquisa:

Listar os sistemas escolhidos

**RQ1** Com qual frequência a refatoração Mover Classe tem por objetivo alterar a arquitetura do sistema (`batching moving`)?

**RQ2** Com qual frequência o desenvolvedor informa no *log de commit* que a refatoração teve por objetivo alterar a arquitetura do sistema?

**RQ3** Segundo dos desenvolvedores, Mover Classe foi efetivamente utilizado para alterar a arquitetura dos sistema?

Ao responder as questões proposta neste trabalho 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 as questões de pesquisas propostas; 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 on line do mundo. 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 que foram baseados em boas práticas recomendadas na literatura[Bird et al. 2009]. Em síntese, um projeto para ser escolhido deve atender aos seguintes requisitos:

- Os projetos devem ter Java como a linguagem principal, por limitações das ferramentas de analise utilizadas.
- Os projetos devem ter no mínimo seis meses de desenvolvimento, para evitar projetos que não tenham passado por um tempo de manutenção relevante.
- Os projetos devem ter no mínimo 200 revisões pelos mesmos motivos da restrição anterior.

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

- Os projetos não devem ser ramificações (*forks*) de um 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 critério *"Most stars"* até o limite de 120 projetos. Do conjunto recuperado foi aplicado os critérios de escolha que foram definidos previamente. Ao final do processo foram escolhidos os projetos descritos na Tabela tab:projetos. *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 dos `batching moving`

#### 2.2.1. Detecção do Mover Classe

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 necessária detectar a refatoração *Mover Classe*. Com este objetivo, cada alteração de código existente no histório 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 divergem. No caso de um futuro *merge* que contenha pelo menos dois pais pode resultar em relatórios de refactoring duplicados, caso a atividade de refatoração tenha ocorrido em pelo menos um dos *branches*. A Figura 1 que exibe uma sequência de *commits* ilustrando o problema da da duplicação da contagem de refatorações. No caso de um refactoring em $c_4$ ele será detectado quando comparamos as versões $c_4$ e $c_2$ e também na comparação das versões $c_5$ e $c_3$. Para evitar este problema foram excluídas das análises as versões que representem uma operação de `merge`, ou seja, que possuam dois ou mais pais.

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

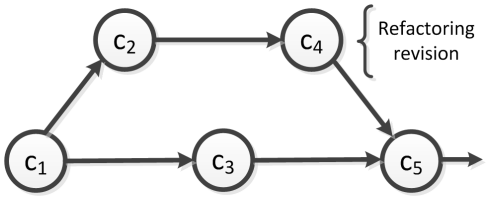


**Figura 1. Exemplo de implicit branches [Tsantalis et al. 2013].**

As refatorações foram detectadas utilizando a técnica proposta Tsantalis et al. [Tsantalis et al. 2013] 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 conforme 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 $t$ que corresponde uma comparação entre duas revisões sucessivas
- 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 da movimentação das classes consiste em verificar se uma classe que previamente existe um em pacote $p$ em uma versão $v_1$ está no pacote $p^{'}$ na versão subsequente $v_2$. No estudo empírico de [Tsantalis et al. 2013] 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.

### 2.2.2. Seleção dos `batching moving`

Conforme a definição proposta neste trabalho, um `batching moving` representa uma movimentação de classe em um número maior do que padrão do sistema. Desta forma, para detectamos este tipo de evento foi necessário avaliar a frequência que a refatoração Mover Classe ocorre nos sistemas em estudo. Neste estudo foi definido que a ocorrência de `batching moving` é determinada por um total de classe movimentadas que é maior o igual 1.5 vezes a média de ocorrência da refatoração Mover Classe no sistema. A Tabela 2 exibe os valores da média, mediana e desvio padrão da distribuição de Mover Classe entre diferente versões dos sistemas. A coluna *Valor do batching moving* apresenta o valor de referência utilizado para detectar movimentações de classes atípicas no sistema.

## 2.3. Análise das Mensagens de Commit

Com o objetivo de analisar se os desenvolvedores reportam que a refatoração Mover Classe teve como objetivo alterar a arquitetura do sistema, foi realizada uma inspeção manual nas mensagens dos commits. Foi avaliado aqueles commits que movimentaram

| 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**

classe um número maior do que 1.5 vezes a média do sistema. Na busca foi avaliado a existência de evidências que a refatoração visava melhorar algum atributo interno do sistema ou ainda adequar a alguma arquitetura previamente definida.

## 3. Resultados

Neste seção apresentados os resultados para as questões 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. 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 número 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 e o número de ocorrência da refatoração Moves Classe. Por exemplo, na Figura 2 é possível verificar a ocorrência de uma refatoração do tipo Mover Classe onde um total de 54 itens foram movidos. A linha tracejada na horizontal representa o valor acumulado da frequência da refatoração estudada. Por outro lado, a linha contínua na horizonte refere-se ao valor limite para determinação de um `batching moving`. Neste exemplo, temos que apenas uma única vez para o sistema `clojure` ocorreu uma movimentação de classe acima do padrão.

## 4. Ameaças à Validade

## 5. Trabalhos Relacionados

## 6. Conclusão

## Referências

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

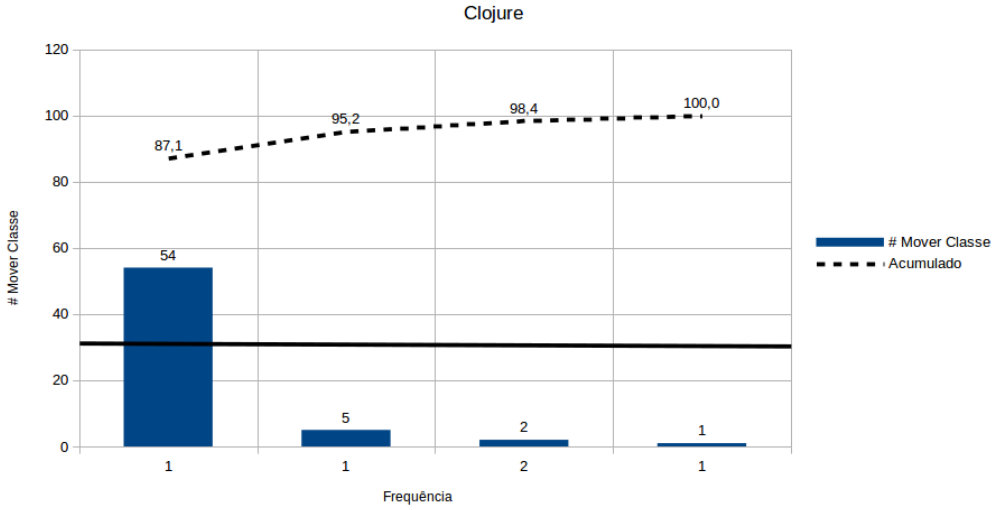


**Figura 2.**

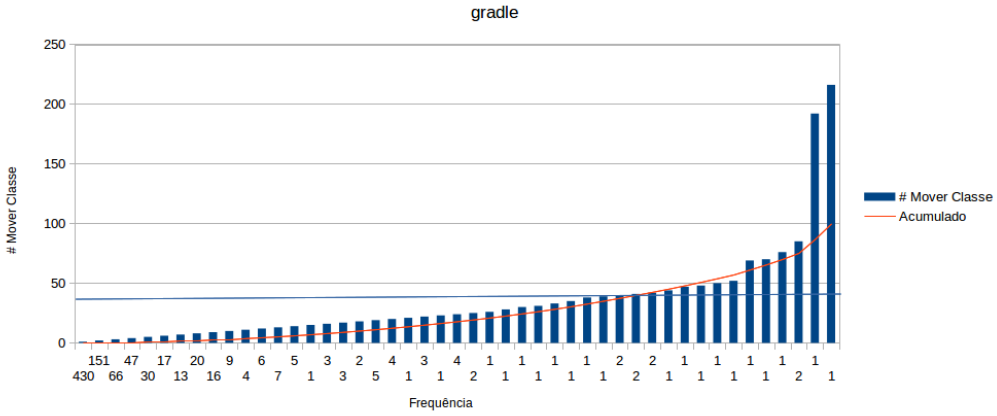


**Figura 3.**

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.

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

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.

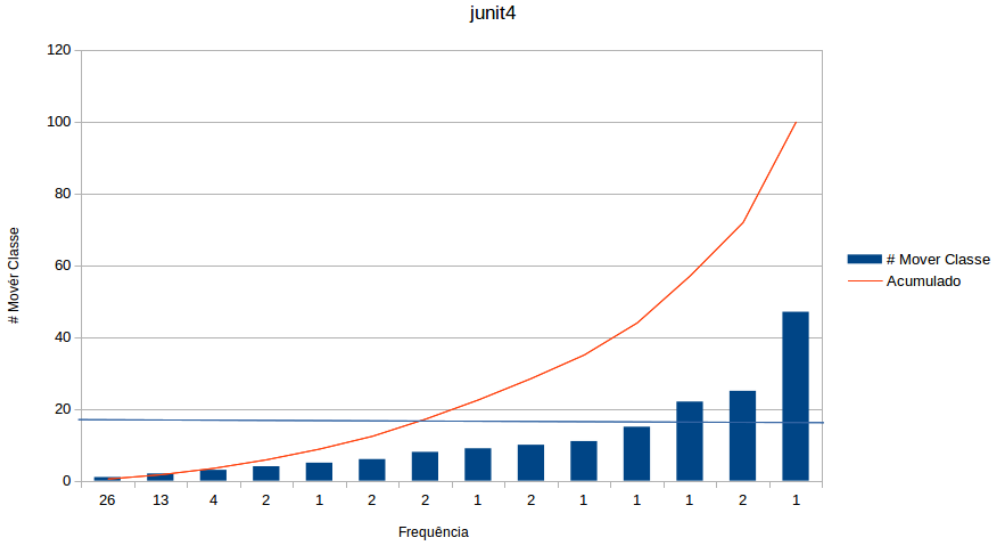


**Figura 4.**

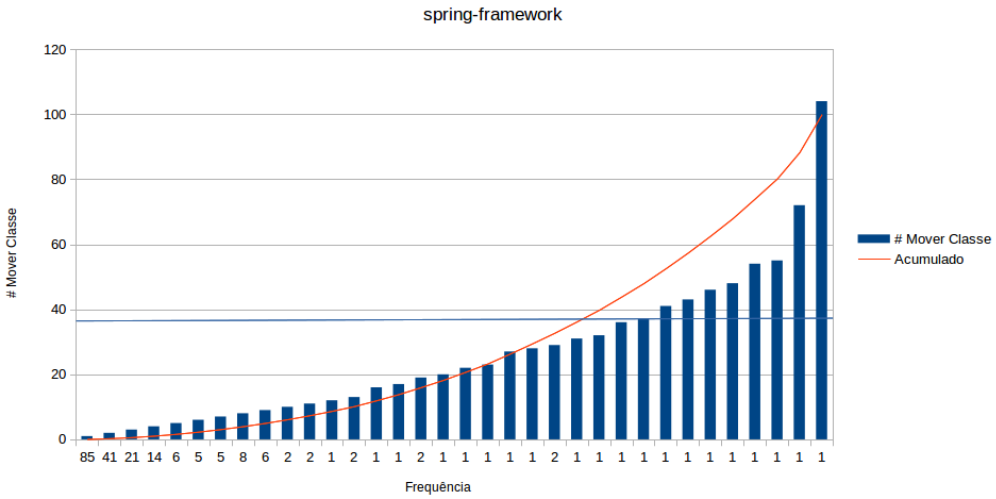


**Figura 5.**

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.

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.

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. (2013). A Multidimensional Empirical Study on Refactoring Activity. *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pages 132–146.

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.