Minerando Refactorings no GitHub

Vagner Clementino
Universidade Federal de Minas Gerais - UFMG
Instituto de Ciências Exatas - ICEX
Departamento de Ciência da Computação
Belo Horizonte - Minas Gerais
E-mail: vagnercs@dcc.ufmg.br

***Resumo*—O estudo dos padrões de refactorings vêm sendo bastante explorado na literatura. Contudo, no tocante à obtenção de seu dataset, a grande parte dos trabalhos focam nos Sistemas de Controle de Versão (SCV) centralizados . Paralelo a isso, o GitHub, que utiliza um SCV descentralizado, vêm nos últimos anos se apresentando como o principal repositório de software da Internet. Neste contexto, o presente trabalho propõe a analisar refactorings em projetos de código escrito em Java e hospedados no GitHub. Com base na análise de 4393 versões de sistema, pertencentes a 16 projetos, foi possível detectar que (i) o tipo de software afeta o padrão de refactorings; (ii) e a atividade de refatoração é restrita a um pequeno grupo de desenvolvedores. O estudo apresenta ainda uma nova abordagem para solucionar o problema do implicit branches que intrínseco à SCV descentralizados e pode causar a duplicidade de refactorings.**

I. INTRODUÇÃO

Refactoring é uma técnica de programação crucial no que se refere a manter o bom desenho e a correta estrutura de um software. Segundo a definição proposta por Fowler [1], trata-se de uma *"alteração realizada na estrutura interna de um software a fim de torná-lo mais fácil de entender e mais barato para modificar, contudo, sem alterar o seu comportamento observável"*. Sua importância chega ao ponto de ser prescrita como atividade chave em alguns processos de desenvolvimento de software, tais como eXtreme Programming(XP) [4]. Ademais, as principais IDE[1] do mercado(Eclipse, NetBeans, IntelliJ IDEA e Visual Studio) incorporam alguns tipos refactoring como funcionalidades nativas em seus menus.

Apesar de sua reconhecida importância, tanto pela comunidade acadêmica quanto pela indústria de software, o conhecimento da dinâmica de uso de refactorings por parte dos desenvolvedores é relativamente baixo. Diversas trabalhos vêm sendo propostos com o objetivo de avançar neste entendimento [12], [14], [26], [27], [29], [31], [34], [36]. O conhecimento trazido por estes estudos é importante tanto para os programadores, que podem melhorar a qualidade do código produzido ao aplicar o que existe de mais atual nesta área, quanto para os desenvolvedores de ferramentas de refactoring, que podem obter *insights* que subsidiar o aprimoramento de tais ferramentas ou mesmo a criação de novas funcionalidades.

A análise dos principais estudos relacionados à refatoração possibilita dividi-los em três grupos distintos, no tocante à metodologia utilizada:

- G_1*:* Estudos que observam a refatoração em laboratório durante um determinado período de tempo;
- G_2*:* Trabalhos que registram um `log` das atividades do programador diretamente na IDE para posteriormente obter os refactorings;
- G_3*:* Pesquisas que analisam `snapshots` de duas versões de um sistema armazenadas em um VCS[2].

Trabalhos que adotam a metodologia em G_1 [25] são naturalmente mais precisos, tendo em vista que permitem observar diretamente todas as alterações realizadas no código. Entretanto, são limitados e de difícil aplicação prática, ao passo que é complicado dispor de desenvolvedores em número significativo e em tempo suficiente. Tais dificuldades tornam este tipo de estudo pouco representativo. Além disso, a observação direta do programador pode inibi-lo ao ponto de influenciar nos resultados.

A abordagem G_2 foi proposta a fim de suprir uma lacuna das pesquisas em G_3 (discutido posteriormente). Devido à metodologia adotada em G_3 não é possível separar refactorings realizados pelo desenvolvedor, chamados de manuais, daqueles produzidos pela IDE, denominados automáticos. Os defensores de G_2 argumentam que esta abordagem é capaz de registrar todas as refatorações realizada no código, situação oposta daqueles trabalhos que analisam `snapshots`, os quais capturam apenas refactorings do código versionado. Nesta abordagem é necessário registrar as atividades de interesse diretamente da IDE. Por conta disso, um importante `tradeoff` a ser analisado é a quantidade de recursos (memória, CPU, espaço em disco e etc) que serão consumidos [27], [29]. Outra limitação está no fato deste tipo de trabalho limitar-se normalmente a determinada linguagem/paradigma de programação [29]. Tal característica leva a necessidade de se criar uma versão do algoritmo de coleta dos dados para cada versão de linguagem/paradigma a ser avaliado, um limitador para a aplicação deste tipo de trabalho em contextos diferentes.

Conforme anteriormente exposto, um dos principais argumentos dos defensores de G_2 é que os trabalhos que analisam versões do sistema em CVS's acabam por não registrar todos os refactorings. Contudo, cabe refletir se é necessário registar todas refatorações do código e qual é realmente o tamanho desta perda. Em [27], que está em G_2, verifica-se que para 06 dos 10 refactorings mais utilizados [41] o percentual de refatorações que não chegam ao VCS é de aproximadamente 5%. Diante do exposto, ao menos que que o foco da pesquisa seja comparar refatoração automática com a manual, não se justifica aplicar metodologia em G_2, tendo em vista a complexidade deste tipo de abordagem.

[1]Integrated Development Environment

[2]Version Control System - Sistema de Controle de Versão

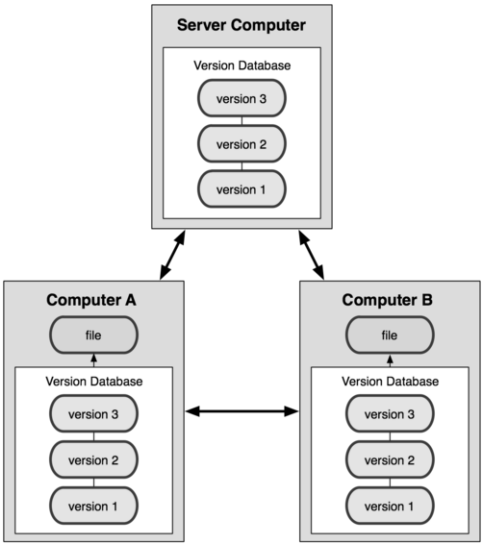

Figura 1. Diagrama SVC Distribuído [7].

O terceiro grupo de trabalhos é o mais recorrente na literatura. Podemos dividi-los entre aqueles que analisam versões de um software armazenadas em um Sistema de Controle de Versão de forma *manual* [3], [12], [21], [22] ou *automática* [2], [10], [11], [15], [17], [38], [39]. A principal vantagem desta abordagem é que ela teoricamente não limita o pesquisador no número de projetos a serem utilizados. Neste sentido, aproveitando-se do crescente uso de repositórios de software abertos, tais como o `GitHub`, trabalhos nesta linha são capazes de captar e analisar um *dataset* bastante diversificado. Tal dataset enriquece a pesquisa ao mesmo tempo que propicia uma maior segurança na proposição de generalizações.

No últimos anos o `GitHub`[3] têm se mostrado um popular repositório de diferentes tipos deprojetos, ao mesmo tempo que se apresenta como uma importante plataforma de colaboração. É objeto de diversos trabalhos em áreas como *social computing* [9], [23], [43], Engenharia de Software [16], [35], [42] dentre outras. Em janeiro de 2013 a plataforma tinha cerca de cinco milhões de projetos e um total de 3 milhões usuários registrados [33]. O GitHub utiliza o `git` [20] para gerenciar o versionamento de código. Diferentemente de outros Sistemas de Controle de Versão, como o CVS[4], Subversion[5] e Perforce[6], o git gerencia o versionamento do código de forma *descentralizada* [7]. Em síntese, em um Sistema de Controle de Versão Descentralizado - SCVD cada máquina cliente possui uma cópia integral e atualizada de todo o código versionado. Neste sentido, cada `checkout` é na realidade um backup completo de todos os dados do projeto, inclusive versões anteriores. A figura1 exibe a arquitetura de um SCVD.

Neste contexto, devido ao seu porte e estrutura diferenciada, o GitHub se mostra um terreno fértil para a realização de trabalhos científicos. Explorando esta tendência este trabalho se propõe a avaliar os padrões e comportamento de refactoring por meio da obtenção e análise de projetos de código aberto hospedados no GitHub e implementados em Java. Os projetos foram coletados do GitHub através da ferramenta `GitScrapper` e os Refactorings foram detectados por com o uso do `RefDetector` [40]. A partir dos dados coletados este trabalho se propõe a responder a seguintes questões:

- *RQ1*: Quais os Refactorings mais recorrentes nos projetos avaliados? O padrão é o mesmo para todos os tipos de software?
- *RQ2*: Qual padrão de contribuição dos Refactorings? Existem colaboradores que executam refatorações com maior recorrência?

O restante de artigo é organizado da seguinte forma: a Seção II detalha os processos de coleta e detecção dos refactorings; na Seção III apresentamos os resultados obtidos e em seguida, na Seção IV, estes dados serão avaliados; a Seção V descreve os recentes trabalhos realizados na área, traçando um paralelo com o estudo aqui proposto; o artigo é finalizado na Seção VI onde se discute os principais achados e contribuições deste trabalho.

II. ABORDAGEM PROPOSTA

Nesta seção discutimos a metodologia de pesquisa adotada. Inicialmente discute-se os critérios de escolha os projetos bem como a sua coleta. Na segunda parte descreve-se a solução proposta a fim de solucionar o problema de comparar revisões subsequentes em repositórios descentralizados. Posteriormente detalha-se o processo de detecção dos refactorings. Esta seção é finalizada com a discussão sobre o uso de amostragem a fim de obter os dados sobre os refactorings.

A. Coletando Repositórios

O termo mais apropriado para esta parte do trabalho seria *"raspagem"* ao invés de coleta de projetos. A ferramente utilizada, denominada `GitScraper` [8], realiza um processo conhecido como *data scraping* [30] a fim de recuperar informações sobre projetos na página de pesquisa do GitHub[7]. Utilizando a API padrão do GitHub a ferramenta é capaz de coletar informações sobre projetos com base em critérios tais como linguagem utilizada, data de criação, data do último *commit* e estado (aberto/fechado). Para cada projeto que atenda aos requisitos especificados o `GitScraper` recupera os seguintes informações:

- Nome do Projeto;
- URL da branch *master* do repositório;
- Data de Criação do projeto;
- Data do último commit;
- Linguagem utilizada;
- Descrição do projeto;
- Total de *stargazers* do projeto;
- Total de *forks* do projeto.

[3]https://github.com/
[4]http://savannah.nongnu.org/projects/cvs
[5]https://subversion.apache.org/
[6]http://www.perforce.com/
[7]https://github.com/search/advanced

Com base na URL fornecida pelo GitScraper é possível clonar localmente um repositório através da API JGit[8], a partir do qual é possível realizar diversas análise no projeto. Conforme discutido anteriormente, em um SCVD um clone local possui todas as informações do repositório, inclusive as versões anteriores. Para este estudo coletou-se projetos que atendem os seguintes critérios:

(a) aberto;
(b) escrito *primordialmente*[9] em Java;
(c) criados a pelo menos dois anos;
(d) último *commit* realizado a pelo menos 30 dias.

Através destas restrições entende-se que é possível recuperar projetos que sejam maduros - critério (c), mas que ainda estejam recebendo contribuições - critérios (a) e (d). A escolha da linguagem é pelo fato de que o `RefDetector` é projetada apenas para coletar Refactorings em Java. Além disso muitos trabalhos nesta área utilizam programas escritos naquela linguagem.

Com base nos dados obtido pelo GitScraper chegou-se a um total de dezesseis projetos escolhidos dentre aqueles com maior número de *fork* e *stargazers*, respectivamente. Um fork é um novo projeto do GitHub que utilizou o código de um projeto já existente como base. Esta métrica reflete a maturidade e estabilidade de determinado projeto partindo-se da premissa que novos projetos não seriam criados de um outro que apresentasse falhas graves. O conceito de stargazers está relacionada com a parte "rede social"do GitHub. Ela indica os usuários do GitHub que "seguem"um determinado projeto e visa refletir a relevância do mesmo dentro de sua própria comunidade. Os projetos foram categorizados em tipos de software (tool, library, framework e android) com base na descrição obtida pelo GitScraper. Cabe ressaltar que o processo descrito anteriormente resultou em 15 projetos. Para o projeto de número 15 escolhemos o *JUnit* tendo em vista o grande número de trabalhos que avaliam aquele sistema. A tabela I exibe os projetos que serão avaliados neste trabalho. Na próxima subseção detalhamos um problema que ocorre quando coletamos refactorings em revisões sucessivas de SCVD, conhecido como *implicit branches*.

B. O Problema das Revisões Sucessivas

Em trabalhos que analisam snapshots do sistema através de Sistemas de Controle de Versão Centralizados - SCVC, tais como o Subversion, a avaliação de refactorings em diferentes versões é relativamente simples, devido à natureza sequencial dos SCVCs. No caso de SVCD como git existe o problema de determinar refatorações em versões do sucessiva do código. O processo de detecção de refactorings consiste basicamente em extrair mudanças que ocorreram em diferentes versões do sistema. Todavia, diferentemente dos repositórios centralizados, que possuem um histórico linear de *commits*, em SVCDs a sequência de versões é modelada como um *Grafo Acíclico Direcionado* [40]. Conforme descrito por [5], quando dois desenvolvedores que colaboram no mesmo projeto realizam alterações locais naturalmente ambos repositórios divergem; contudo, pode ocorrer que posteriormente o git realize um novo *commit* que mesclará as alterações dos dois programadores; neste caso teremos um *merge* que contêm pelo menos dois pais e que poderá resultar na detecção duplicada de refactorings no caso em que a atividade de refatoração tenha ocorrido em pelo menos em uma das *branches*.Este fenômeno é conhecido com *implicit branches*.

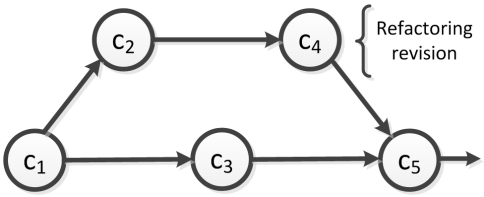

Figura 2. Exemplo de implicit branches [40].

Considere a Figura 2 que exibe uma sequência de *commits* em um projeto git. O nodo c_5 é um *merge* das *braches* c_3 e c_4. Suponha que ocorra um refactoring R em c_4. Caso comparemos as versões c_4 e c_2 é possível detectá-lo, contudo, caso avaliarmos os nodos c_5 e c_3, R também será reportado. Desta forma teríamos a dupla contabilização de um refactoring.

Em [40] os autores propõe como solução a remoção de qualquer versão que possua mais do um pai - c_5 por exemplo. A solução é simples e eficaz, entretanto, a remoção deliberada de versões pode causar a perda de informações relevantes sobre refactorings ocorridas. Com o objetivo de propor uma nova abordagem ao *implicit branches* descreve-se na próxima subseção a solução adotada neste trabalho que visa minimizar a perda de informação.

C. Solução Proposta

No git cada *commit* é unicamente identificável por meio de uma string que é o resultado da aplicação da função hash *SHA-1* no conteúdo enviado para o servidor [7]. Com base nesta informação é possível criar um *Grafo Acíclico Direcionado - GAD* capaz de identificar para cada nodo o grafo o(s) seu(s) respectivo(s) ancestral(is). Na proposta utilizada neste trabalho, realiza-se a detecção de refactorings em versões do sucessiva do código, contudo, utiliza-se as informações obtidas pelo GAD para excluir refatorações que já foram detectadas em algum ancestral da versão avaliada.

Considere a situação apresentada na Figura 2. Conforme exposto caso ocorra um refactoring R em c_4, ele será detectado através da comparação entre c_4 e c_2. Posteriormente R será novamente registrado quando compararmos c_5 e c_3. Na solução proposta o algoritmo de colecta de refactoring utilizaria informação que de c_4 e c_3 são os pais de c_5 a fim de desconsiderar R quando da comparação entre c_5 e c_3. Desta forma, usando a heurística de que um refactoring detectado em merge (commit com dois ou mais pais) é uma duplicata quando ele for detectado em algum ancestral, consegue-se, portanto, tratar de forma simples e eficaz o implicit branche.

D. Detectando Refactorings

A detecção de refatorações é realizada através da ferramenta `RefDetector` [40] que é baseada em uma versão simplificada do algoritmo *UMLDiff* [44]. O algoritmo inicia sua operação através da comparação *top-down* nos nomes dos

[8]https://eclipse.org/jgit/

[9]Existem projetos com a label Java mas que possuem partes em outras linguagens

Tabela I. PROJETOS ANALISADOS

TIPO DE SOFTWARE	NOME	FORKS	STARGAZERS	IDADE(ANOS)[10]	LOC	ÚLTIMO COMMIT
FRAMEWORK	jodd	154	325	2	94110	0f87c00
	junit	1143	2959	5	16682	26f9eba
	libgdx	3362	4842	2	177680	e668972
	spring-framework	3551	4346	3	340319	2f03945
LIBRARY	async-http-client	465	1576	3	23615	a377b17
	dropwizard	1015	2464	3	11126	fc20f7d
	scribe-java	978	2342	4	3420	e47e494
	twitter4j	597	1285	5	21329	2f4a9ff
TOOL	antlr4	173	583	4	30058	10b9fdf
	OpenRefine	352	2259	2	36379	c15f8d1
	SimianArmy	254	2172	2	11417	f7b8d89
	wro4j	128	423	5	27505	eaf3e7d
ANDROID	ActionBarSherlock	17900	3629	3	17900	4a79d53
	Android-ViewPagerIndicator	2500	4434	3	2340	8cd549f
	facebook-android-sdk	1807	3044	4	23049	53bc4fa
	SlidingMenu	3824	6460	2	2455	4254fec

artefatos (classes e interfaces) e na semelhança das assinaturas no caso de métodos. Em seguida, os elementos adicionados / removidos entre duas versões do código são avaliados como compatíveis com base na igualdade de seus nomes a fim de encontrar mudanças em campos e nas assinaturas dos métodos. Ao final deste processo é obtido um conjunto de elementos que sofreram algum tipo de alteração. Na terceira etapa do algoritmo as classes adicionadas ou removidas são comparados com base na similaridade de seus membros a nível de assinatura. Este processo é mais leve do que o algoritmo UMLDiff original, no sentido de que ele não leva em conta as dependências entre os elementos do sistema para calcular a sua similaridade. A tabela II exibe os refactoring passíveis de serem coletados através do RefDetector.

Para cada projeto Git coletado gerou-se um conjunto de versões do sistemas para fins de comparação. Os refatorings foram obtidos através do RefDetector e foram coletados seguindo a proposta descrita na subseção II-C a fim de remover possíveis duplicatas. Os dados obtidos serão apresentados e discutidos seções III e IV.

E. Análise Amostral dos Refactorings

Conforme disposto na subseção II-D, este trabalho utilizou o `UMLDiff` como base para detecção de refactorings. O algoritmo consiste basicamente de gerar uma *Abstract Syntax Tree* (AST) para cada versão do sistema avaliado. A partir da AST gerada é possível identificar alterações na estrutura do código e por conseguinte detectar refactorings. Todavia, o processo de geração da AST é caro, tendo em vista que as informações necessárias à obtenção da árvore são obtidas através da leitura do código-fonte diretamente do disco. Um outro gargalo do processo de detecção de refactórings é o *"checkout"* das versões. Para cada nova versão a ser analisada se faz necessário realizar um checkout, o que é feito diretamente em disco. Conforme dispoto, a detecção de refactorings padece do intrínseco overhead dos processos qeu são realizados em memória secundária.

Devido a restrições de recursos computacionais e de tempo, optou-se neste trabalho por realizar a avaliações dos refactorings apenas em uma amostra do total de commits recuperados para cada projeto. Amostra é um subconjunto de indivíduos extraídos de uma população com a capacidade de representá-la [18]. A utilização de uma amostra implica na aceitação de uma margem de erro denominada *erro amostral*, que é a diferença entre um resultado amostral e o verdadeiro resultado populacional. Neste trabalho, a fim de determinar o tamanho da amostra `n` de cada projeto a fórmula para cálculo da amostra para uma estimativa confiável da proporção populacional, dada pela equação 1.

$$n = \frac{N \cdot Z^2 \cdot p \cdot (1-p)}{(N-1) \cdot e^2 + Z^2 \cdot p \cdot (1-p)} \quad (1)$$

Tabela II. REFACTORINGS DETECTADOS COM PELO REFDETECTOR

REFACTORING	DESCRIÇÃO
Move Operation	Fragmento de código que podem ser agrupados.
Merge Operation	Dois ou mais métodos resultam em único método agrupados.
Extract Operation	Vários clientes utilizarem o mesmo método, ou duas classes têm partes do método em comum
Rename Method	O nome da método não revela seu propósito.
Inline Operation	Consiste em colocar o corpo de um método no o corpo de um dos seus chamadores e remover o método.
Move Attribute	Um campo é, ou será, usado por outra classe mais do que a classe na qual ele está definido.
Extract Superclass	Exitem duas classes com características semelhantes.
Extract Interface	Vários clientes utilizarem o mesmo subconjunto da interface de uma classe, ou duas classes têm partes de suas interfaces em comum.
Move Class	Existe uma classe que faz o trabalho que deve ser feito por duas.
Rename Class	O nome da classe não revela seu propósito.
Rename Method	O nome da método não revela seu propósito.
Extract & Move Operation	Realização simultânea das duas operações.
Convert Anonymous Class to Type	Conversão de uma classe anônima em uma nova classe que implementa o método que recebia a classe anônima como argumento.

O valor de N representa o tamanho do universo, que neste trabalho significa o total de commits recuperados para um dado projeto. A variável Z é o desvio que aceitamos para alcançar o nível de confiança desejado, utilizou neste estudo um nível de confiança igual a 95% ($Z = 1,96$). O valor de e corresponde a margem de erro máxima que devemos admitir, ou seja, o erro amostral. Neste trabalho foi definido um erro amostral máximo de 5%. Em p definimos a proporção populacional de indivíduos que pertence a categoria que estamos interessados em estudar. Nos caso em que não tem qualquer conhecimento prévio da população, como neste estudo, o valor usual de p definido como 0,5 [19]. A tabela III exibe os valores da população (total de commits) e a amostra utilizada para cada projeto.

III. RESULTADOS

Esta seção apresenta os resultados após a realização do processo descrito na seção II. Os dados foram obtidos da avaliação de 4393 versões dos 16 projetos constantes da tabela I. O proceso de coleta dos refactorings 40 horas e resultou em um dataset de 20.872 refatorações recuperadas. Nas próximas subseções iremos apresentar os resultados para cada uma das questões de pesquisa propostas.

A. A Frequência dos Refactorings

A primeira questão propostas foi: *RQ1 - Quais os Refactorings mais recorrentes nos projetos avaliados? O padrão é o mesmo para todos os tipos de software?* A partir desta questão pretendeu-se entender o padrão dos refactorings nos projetos a partir de sua frequência. A figura 3 exibe todos os refactorings recuperados bem como a sua frquência.

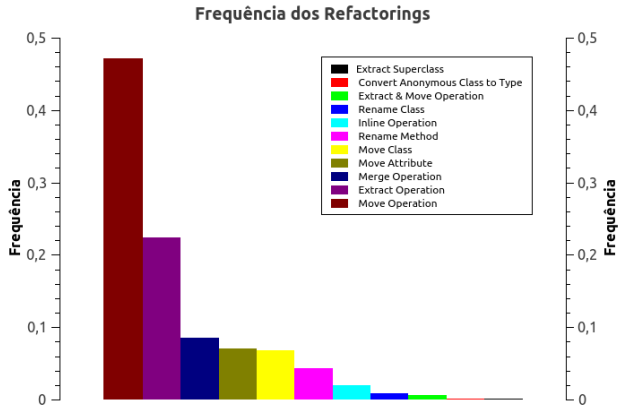

Figura 3. Frequência de refactorings

É possível perceber uma tendência de cauda longa dos refactorings, como uma maior frequência para o *Move Operations*. Os dados são condizentes com outros tabalhos, tanto que dois dos três refactorings de maior frequência (*Move Operations* e *Extract Operation*) também foram os mais observados em [41].

Entrando mais a fundo na questão da frequência do refactorings, foi proposto neste estudo avaliar se o mesmo padrão de comportamento ocorre em diferentes tipos de software. As figuras 4, 5, 6 e 7 apresentam, respectivamente, o percentual de ocorrência das rafatorações em softwares do tipo framework, library, tools e android.

É possível perceber que os que refactorings de maior acabam por se repetir entre as diferenças categorias, como por exemplo, o *Move Operation*. Contudo, o que se verifica é diferente dispersão dos dados, como no caso de projetos do tipo framework (figura 4), onde um único tipo de refactoring representa 80% dos casos.

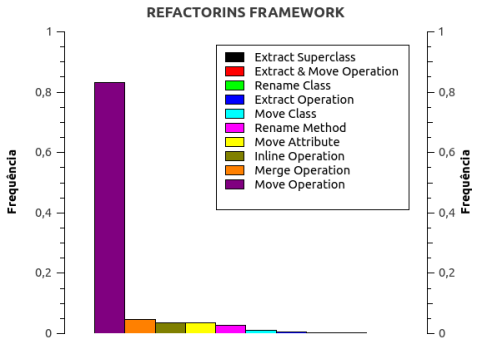

Figura 4. Frequência de refactorings Framework

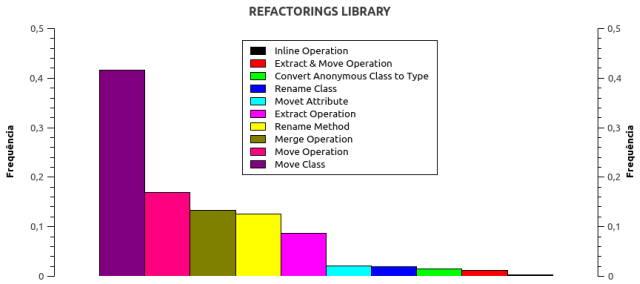

Figura 5. Frequência de refactorings Library

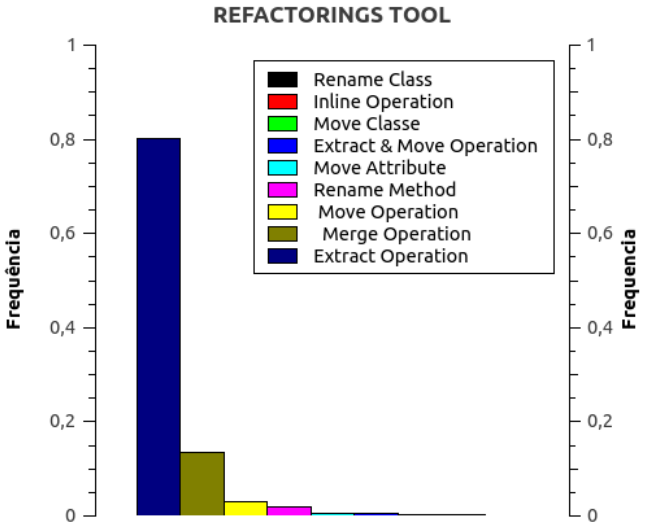

Figura 6. Frequência de refactorings Tools

Tabela III. TOTAL DE COMMITS E DE AMOSTRAS

NOME	TOTAL DE COMMITS	NÚMERO DE AMOSTRAS
jodd	2948	341
junit	1952	322
libgdx	10294	372
spring-framework	133	99
async-http-client	2461	333
dropwizard	2440	333
scribe-java	365	188
twitter4j	1940	322
antlr4	2951	341
OpenRefine	1997	323
SimianArmy	540	225
wro4j	3614	348
ActionBarSherlock	1479	306
Android-ViewPagerIndicator	241	149
facebook-android-sdk	463	211
SlidingMenu	336	180
Total	**34154**	**4393**

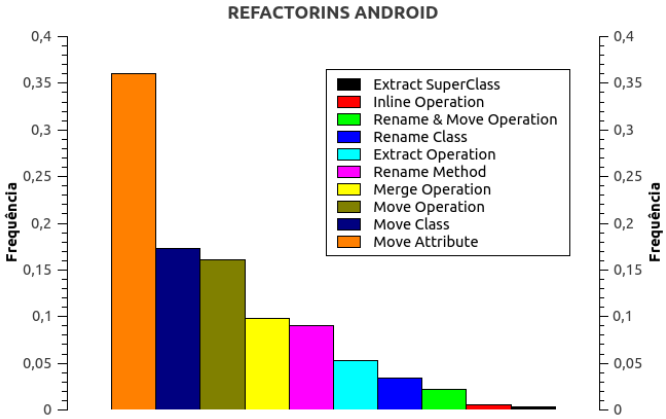

Figura 7. Frequência de refactorings Android

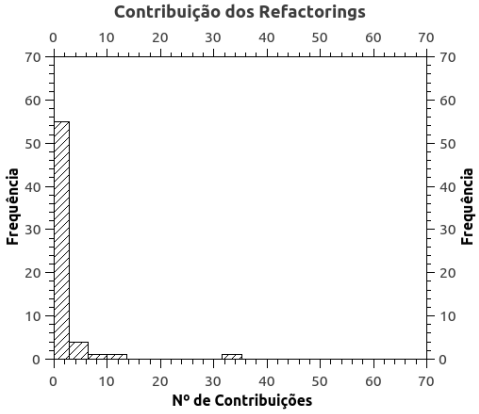

Figura 8. Contribuição dos Refactorings

B. *Perfil de Contribuição dos Refactorings*

Na segunda questão proposta - *RQ2 - Qual padrão de contribuição dos Refactorings? Existem colaboradores que executam refatorações com maior recorrência?* pretende-se analisar qual o perfil de contribuição dos participantes do projeto na realização dos refactorings. A tabela IV exibe o número total de contribuintes nas refatorações, a média e o total de refactorings coletados para cada um dos projetos analisados.

Os dados comprovam uma propriedade dos softwares livres: indepedente do tamanho da comunidade, existe um grupo restrito de desenvolvedores, denominados *core-developers*, que efetivamente realizam as principais atividades do projeto. Para exemplificar analisemos o projeto *junit*. Apesar de ter 1143 forks e 2959 stargazers, apenas 16 desenvolvedores realizam refactorings dentre a amostra analisada. Tal número representa 0,5% do total de stargazers do projeto. Outro exemplo é o projeto *jodd* no qual apenas um desenvolvedor foi responsável por todos os refactorings detectados, apesar de sua comunidade possuir um total de 325 stargazers.

Reforçando a existência dos *core-developers*, a figura 8 exibe a frequência de contribuição nos refactorings por parte dos desenvolvedores. Conforme pode ser observado, aproximadamente 70% dos programadores contribuiem com no máximo 10 refactorings. Daquele total, quase 55% corresponde a no máximo 02 refactorings.

IV. AVALIAÇÃO

Esta seção é dedicada a uma análise mais aprofundada dos resultados obtidos. Ao mesmo tempo avaliamos as possíveis ameaças à validade deste trabalho.

A. *Análise dos Dados*

Os dados sobre a frequências dos refactorings, especialmente o exibido na figura 3, é condizente àqueles em outros trabalhos sobre o assunto. Uma nova contribuição para o estudo dos refactorings está na análise da frequência por tipo de software. Com base na análise das figuras 4, 5, 6 e 7 pode-se perceber que existe uma profunda diferença na distribuição dos refactorings. Tal distinção deve-se ao fato das diferenças arquiteturais e de processo intríseca a cada tipo de software. Por exemplo, aplicações da categoria *android*, possui uma menor dispersão na frequênia dos refactorings, o que pode

Tabela IV. REFACTORINGS POR PROJETO

TIPO DE SOFTWARE	NOME	TOTAL DE CONTRIBUIDORES	MÉDIA DE REFACTORINGS	TOTAL DE REFACTORINGS
FRAMEWORK	jodd	1	139,00	139
	junit	16	22,75	364
	libgdx	36	283,31	10199
	spring-framework	1	5,00	5
LIBRARY	async-http-client	6	99,67	598
	dropwizard	13	74,62	970
	scribe-java	3	7,67	23
	twitter4j	15	22,47	337
TOOL	antlr4	3	52,33	157
	OpenRefine	5	26,80	134
	SimianArmy	16	5,44	87
	wro4j	12	414,08	4969
ANDROID	ActionBarSherlock	5	180,40	902
	Android-ViewPagerIndicator	2	54,00	108
	facebook-android-sdk	16	39,38	630
	SlidingMenu	9	138,89	1250
	Total	**159**	**95,13**	**20872**

ser resultado do fato daqueles projetos estarem vinculados a um framework/arquitetura mais engessado.

No tocante a análise da participação dos desenvolvedores nos refactorings, os resultados demostraram uma tendência anteriormente observada por [24]. Neste sentido, da mesma forma como em atividades de evolução dos software, o refactoring acaba por ser responsabilidade de um pequeno grupo de desenvolvedores. Conformo exposto, está é uma característica dos projetos de código aberto. Contudo, o refactoring é uma técnica que exige do programador um profundo conhecimento do código a ser refatorado, portanto, naturalmente será realizada por algúem do *core-development team*.

B. Ameaças à Validade

Este trabalho utilizou-se de dados coletados via API do GitHub afim de construir o seu dataset. Aqueles dados são fornecidos em um formato não estruturado, especificamente no padrão JSON[11]. O processo e coleta e transformação em formato relacional pode acarretar em perda ou alteração de informação.

Outra ameaça aos resultados ora apresentados está no uso de uma amostra dos commits ao invés de todo o histórico coletado. Conforme exposto, devido à restrições de tempo e de recursos optou-se em avaliar apenas uma parte dos commits. Apesar do rigor adotado, o uso de amostra implica necessariamente na aceitação de uma taxa de erro. Neste estudo adotou-se uma taxa de erro de 5%, valor este amplamente adotado por pesquisadores em diversas áreas do conhecimento.

O uso do *UMLDiff* para detecção dos refactorings reflete diretamente nos resultados obtidos. A existência de falsos positivou ou falsos negativos não foi minizada, tendo em vista que coube exclusivamente ao *RefDector* o processo de obtenção de refactornigs. Todavia, estas ferramentas foram utilizadas em outros estudos apresentado resultados satisfatórios.

Um outro ponto sensível deste trabalho foi o uso de uma heurística como solução ao problema do *implicit branche*. Ao definir que um refactoring detectado em um "merge commit"`C`(commit com dois ou mais pais) e que foi posteriormente observado em algum anscentral `C'` de `C` deverá ser descartado, naturalmente corre-se o risco de remoções indevidas (falsos positivos) e manutenção de duplicatas (falsos negativos). Neste sentido, um possível desdobramento deste trabalho seria aplicar a heurística proposta em um dataset maior com objetivo de avaliar de forma concisa a sua efetividade.

[11] http://www.json.org/

V. TRABALHOS RELACIONADOS

Apesar de sua reconhecida importância, os padrões de uso do refactoring é ainda são poucos conhecido. Além disso, a sua aplicação de forma manual pelos desenvolvedores é bastante custosa. Com o objetivo de preencher estas lacunas diversos trabalhos vêm sendo propostos. No trabalho de [13] é apresentado uma ferramenta que visa facilitar o desenvolvimento de aplicações multi-threads em java por meio da realização 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 conseguiu uma boa taxa de conversão de código. Em [37], 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 [32] 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 apresentados pelo *JDeodorant*[12], um sistema de recomendações de refactorings bem conhecido.

Com um foco bem distinto, contudo, na mesma linha de pesquisa,temos em [14] uma proposta de refatoração automatizada para 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 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. Visando 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 visa avaliar e entender os padrões de uso do refactorings. 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 [29] 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 avaliaram 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 [27] se propõe a analisar as diferenças entre refactorings manual e automáticos, ou seja, entre aqueles realizados diretamente pelo desenvolvedor e aqueles produzidos por meio de duma funcionalidade da IDE. Para tanto, os autores desenvolveram um plugin que coletou um total de 1.520 de horas 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 [29].

Entre os trabalhos que analisam diferentes versões do código através dos VCS's, temos aqueles que realizam análise de forma manual*manual* [3], [12], [21], [22] ou automática [2], [10], [11], [15], [17], [38], [39]. Em Dig *et al.* apresenta-se um algoritmo que recupera os refatorings realizados durante a evolução de um componente afim de reaplicá-los caso o componente venha sofrer algum tipo de problema. Os resultados do trabalho afirmam que o algoritmo proposto possui uma precisão de 85%.

Em [44] 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 [40]. Este trabalho utiliza este algoritmo como base para detecção de refactorings.

Utlizando o *UMLDiff*, materializado através da ferramenta *RefDetector*, Tsantalis *et al.* [40]. apresentam um trabalho que analisa o padrão de refactorings no GitHub. A principal contribuição daquele trabalho está em utilizar um VCS descentralizado. Todavia, os autores utilizam um conjunto pequeno de projetos a fim propor suas conclusões. Ademais, devido ao *Problema das Revisões Sucessivas* (subseção II-B), trabalho é realizado apenas em versões com apenas apenas um "pai", tratando o git com um VCS centralizado. Este trabalho visa avançar nas ideias propostas por Tsantalis *et al.* tanto por avaliar um número maior de projetos tanto por apresentar uma solução ao *Problema das Revisões Sucessivas*.

O GitHub por conta de sua característica de ser um repositório de software ao mesmo tempo que apresenta-se como um rede social, vêm despertando o interesse de diversos pesquisadores. Em [6] os autores analisam a crescente popularidade dos VCS descentralizados em comparação ao tradicional modelo centralizado, bem como o impacto desta mudança no processo de evolução dos softwares. O estudo envolveu 820 participantes e um total de 358300 commits, realizados por 5890 desenvolvedores, obtidos de 132 repositórios. A partir destes dados foi possível (i) verificar que percentual de commmits em repositórios distribuídos é 32% menor do que nos centralizados. Além disso verificou-se que estes commits são mais divididos (no-batch) e relacionados diretamente a resolução de demandas do software.

No trabalho de Onoue *et al.* [28] o foco é entender a dinâmica das atividades realizadas por programadores que participam dos projetos de código aberto hospedados no GitHub. Os estudos sobre a manutenção e evolução dos softwares realizam suas análises com base nos dados obtidos dos sistemas, sem, contudo, levar em consideração o principal agente de mudanças o software avaliado: o desenvolvedor. Os autores propuseram uma categorização dos programadores com base em questões tais como: se preferem que a comunicação no projeto seja realizada através de código ou comentários, ou se eles são definiam como especialistas (responsáveis por diversas atividades no projeto) ou generalistas (sem atividade definida). Utilizando de dados obtidos da API do GitHub foi possível perceber que existe uma divisão clara entre as responsabilidades do desenvolvedores. Por exemplo, alguns deles tipicante são responsáveis por codificar, comentar e resolução de demandas. Além disso, alguns desenvolvedores eram especialistas, realizando atividades específicas em módulos determinados módulos do sistema. Um outro dado interessante mostra que as taxas de atividades variaram de dias a anos entre os desenvolvedores. Todas estas descobertas corroboram que o que foi descrito no clássico artigo de Mockus *et al.* sobre o desenvolvimento de sistemas de código aberto [24].

Em [42] os autores apresentam um trabalho com objetivo entender as razões que levam a redução da taxa de commmits em projetos de código aberto. Este fenômeno interfere diretamente no desenho e no plano de desenvolvimento da aplicação.

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

A partir dos logs de commits no GitHub, analisados por uma ferramenta criada pelos pesquisadores,consegui-se revelar os seguintes padrões: alterações futuras no código podem ser associados com outras mudanças ocorridas anteriormente; os commits dos desenvolvedores podem ser agrupados em baths, contudo, a frequência de commits que afetam muitos arquivos é três vezes menor do que aqueles que alteram um menor grupo de código fonte. Essa informação é valiosa para o planejamento de projetos que utilizam o GitHub como repositório.

VI. CONCLUSÃO

O presente trabalho analisou refactorings em projetos de código aberto escritos em Java e hospedados no GitHub. Desenvolveu-se uma ferramenta que por meio da API oficial do GitHub coletou-se dados de 16 projetos. A partir de uma amostra de 4393 commits foi possível detectar que

(i) as frequências dos refactorings segue o mesmo padrão apontados em outros estudos;
(ii) o padrão dos refactorings varia dependendo do tipo de software analisado;
(iii) assim como outras atividades nos projetos de software livre, a refatoração está restrita a um grupo pequeno de desenvolvedores.

Além disso o trabalho propôs uma nova abordagem para solucionar o problema do *implicit branche*. Apesar da solução conseguir resolver o problema no contexto deste trabalho, se faz necessário a validação de sua eficiência em um conjunto de dados maior.

A despeito dos resultados satisfatórios deste estudo, ele permite o desdobramento em trabalhos futuros. São eles:

(i) avaliar a heurística para o problema do *implicit branche* em um dataset maior;
(ii) analisar o refactoring de forma integral, avaliando todos os commits de um grupo maior de projeto com objetivo de aumentar a confidenciabilidade dos resultados;
(iii) propor um novo algoritmo de detecção de refactorins que seja mais leve, possibilitando a sua aplicação em datasets maiores;
(iv) desenvolver ferramentas que facilitem a automatização da coleta de dados do GitHub.

A necessidade de entender o padrão comportamental dos refactorings deixou de ser uma tendência para ser tornar uma realidade. Pesquisadores, engenheiros de software e desenvolvedores estão cada mais interessados no desenvolvimento de técnicas e ferramentas com o objetivo de melhorar a qualidade do software produzido bem como que reduzam os custos de manutenção e evolução dos sistemas. Paralelo a tudo isso, o GitHub vêm se tornando de fato o principal de repositório de software da internet. Neste sentido, pesquisas que envolvam processos da engenharia de software a partir dos dados do GitHub se mostram bastante promissoras.

REFERÊNCIAS

[1] *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] G. Antoniol, M. D. Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *In IWPSE*. IEEE Computer Society, 2004, pp. 31–40.

[3] J. Bansiya, *Object-Oriented Application Frameworks: Problems and Perspectives*, 1998, ch. Evaluating Application Framework Architecture Structural and Functional Stability.

[4] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004.

[5] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSR.2009.5069475

[6] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 322–333. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568322

[7] S. Chacon, *Pro Git*, ser. Books for professionals by professionals. Apress, 2009. [Online]. Available: http://books.google.com.br/books?id=3XcW4oJ8goIC

[8] V. Clementino. (2014) Gitscraper wiki. Acessado em 12/10/2014. [Online]. Available: https://github.com/vagnerclementino/GitScraper/wiki

[9] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: Transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, ser. CSCW '12. New York, NY, USA: ACM, 2012, pp. 1277–1286. [Online]. Available: http://doi.acm.org/10.1145/2145204.2145396

[10] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *IN PROCEEDINGS OF OOPSLA '2000 (INTERNATIONAL CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS*. ACM Press, 1999, pp. 166–177.

[11] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *in Proceedings of the 20th European Conference on Object-Oriented Programming, 2006*, pp. 404–428.

[12] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006. [Online]. Available: http://dx.doi.org/10.1002/smr.328

[13] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential java code for concurrency via concurrent libraries," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 397–407. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070539

[14] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip, "Tool-supported refactoring for javascript," 2011.

[15] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," 2005.

[16] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 92–101. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597074

[17] S. Kim, K. Pan, and E. J. Whitehead, Jr., "When functions change their names: Automatic detection of origin relationships," in *Proceedings of the 12th Working Conference on Reverse Engineering*, ser. WCRE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 143–152. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2005.33

[18] J. Levin, *Estatística aplicada a ciencias humanas*. Harbra, 1987. [Online]. Available: http://books.google.com.br/books?id=m9SVPgAACAAJ

[19] D. M. Levine, M. L. Berenson, and D. Stephan, *EstatíStica: Teoria e AplicaçõEs - Usando Microsoft Excel Português*. LTC, 2005.

[20] J. Loeliger and M. McCullough, *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*, ser. Oreilly

and Associate Series. O'Reilly Media, Incorporated, 2012. [Online]. Available: http://books.google.com.br/books?id=ZkXELyQWf4UC

[21] M. Mattsson and J. Bosch, "Frameworks as components: a classification of framework evolution," *Proceedings Nordic Workshop on Programming Environment Research 1998*, pp. 163–174, 1998.

[22] ——, "Three evaluation methods for object-oriented framework evolution – application, assessment and comparison," Tech. Rep., 1999.

[23] N. McDonald and S. Goggins, "Performance and participation in open source software on github," in *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '13. New York, NY, USA: ACM, 2013, pp. 139–144. [Online]. Available: http://doi.acm.org/10.1145/2468356.2468382

[24] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 309–346, Jul. 2002. [Online]. Available: http://doi.acm.org/10.1145/567793.567795

[25] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: Observations and tools for extract method," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 421–430. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368146

[26] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 287–297. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070529

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

[28] S. Onoue, H. Hata, and K.-i. Matsumoto, "A study of the characteristics of developers' activities in github," in *Proceedings of the 2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, ser. APSEC '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 7–12. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2013.104

[29] R. Robbes, "Mining a change-based software repository," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 15–. [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.18

[30] M. Russell and M. Russell, *Mining the Social Web: Analyzing Data from Facebook, Twitter, LinkedIn, and Other Social Media Sites*, ser. Head First Series. O'Reilly Media, Incorporated, 2011. [Online]. Available: http://books.google.com.br/books?id=SYM1lrQdrdsC

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

[32] ——, "Recommending move method refactorings using dependency sets," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 232–241.

[33] R. Sanheim. (2013) Three million users. Acessado em 12/10/2014. [Online]. Available: https://github.com/blog/1382-three-million-users

[34] S. Schulze, M. Lochau, and S. Brunswig, "Implementing refactorings for fop: Lessons learned and challenges ahead," in *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, ser. FOSD '13. New York, NY, USA: ACM, 2013, pp. 33–40. [Online]. Available: http://doi.acm.org/10.1145/2528265.2528271

[35] J. Sheoran, K. Blincoe, E. Kalliamvakou, D. Damian, and J. Ell, "Understanding "watchers"on github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 336–339. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597114

[36] D. Silva, R. Terra, and M. T. Valente, "Recommending automated extract method refactorings," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 146–156. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597141

[37] ——, "Recommending automated extract method refactorings," in *22nd IEEE International Conference on Program Comprehension (ICPC)*, 2014, pp. 146–156.

[38] D. Steidl, B. Hummel, and E. Juergens, "Incremental origin analysis of source code files," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 42–51. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597111

[39] R. Stevens, C. De Roover, C. Noguera, and V. Jonckers, "A history querying tool and its application to detect multi-version refactorings," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, March 2013, pp. 335–338.

[40] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '13. Riverton, NJ, USA: IBM Corp., 2013, pp. 132–146. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555523.2555539

[41] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 233–243. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337251

[42] Y. Weicheng, S. Beijun, and X. Ben, "Mining github: Why commit stops – exploring the relationship between developer's commit pattern and file version evolution," in *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, vol. 2, Dec 2013, pp. 165–169.

[43] Y. Wu, J. Kropczynski, P. C. Shih, and J. M. Carroll, "Exploring the ecosystem of software developers on github and other platforms," in *Proceedings of the Companion Publication of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, ser. CSCW Companion '14. New York, NY, USA: ACM, 2014, pp. 265–268. [Online]. Available: http://doi.acm.org/10.1145/2556420.2556483

[44] Z. Xing and E. Stroulia, "Umldiff: An algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 54–65. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101919