RefDiff: Detecting Refactorings in Version Histories

Danilo Silva*, Marco Tulio Valente†
Department of Computer Science
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
Email: *danilofs@dcc.ufmg.br, †mtov@dcc.ufmg.br

***Abstract*—Refactoring is a well-known technique that is widely adopted by software engineers to improve the design and enable the evolution of a system. Knowing which refactoring operations were applied in a code change is a valuable information to understand software evolution, adapt software components, merge code changes, and other applications. In this paper, we present RefDiff, an automated approach that identifies refactorings performed between two code revisions in a git repository. RefDiff employs a combination of heuristics based on static analysis and code similarity to detect 13 well-known refactoring types. In an evaluation using an oracle of 448 known refactoring operations, distributed across seven Java projects, our approach achieved precision of 100% and recall of 88%. Moreover, our evaluation suggests that RefDiff has superior precision and recall than existing state-of-the-art approaches.**

***Keywords*-refactoring; software evolution; software repositories; git.**

I. Introduction

Refactoring is a well-known technique to improve the design of a system and enable its evolution [1]. In fact, existing studies [2]–[6] present strong evidences that refactoring is frequently applied by development teams, and it is an important aspect of their software maintenance workflow.

Therefore, knowing about the refactoring activity in a code change is a valuable information to help researchers to understand software evolution. For example, past studies have used such information to shed light on important aspects of refactoring practice, such as: how developers refactor [2], the usage of refactoring tools [2], [7], the motivations driving refactoring [4]–[6], the risks of refactoring [4], [5], [8]–[10], and the impact of refactoring on code quality metrics [4], [5]. Moreover, knowing which refactoring operations were applied in the version history of a system may help in several practical tasks. For example, in a study by Kim et al. [4], many developers mentioned the difficulties they face when reviewing or integrating code changes after large refactoring operations, which moves or renames several code elements. Thus, developers feel discouraged to refactor their code. If a tool is able to identify such refactoring operations, it can possibly resolve merge conflicts automatically. Moreover, diff visualization tools can also benefit from such information, presenting refactored code elements side-by-side with their corresponding version before the change. Another application for such information is adapting client code to a refactored version of an API it uses [11], [12]. If we are able to detect the refactorings that were applied to an API, we can replay them on the client code automatically.

Although there are approaches capable of detecting refactorings automatically, there are still some issues that hinder their application. Specifically, the precision and recall of such approaches still need improvements. In this paper, we try to fill this gap by proposing RefDiff, an automated approach that identifies refactorings performed in the version history of a system. RefDiff employs a combination of heuristics based on static analysis and code similarity to detect 13 well-known refactoring types. When compared to existing approaches, RefDiff leverages existing techniques and also introduces some novel ideas, such as the adaptation of the classical TF-IDF similarity measure from information retrieval to compare refactored code elements, and a new strategy to compare the similarity of fields by taking into account the similarity of the statements that reads from or writes to them.

In the paper, we also describe in details a study to evaluate the precision and recall of RefDiff and three existing refactoring detection approaches: Refactoring Miner [6], Refactoring Crawler [13], and Ref-Finder [14], [15]. In our study, RefDiff achieved precision of 100% and recall of 88%, which were the best results among the evaluated approaches.

In summary, the contributions we deliver in this work are:

- RefDiff, which is a new approach to detect refactoring in version histories. We provide a publicly available[1] implementation of our approach that is capable of finding refactorings in Java code within git repositories in a fully automated way;
- a publicly available oracle of 448 known refactoring operations, applied to seven Java systems, that serves as an evaluation benchmark for refactoring detection approaches; and
- an evaluation of the precision and recall of RefDiff, comparing it with three state-of-the-art approaches.

The remainder of this paper is structured as follows. Section II describes related work, focusing on the three approaches we compare with RefDiff. Section III presents the proposed approach in details. Section IV describes how we evaluated RefDiff and discusses the achieved results. Section V discusses threats to validity and we conclude the paper in Section VI.

[1] RefDiff and all evaluation data are public available in GitHub: https://github.com/aserg-ufmg/RefDiff

II. Related Work

Empirical studies on refactoring rely on means to identify refactoring activity. Thus, many different techniques have been proposed and employed for this task. For example, Murphy-Hill et al. [2] collected refactoring usage data using a framework that monitors user actions in the Eclipse IDE, including calls to refactoring commands. Negara et al. [7] also used the strategy of instrumenting the IDE to infer refactorings from fine-grained code edits. Other studies use metadata from version control systems to identify refactoring changes. For example, Ratzinger et al. [16] search for a predefined set of terms in commit messages to classify them as refactoring changes. In specific scenarios, a branch may be created exclusively to refactor the code, as reported by Kim et al. [5]. Another strategy is employed by Soares et al. [17]. They propose an approach that identify behavior-preserving changes by automatically generating and running test-cases. While their approach is intended to guarantee the correct behavior of a system after refactoring, it may also be employed to classify commits as behavior-preserving. Moreover, many existing approaches are based on static analysis. This is the case of the approach proposed by Demeyer et al. [18], which finds refactored elements by observing changes in code metrics.

Static analysis is also frequently used to find differences in the source code [3], [13]–[15], [19]. Approaches based on comparing source code differences have the advantage of beeing able to identify each refactoring operation performed. As RefDiff is one of these approaches, it can be directly compared with others within this category. In the next sections, we will describe three of such approaches.

A. Refactoring Miner

Refactoring Miner is an approach introduced by Tsantalis et al. [3], that was later extend by Silva et al. [6] to mine refactorings in large scale in git repositories. This tool is capable of identifying 14 high-level refactoring types: *Rename Package/Class/Method*, *Move Class/Method/Field*, *Pull Up Method/Field*, *Push Down Method/Field*, *Extract Method*, *Inline Method*, and *Extract Superclass/Interface*.

Refactoring Miner runs a lightweight algorithm, similar to the UMLDiff proposed by Xing and Stroulia [20], for differencing object-oriented models, inferring the set of classes, methods, and fields added, deleted or moved between two code revisions. First, the algorithm matches code entities in a top-down order (starting from the classes and going to the methods and fields) looking for exact matches on their names and signatures (in the case of methods). Next, the removed/added elements between the two models are matched based only on the equality of their names in order to find changes in the signatures of fields and methods. Third, the removed/added classes are matched based on the similarity of their members at signature level. Finally, a set of rules enforcing structural constraints is applied to identify specific types of refactorings.

In a first study, using the version histories of JUnit, HTTPCore, and HTTPClient, Tsantalis et al. [3] found 8 false positives for the *Extract Method* refactoring (96.4% precision) and 4 false positives for the *Rename Class* refactoring (97.6% precision). No false positives were found for the remaining refactorings. In a second study that mined refactorings in 285 GitHub hosted Java repositories, Silva et al. [6] found 1,030 false positives out of 2,441 refactorings (63% precision). However, the authors also evaluated Refactoring Miner using as a benchmark the dataset reported by Chaparro et al. [21], in which it achieved 93% precision and 98% recall.

B. Refactoring Crawler

Refactoring Crawler, proposed by Dig et al. [13], is an approach capable of finding seven high-level refactoring types: *Rename Package/Class/Method*, *Pull Up Method*, *Push Down Method*, *Move Method*, and *Change Method Signature*. It uses a combination of a syntactic analysis to detect refactoring candidates and a more expensive reference graph analysis to refine the results.

First, Refactoring Crawler analyzes the abstract syntax tree of a program and produces a tree, in which each node represents a source code entity (package, class, method, or field). Then, it employs a technique known as *shingles encoding* to find similar pairs of entities, which are candidates for refactorings. Shingles are representations for strings with the following property: if a string changes slightly, then its shingles also change slightly. In a second phase, Refactoring Crawler applies specific strategies for detecting each refactoring type, and computes a more costly metric that determines the similarity of references among code entities in the two versions of the system. For example, two methods are similar if the sets of methods that call them are similar, and the sets of methods they call are also similar. The strategies to detect refactorings are repeated in a loop until no new refactorings are found. Therefore, the detection of a refactoring, such as a rename, may change the reference graph of code elements and enable the detection of new refactorings.

The authors evaluated Refactoring Crawler comparing pairs of releases of three open source software components: Eclipse UI, Struts, and JHotDraw. Such components were chosen because they provided detailed release notes describing API changes. The authors relied on such information and on manual inspection to build an oracle of known refactorings in those releases, containing 131 refactorings in total. The reported results are: Eclipse UI (90% precision and 86% recall), Struts (100% precision and 86% recall), and JHotDraw (100% precision and 100% recall).

C. Ref-Finder

Ref-Finder, proposed by Prete et al. [14], [15], is an approach based on logic programming capable of identifying 63 refactoring types from the Fowler's catalog [1]. The authors express each refactoring type by defining structural constraints, before and after applying a refactoring to a program, in terms of template logic rules.

First, Ref-Finder traverses the abstract syntax tree of a program and extracts facts about code elements, structural dependencies, and the content of code elements, to represent the program in terms of a database of logic facts. Then, it uses a logic programming engine to infer concrete refactoring instances, by creating a logic query based on the constraints defined for each refactoring type. The definition of refactoring types also consider ordering dependencies among them. This way, lower-level refactorings may be queried to identify higher-level, composite refactorings. The detection of some types of refactoring requires a special logic predicate that indicates that the similarity between two methods is above a threshold. For this purpose, the authors implemented a block-level clone detection technique, which removes any beginning and trailing parenthesis, escape characters, white spaces and return keywords and computes word-level similarity between the two texts using the longest common sub-sequence algorithm.

The authors evaluated Ref-Finder in two case studies. In the first one, they used code examples from the Fowler's catalog to create instances of the 63 refactoring types. The authors reported 93.7% recall and 97.0% precision for this first study. In the second study, the authors used three open-source projects: Carol, jEdit, and Columba. In this case, Ref-Finder was executed in randomly selected pairs of versions. From the 774 refactoring instances found, the authors manually inspected a sample of 344 instances and found that 254 were correct (73.8% precision). However, in a study by Soares et al. [22] using a set of randomly select versions of JHotDraw and Apache Common Collections containing 81 refactoring instances in total, Ref-Finder achieved only 35% precision and 24% recall.

III. Proposed Refactoring Detection Algorithm

RefDiff employs a combination of heuristics based on static analysis and code similarity to detect refactorings between two revisions of a system. Thus, RefDiff takes as input two versions of a system, and outputs a list of refactorings found.

The detection algorithm is divided in two main phases: Source Code Analysis and Relationship Analysis. In the first phase, the source code of the system is parsed and analyzed to build a model that represents each high level source code entity, such as types, methods, and fields. Two models are built to represent the system before (E_b) and after the changes (E_a). For efficiency, only code entities that belong to modified source files (added, removed or edited) are analyzed. Each of these two models is a set of types, method, and fields contained in the source code. Specifically, $E_b = (T_b \cup M_b \cup F_b)$, such that T_b, M_b, and F_b are the sets of types, methods, and fields in the source code before the changes, and $E_a = (T_a \cup M_a \cup F_a)$, such that T_a, M_a, and F_a are the sets of types, methods, and fields after the changes.

The second phase of the algorithm, Relationship Analysis, consists in finding relationships between source code entities before and after the code changes. Specifically, the algorithm builds a bipartite graph with two sets of vertices: code entities before (E_b) and code entities after (E_a). The edges of this graph are represented by the set of relationships R between code entities. For example, a certain method $m_1 \in M_b$ may correspond to a method $m_2 \in M_a$ that was renamed by a developer. This would correspond to a *Rename Method* relationship between m_1 and m_2 and, consequently, to a *Rename Method* refactoring.

Table I presents all relationships that RefDiff can identify between types, methods, or fields. We search for relationships between source code entities considering each relationship type in the order they are presented in the table. The following sections detail how such relationships are identified.

A. Matching Relationships

Some kinds of relationships map code entities before the change to code entities after the change. For example, let $t_1 \in T_b$ be a type in the version before the change. If our algorithm finds another type $t_2 \in T_a$ with the same qualified name, it adds a relationship *Same Type* between t_1 and t_2 in R. This is a matching relationship, because t_1 corresponds to t_2 after the change. Other examples of matching relationship are *Move Type*, *Rename Type*, and *Pull Up Method*. In contrast, suppose that our algorithm finds that m_2 is a method that was extracted from another method m_1. In this case, there is an *Extract Method* relationship between m_1 and m_2, but this is not a matching relationship, because m_1 does not correspond to m_2 after the change. From this point on, we use the notation $e_1 \sim e_2$ to represent a matching relationship between e_1 and e_2.

We discriminate matching relationships from non-matching relationships because their detection algorithm is similar. For each matching relationship type, we find all pairs of entities $(e_b, e_a) \in E_b \times E_a$ that fall under the conditions specified in Table I. Each relationship type has its specific conditions. For example, as presented in Table I, the conditions for identifying a *Rename Method* between $m_1 \in M_b$ and $m_2 \in M_a$ are:

- the names of m_1 and m_2 should be different;
- there should exist a matching relationship between the container classes of m_1 and m_2; and
- the similarity index between m_1 and m_2, denoted by $\mathrm{sim}(m_1, m_2)$, should be greater than a threshold τ.

Whenever these conditions hold, we add the triple $(e_b, e_a, \mathrm{sim}(e_b, e_a))$ in a list of potential *Rename Method* relationships.

The last step to find the actual relationships consists in selecting non-conflicting relationships from the list of potential relationships and add them to the graph. For example, there may be in the list two potential *Rename Method* relationships: $(e_1, e_2, 0.5)$ and $(e_1, e_3, 0.8)$. However, a code entity can not be involved in more than one matching relationship. Thus, only one of them must be chosen, because e_1 could not be renamed to e_2 and to e_3. The criterion we use is to choose the triple with the higher similarity index. This means that, in the aforementioned example, we would choose the triple $(e_1, e_3, 0.8)$ and discard $(e_1, e_2, 0.5)$. In Section III-C we describe in details how the similarity index is computed.

TABLE I
RELATIONSHIP TYPES

Relationship	Condition
	$(t_b, t_a) \in T_b \times T_a$, such that:
Same Type	$\mathrm{name}(t_b) = \mathrm{name}(t_a) \wedge \pi(t_b) \sim \pi(t_a)$
Rename Type	$\mathrm{name}(t_b) \neq \mathrm{name}(t_a) \wedge \pi(t_b) \sim \pi(t_a) \wedge \mathrm{sim}(t_b, t_a) > \tau$
Move Type	$\mathrm{name}(t_b) = \mathrm{name}(t_a) \wedge \pi(t_b) \nsim \pi(t_a) \wedge \mathrm{sim}(t_b, t_a) > \tau$
Move and Rename Type	$\mathrm{name}(t_b) \neq \mathrm{name}(t_a) \wedge \pi(t_b) \nsim \pi(t_a) \wedge \mathrm{sim}(t_b, t_a) > \tau$
Extract Supertype	$(\nexists x \in T_b \mid x \sim t_a) \wedge (\exists y \in T_a \mid t_b \sim y \wedge \mathrm{subtype}(y, t_a)) \wedge \mathrm{sim}_\mathrm{p}(t_a, t_b) > \tau$
	$(m_b, m_a) \in M_b \times M_a$, such that:
Same Method	$\mathrm{sig}(m_b) = \mathrm{sig}(m_a) \wedge \pi(m_b) \sim \pi(m_a)$
Rename Method	$\mathrm{name}(m_b) \neq \mathrm{name}(m_a) \wedge \pi(m_b) \sim \pi(m_a) \wedge \mathrm{sim}(m_b, m_a) > \tau$
Change Method Signature	$\mathrm{name}(m_b) = \mathrm{name}(m_a) \wedge \mathrm{sig}(m_b) \neq \mathrm{sig}(m_a) \wedge \pi(m_b) \sim \pi(m_a) \wedge \mathrm{sim}(m_b, m_a) > \tau$
Pull Up Method	$\mathrm{sig}(m_b) = \mathrm{sig}(m_a) \wedge \mathrm{subtype}(\pi(m_b)^{\sim}, \pi(m_a)) \wedge \mathrm{sim}(m_b, m_a) > \tau$
Push Down Method	$\mathrm{sig}(m_b) = \mathrm{sig}(m_a) \wedge \mathrm{supertype}(\pi(m_b)^{\sim}, \pi(m_a)) \wedge \mathrm{sim}(m_b, m_a) > \tau$
Move Method	$\mathrm{name}(m_b) = \mathrm{name}(m_a) \wedge \pi(m_b) \nsim \pi(m_a) \wedge \neg\,\mathrm{subOrSuper}(\pi(m_b)^{\sim}, \pi(m_a)) \wedge \mathrm{sim}(m_b, m_a) > \tau$
Extract Method	$(\nexists x \in M_b \mid x \sim m_a) \wedge (\exists y \in M_a \mid m_b \sim y \wedge y \in \mathrm{callers}(m_a)) \wedge \mathrm{sim}_\mathrm{p}(m_a, m_b) > \tau$
Inline Method	$(\nexists x \in M_a \mid m_b \sim x) \wedge (\exists y \in M_b \mid y \sim m_a \wedge y \in \mathrm{callers}(m_b)) \wedge \mathrm{sim}_\mathrm{p}(m_b, m_a) > \tau$
	$(f_b, f_a) \in F_b \times F_a$, such that:
Same Field	$\mathrm{name}(f_b) = \mathrm{name}(f_a) \wedge \mathrm{type}(f_b) = \mathrm{type}(f_a) \wedge \pi(f_b) \sim \pi(f_a)$
Pull Up Field	$\mathrm{name}(f_b) = \mathrm{name}(f_a) \wedge \mathrm{type}(f_b) = \mathrm{type}(f_a) \wedge \mathrm{subtype}(\pi(f_b)^{\sim}, \pi(f_a)) \wedge \mathrm{sim}(f_b, f_a) > \tau$
Push Down Field	$\mathrm{name}(f_b) = \mathrm{name}(f_a) \wedge \mathrm{type}(f_b) = \mathrm{type}(f_a) \wedge \mathrm{supertype}(\pi(f_b)^{\sim}, \pi(f_a)) \wedge \mathrm{sim}(f_b, f_a) > \tau$
Move Field	$\mathrm{name}(f_b) = \mathrm{name}(f_a) \wedge \mathrm{type}(f_b) = \mathrm{type}(f_a) \wedge \pi(f_b) \nsim \pi(f_a) \wedge \neg\,\mathrm{subOrSuper}(\pi(f_b)^{\sim}, \pi(f_a)) \wedge \mathrm{sim}(f_b, f_a) > \tau$

$\mathrm{name}(e)$	simple name of a code entity e	$\pi(e)$	container entity of a code entity e (it may be a type or a package)
$\mathrm{sig}(m)$	signature of a method m	$e_1 \sim e_2$	exists a matching relationship between e_1 and e_2
$\mathrm{type}(f)$	type of a field f	$e_1 \nsim e_2$	does not exists a matching relationship between e_1 and e_2
$\mathrm{subtype}(t_1, t_2)$	t_1 is subtype of t_2	$e^{\sim}$	the code entity that matches with e after the change
$\mathrm{supertype}(t_1, t_2)$	t_1 is supertype of t_2	$\mathrm{callers}(m_a)$	the set of methods that call m_a
$\mathrm{subOrSuper}(t_1, t_2)$	$\mathrm{subtype}(t_1, t_2) \vee \mathrm{supertype}(t_1, t_2)$	$\mathrm{sim}(e_1, e_2)$	similarity index between e_1 and e_2
		$\mathrm{sim}_\mathrm{p}(e_1, e_2)$	similarity index between e_1 and e_2 for non-matching relationships

B. Non-matching Relationships

In the previous section, we discussed that an entity could not be involved in multiple matching relationships, but this property does not hold for non-matching relationships. For example, suppose that a developer extracted some code from a method m_1 into a new method m_2, i.e., an *Extract Method* refactoring was applied. It is also possible that the developer extracted another part of m_1 into a new method m_3.

Given that non-matching relationships do not conflict with each other, the algorithm to identify them is simpler. We just need to find all pairs of entities $(e_b, e_a) \in E_b \times E_a$ that fall under the conditions specified in Table I. For example, the conditions for identifying an *Extract Method* relationship between $m_1 \in M_b$ and $m_2 \in M_a$ are:

- there should not exist a method $x \in M_b$ such that $x \sim m_2$ (i.e., m_2 was added);
- there should exist a method $y \in M_a$ such that $m_1 \sim y$ (i.e., m_1 was not removed);
- y should call m_2; and
- the similarity index between m_2 and m_1, denoted by $\mathrm{sim}_\mathrm{p}(m_2, m_1)$, should be greater than a threshold τ.

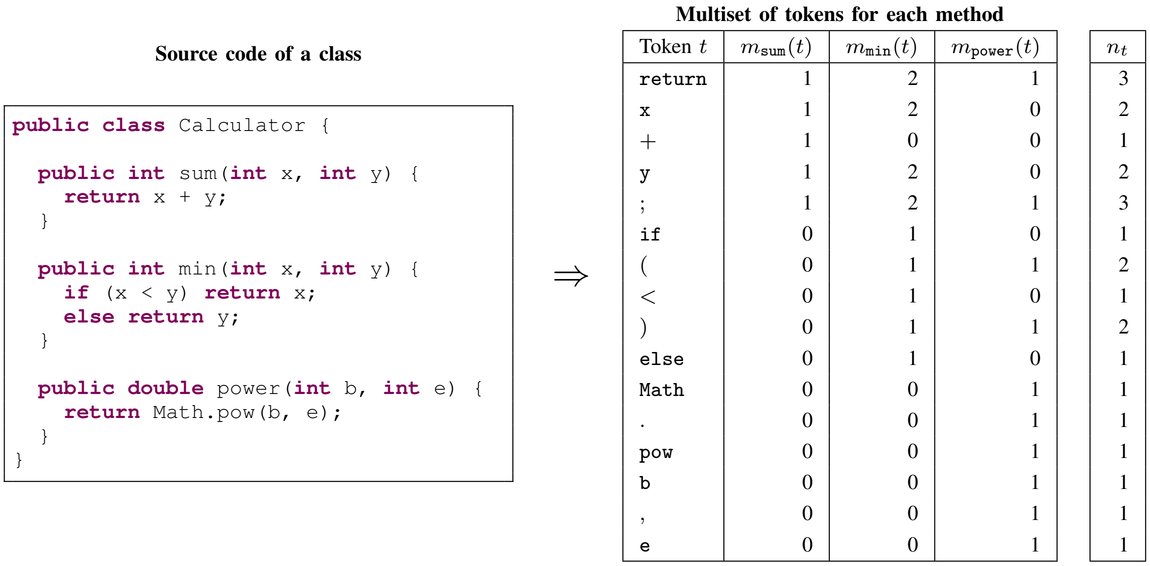
Besides *Extract Method*, our approach supports the detection of *Inline Method* and *Extract Supertype* relationships.

C. Computing Similarity

A key element of our algorithm to find relationships, as mentioned previously, is computing the similarity between entities. The first step to compute similarity of code entities is to represent their source code as a multiset (or bag) of tokens. A multiset is a generalization of the concept of a set, but it allows multiple instances of the same element. The multiplicity of an element is the number of occurrences of that element within the multiset. Formally, a multiset can be defined in terms of a multiplicity function $m : U \to \mathbb{N}$, where U is the set of all possible elements. In other words, $m(t)$ is the multiplicity of the element t in the multiset. Note that the multiplicity of an element that is not in the multiset is zero.

For example, Figure 1 depicts the transformation of the source code of three methods (`sum`, `min`, and `power`), of the class `Calculator`, into multisets of tokens. In the Figure, the multiplicity function m for each method is represented in a tabular form. For example, the multiplicity of the token `y` in method `min` is two (i.e., $m_{\texttt{min}}(\texttt{y}) = 2$), whilst the multiplicity of the token `if` in method `power` is zero (i.e., $m_{\texttt{power}}(\texttt{if}) = 0$).

Later, to compute the similarity between two source code entities e_1 and e_2, we use a generalization of the Jaccard



Source code of a class

```
public class Calculator {

  public int sum(int x, int y) {
    return x + y;
  }

  public int min(int x, int y) {
    if (x < y) return x;
    else return y;
  }

  public double power(int b, int e) {
    return Math.pow(b, e);
  }
}
```

⇒

Multiset of tokens for each method

Token t	$m_{\texttt{sum}}(t)$	$m_{\texttt{min}}(t)$	$m_{\texttt{power}}(t)$	n_t
`return`	1	2	1	3
`x`	1	2	0	2
+	1	0	0	1
`y`	1	2	0	2
;	1	2	1	3
`if`	0	1	0	1
(	0	1	1	2
<	0	1	0	1
)	0	1	1	2
`else`	0	1	0	1
`Math`	0	0	1	1
.	0	0	1	1
`pow`	0	0	1	1
`b`	0	0	1	1
,	0	0	1	1
`e`	0	0	1	1

Fig. 1. Transformation of the body of each method into a multiset of tokens

coefficient, known as weighted Jaccard coefficient [23]. Let U be the set of all possible tokens and $w(e, t)$ be a weight function of a token t for the entity e. We define the similarity between e_1 and e_2 by the following formula:

$$\text{sim}(e_1, e_2) = \frac{\sum_{t \in U} \min(w(e_1, t), w(e_2, t))}{\sum_{t \in U} \max(w(e_1, t), w(e_2, t))} \qquad (1)$$

1) Weight of a token for a code entity: Our similarity function is based on a weighting function $w(e, t)$ that expresses the importance a token t for a code entity e. In fact, some tokens are more important than others to discriminate a code element. For example, in Figure 1, all three methods contain the token `return`. In contrast, only one method (`power`) contains the token `Math`. Therefore, the later is a better indicator of similarity between methods than the former.

In order to take this into account, we employ a variation of the TF-IDF weighting scheme [24], which is a well-known technique from information retrieval. TF-IDF, which is the short form of *Term Frequency–Inverse Document Frequency*, reflects how important a term is to a document within a collection of documents. In the context of code entities, we consider a token as a term, and the body of a method (or class) as a document.

Let E be the set of all code entities and n_t be the number of entities in E that contains the token t, we define the weight of t for a code entity e as the function $w(e, t)$, which is defined by the following formula:

$$w(e, t) = m_e(t) \times \mathit{idf}(t) \qquad (2)$$

where $m_e(t)$ is the multiplicity of t in e, and $\mathit{idf}(t)$ is the Inverse Document Frequency, which is defined as:

$$\mathit{idf}(t) = \log(1 + \frac{|E|}{n_t}) \qquad (3)$$

Note that the value of $\mathit{idf}(t)$ decreases as n_t increases, because the more frequent a token is among the collection of code entities, the less important it is to distinguish code elements. For example, in Figure 1, the token `y` occurs in two methods (`sum` and `min`). Thus, its idf is:

$$\mathit{idf}(\texttt{y}) = \log(1 + \frac{|E|}{n_t}) = \log(1 + \frac{3}{2}) = 0.398$$

On the other hand, the token `else` occurs in one method (`min`), and its idf is:

$$\mathit{idf}(\texttt{else}) = \log(1 + \frac{|E|}{n_t}) = \log(1 + \frac{3}{1}) = 0.602$$

2) Similarity of fields: The similarity of types and methods can be directly computed by the aforementioned similarity function by scanning the source code within their bodies and building the multiset of tokens. However, such strategy is not suitable to compute the similarity of fields, because they do not have a body. To address this limitation, we defined the concept of a virtual body of a field, which is composed of all statements that access the field (read or write) found in the source code of the system. Thus, we are able to compute the multiset of tokens for a field f_1 by extracting the tokens of all statements that access f_1. The rationale behind such strategy is that if a field f_1 corresponds to a field f_2 after a change, the statements that directly used f_1 should use f_2 after the change, thus, they would likely be similar.

3) Similarity for non-matching relationships: While the similarity function presented previously is suitable to compute whether the source code of two entities are similar, in some situations we need to assess whether the source code of an entity is contained within another one. This is the case of *Extract Supertype*, *Extract Method*, and *Inline Method* relationships. For example, if a method m_2 is extracted from m_1, the source code of m_2 should be contained within m_1 prior to the refactoring, although m_1 and m_2 may be significantly different from each other. Analogously, if a method m_1 is inlined into m_2, the source code of m_1 should be contained within m_2. Thus, we defined a specialized version of the similarity function $\mathrm{sim_p}$ defined by the following formula:

$$\mathrm{sim_p}(e_1, e_2) = \frac{\sum_{t \in U} \min(w(e_1, t), w(e_2, t))}{\sum_{t \in U} w(e_1, t)} \qquad (4)$$

The rationale behind this formula is that the similarity is at maximum when the multiset of tokens of e_1 is a subset of the multiset of tokens of e_2, i.e., all tokens from e_1 can be found in e_2. Note that, given this definition, $\mathrm{sim_p}(e_1, e_2) \neq \mathrm{sim_p}(e_2, e_1)$.

D. *Calibration of similarity thresholds*

Our algorithm relies on thresholds to find relationships between entities, as discussed in Section III-A and Section III-B. Specifically, for each relationship type, we define a threshold τ for the minimum similarity that the involved entities should have so that we consider them as a potential relationship. Therefore, the thresholds we choose may affect the precision an recall of the algorithm. For this reason, we selected such thresholds by applying a well-defined calibration process.

First, we randomly selected a set of ten commits that contain refactorings, in ten different projects (see Table II), drawn from a public dataset used to investigate the reasons for refactoring operations [6]. We ensured that every refactoring type is covered by at least one commit. All refactorings reported in those commits were initially added to an oracle of known refactorings. Later, for each refactoring type, we run our algorithm using different thresholds values, ranging from 0.1 to 0.9 by 0.1 increments. The output of our algorithm (i.e., the refactorings found) were then compared to the known refactorings from the oracle. Refactorings contained in our oracle were initially classified as true positives, whilst refactorings not contained in our oracle were classified as false positives. Moreover, refactorings in our oracle that were not found were classified as false negatives. In a second pass, the false positive refactorings were manually inspected to find potential true positives incorrectly classified. This step was necessary because the oracle obtained in the first step may not contain all refactorings in a commit.

Last, we selected the threshold value by choosing the one that yields the best compromise between precision and recall. Specifically, we choose the value that optimize the F_1 score, which is defined as:

$$F_1 = 2 \times \frac{\mathrm{precision} \times \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} \qquad (5)$$

where precision and recall are respectively:

$$\mathrm{precision} = \frac{\mathrm{tp}}{\mathrm{tp} + \mathrm{fp}} \qquad \mathrm{recall} = \frac{\mathrm{tp}}{\mathrm{tp} + \mathrm{fn}} \qquad (6)$$

The set of threshold values chosen are presented in the third column of Table III (τ). It is worth noting that the threshold calibration for each relationship type was performed in the order presented in Table III, to minimize the effect of dependencies between different relationship types. For example, suppose that an existing *Move Class* refactoring is not identified. It is likely that false positive *Move Method* instances will be reported, because there should be some similar (or identical) methods when comparing the moved class with itself after the move operation. Therefore, it is important to calibrate the *Move Type* threshold before the *Move Method* threshold.

TABLE II
PROJECTS/COMMITS USED INT THE CALIBRATION

Repository URL	Commit
github.com/linkedin/rest.li	54fa890
github.com/droolsjbpm/jbpm	3815f29
github.com/gradle/gradle	44aab62
github.com/jenkinsci/workflow-plugin	d0e374c
github.com/spring-projects/spring-roo	0bb4cca
github.com/BuildCraft/BuildCraft	a5cdd8c
github.com/droolsjbpm/drools	1bf2875
github.com/jersey/jersey	d94ca2b
github.com/undertow-io/undertow	d5b2bb8
github.com/kuujo/copycat	19a49f8

TABLE III
THRESHOLDS CALIBRATION RESULTS

Ref. Type	#	τ	TP	FP	FN	Precision	Recall
Rename Type	2	0.4	2	0	0	1.000	1.000
Move Type	2	0.9	2	0	0	1.000	1.000
Extract Superclass	2	0.8	2	0	0	1.000	1.000
Rename Method	24	0.3	22	3	2	0.880	0.917
Pull Up Method	7	0.4	7	0	0	1.000	1.000
Push Down Method	2	0.6	2	0	0	1.000	1.000
Move Method	24	0.4	21	1	3	0.955	0.875
Extract Method	25	0.1	25	9	0	0.735	1.000
Inline Method	6	0.3	5	2	1	0.714	0.833
Pull Up Field	2	0.5	2	0	0	1.000	1.000
Push Down Field	5	0.3	5	0	0	1.000	1.000
Move Field	1	0.5	1	1	0	0.500	1.000
Total	102		96	16	6	0.857	0.941

Table III also presents the number of entries (#) in the oracle for each refactoring relationship type, and the results we achieved using the optimal thresholds devised from the calibration process. The four rightmost columns show, respectively, the number of true positives (TP), the number of false positives (FP), the number of false negatives (FN), the precision and the recall. In total, 85.7% of the refactoring relationships reported by RefDiff were correct (precision) and it was able to find 94.1% of the known refactorings (recall).

IV. EVALUATION

A. *Precision and Recall*

To evaluate precision and recall, we compared the output of RefDiff with an oracle of known refactoring instances, simi-

TABLE IV
SELECTED PROJECTS

Repository URL	Description	LOC
github.com/Atmosphere/atmosphere	The Asynchronous WebSocket/Comet Framework	65,841
github.com/clojure/clojure	The Clojure programming language	58,417
github.com/google/guava	Google Core Libraries for Java 6+	374,068
github.com/dropwizard/metrics	Capturing JVM- and application-level metrics, so you know what's going on	24,242
github.com/orientechnologies/orientdb	An Open Source NoSQL DBMS with the features of both Document and Graph DBMSs	168,924
github.com/square/retrofit	Type-safe HTTP client for Android and Java by Square, Inc.	17,073
github.com/spring-projects/spring-boot	Spring Boot makes it easy to create Spring-powered, production-grade applications and services with absolute minimum fuss	39,190

larly to the calibration procedure described in Section III-D. Besides, we compared our tool with three existing approaches, namely Refactoring Miner [6], Refactoring Crawler [13], and Ref-Finder [15].

1) Construction of the oracle: In the calibration procedure (Section III), we used an oracle containing refactoring instances found on commits from open-source software repositories. This strategy has the advantage of using real refactoring instances, but it also has a drawback. There are no practical means of assuring that the oracle contains all existing refactoring instances in a given commit. In many cases, a single commit changes several files in non trivial ways, and a manual inspection of all changes using diff tools is time-consuming and error-prone. Thus, refactoring instances might be missed by the tool under evaluation and also by the oracle. Therefore, the computation of recall may not be reliable.

To be able to reliably compute recall, we employed the strategy of building an evaluation oracle by deliberately applying refactoring in software repositories in a controlled manner, similarly to Chaparro et al. [21]. Such refactorings were applied by graduate students of a Software Architecture course. First, we randomly selected a list of 20 GitHub hosted Java repositories from the dataset of Silva et al. [6] that contained a Maven project file (`pom.xml`). This way, we could use the Maven tool to build the project and import its source code to Eclipse IDE. Then, the professor of the course (an author of this paper) asked the students to:

1) Choose one of the 20 Java repositories in the list, given the constraint that a repository could not be taken by two students.
2) Analyze the latest revision of the source code, apply a specified number of refactoring operations on it, and commit the changes. The students were instructed to apply at least three refactorings of each refactoring type listed in Table V.
3) Document all refactoring operations applied in a spreadsheet, using a specified format.

It is worth noting that refactoring operations documented by them were confirmed by the first author of the paper by inspecting the source code. In this step, minor mistakes and typos were fixed. For example, in some cases students typed the name of a class or method incorrectly. There were also a few cases of refactorings actually applied that were not reported in the spreadsheet. For example, a student inlined a method into the body of two other methods, but incorrectly reported only one of them in the spreadsheet.

TABLE V
REFACTORING TYPES IN THE EVALUATION ORACLE

		Supported by			
Ref. Type	#	RDiff	RMinr	RCraw	RFind
Rename Type	35	yes	yes	yes	no
Move Type	31	yes	yes	no	no
Extract Superclass	16	yes	yes	no	yes
Rename Method	70	yes	yes	yes	yes
Pull Up Method	15	yes	yes	yes	yes
Push Down Method	68	yes	yes	yes	yes
Move Method	31	yes	yes	yes	yes
Extract Method	29	yes	yes	no	yes
Inline Method	52	yes	yes	no	yes
Pull Up Field	33	yes	yes	no	yes
Push Down Field	42	yes	yes	no	yes
Move Field	26	yes	yes	no	yes
Total	448				

By the end of the deadline, seven students properly completed the tasks and applied the refactorings in the repositories listed in Table IV. Note that the repositories contain relevant Java projects such as Google Guava, Spring Boot and OrientDB. Table IV also presents a short description of the project and the number of lines of Java code within each repository.

In total, we included 448 refactoring relationships in the evaluation oracle, as presented in Table V, covering 12 well-known refactoring types. Note that a refactoring operation may be represented by more than one refactoring relationship. For example, a method m may be extract from both x and y. In this case, the oracle would contain the relationships $\mathit{ExtractMethod}(x, m)$ and $\mathit{ExtractMethod}(y, m)$.

2) Execution of the selected approaches: After constructing the evaluation oracle, we run RefDiff, Refactoring Miner (1.0.0), Refactoring Crawler (1.0.0), and Ref-Finder (1.0.4) to compare their output with the refactoring relationships in the oracle. Refactoring Miner can be used as an API, and it provides mechanisms to export its output, thus, we only needed to transform it into a normalized format. Refactoring Miner and Refactoring Crawler are plug-ins that depend on

TABLE VI
Precision and recall by refactoring type

	RDiff		RMinr		RCraw		RFind	
Ref. Type	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Rename Type	1.000	1.000	1.000	1.000	0.750	0.429		
Move Type	1.000	0.968	1.000	0.968				
Extract Superclass	1.000	0.875	1.000	0.875			0.484	0.938
Rename Method	1.000	0.943	1.000	0.886	0.971	0.486	0.868	0.843
Pull Up Method	1.000	0.600	1.000	0.733	0.500	0.067	1.000	0.571
Push Down Method	1.000	0.971	1.000	0.176	1.000	0.265	1.000	0.491
Move Method	1.000	1.000	1.000	0.742	0.090	0.323	0.054	0.759
Extract Method	1.000	0.897	1.000	0.862			0.607	0.586
Inline Method	1.000	0.981	1.000	0.423			0.917	0.688
Pull Up Field	1.000	0.576	1.000	0.970			1.000	0.394
Push Down Field	1.000	0.929	1.000	0.929			1.000	0.333
Move Field	1.000	0.269	0.583	0.808			0.097	0.923

Eclipse IDE. In both cases, we needed to adapt their source code to enable or to facilitate the evaluation. We faced an issue to run Refactoring Crawler on the selected projects, because it was dependent on an outdated version of the Eclipse IDE, in which we were unable to import the projects through the Maven-Eclipse integration. To resolve such issue, we decided to adapt the source code of Refactoring Crawler to a recent Eclipse release (Mars). The necessary code modifications were simple, but, as a precaution, we assessed whether the results of our modified version of the tool were identical to those achieved by the original implementation using the evaluation dataset provided by the authors. In the case of Ref-Finder, we also had to modify its source code, but the reason was to be able to export its output into a text file.

Another issue we faced with Ref-Finder was related with refactorings that involved methods, because Ref-Finder only displays the name of the method and its class, but not its complete signature. Therefore, when there are overloaded methods in a class, Ref-Finder's output is ambiguous. To overcome this issue, we adopted a less strict check that ignores method parameters when comparing Ref-Finder's output with entries in the oracle. For example, if that the oracle contains the entry:

$\mathit{ExtractMethod}(\texttt{Calc.mult}(\texttt{int},\texttt{int}),\texttt{Calc.add}(\texttt{int},\texttt{int}))$

and Ref-Finder reports:

$\mathit{ExtractMethod}(\texttt{Calc.mult},\texttt{Calc.add})$

we still consider it a true positive.

Last, it is worth noting that the refactoring types contained in the oracle are not supported by all approaches, as detailed in Table V. For example, Refactoring Crawler does not support the detection of *Move Attribute* refactorings. We decided to disregarded such entries of the oracle when counting the number of false negatives. This means that an approach may achieve 1.0 recall even if it does not support all refactoring types in the oracle.

3) Results and discussion: The overall precision and recall for each approach are presented in Table VII. RefDiff achieves the best precision (1.000), followed by Refactoring Miner (0.956), Refactoring Crawler (0.419), and Ref-Finder (0.264). In terms of recall, RefDiff still holds the best result (0.877), followed by Refactoring Miner (0.728), Ref-Finder (0.642), and Refactoring Crawler (0.356).

TABLE VII
Overall precision and recall

Approach	TP	FP	FN	Precision	Recall
RDiff	393	0	55	1.000	0.877
RMinr	326	15	122	0.956	0.728
RCraw	78	108	141	0.419	0.356
RFind	231	645	129	0.264	0.642
RCraw*	78	56	141	0.582	0.356
RFind*	231	241	129	0.489	0.642

Detailed precision and recall results for each refactoring type are presented in Table VI. We can note that the results for some refactoring types stand out from the rest. For example, RefDiff achieved a recall of only 0.269 for *Move Field*. This observation suggests that the threshold for such refactoring could possibly be less restrictive. For Refactoring Miner, the main offender in terms of recall is *Push Down Method*.

When we focus on Refactoring Crawler and Ref-Finder, one fact that clearly draws one's attention is the extremely low precision for *Move Method* and *Move Field*. A more detailed analysis revealed that one reason for this was the lack of *Move Type* and/or *Rename Type* detection support in these approaches. For example, in the case of a class A is moved to become A', several *Move Method* and *Move Field* relationships from members of class A to class A' are mistakenly reported. This issue drastically affects the precision of such approaches. For example, 284 (74%) out of 382 *Move Method* false positives reported by Ref-Finder are due to this reason. Thus, we decided to recompute the overall precision for Refactoring Crawler and Ref-Finder disregarding false positives that fell in that scenario. The last two lines of

TABLE VIII
EXECUTION TIME

		RDiff execution time				RMinr execution time			
Repository	Commits	Min. (ms)	Max. (ms)	Avg. (ms)	Total. (s)	Min. (ms)	Max. (ms)	Avg. (ms)	Total. (s)
androidannotations/androidannotations	29	1	4,956	451	13	1	1,753	211	6
bumptech/glide	41	1	3,349	594	24	2	8,992	466	19
elastic/elasticsearch	946	1	42,344	1,897	1,795	1	103,943	1,105	1,046
libgdx/libgdx	69	0	5,112	805	56	1	6,774	578	40
netty/netty	225	0	3,384	640	144	0	59,736	665	150
PhilJay/MPAndroidChart	14	1	816	245	3	1	310	79	1
ReactiveX/RxJava	120	1	810,744	10,475	1,257	1	17,369	538	65
spring-projects/spring-framework	478	1	15,019	1,205	576	1	6,133	920	440
square/okhttp	45	1	1,526	380	17	1	616	178	8
zxing/zxing	23	1	773	342	8	1	502	230	5
Total	1990	0	810,744	1,956	3,893	0	103,943	894	1,779

Table VII presents the recomputed results for both tools, under the names of *RCraw** and *RFind**. We can note a significant improvement in precision for Refactoring Crawler (from 0.419 to 0.582) and Ref-Finder (from 0.264 to 0.489). However, even in this scenario, RefDiff and Refactoring Miner results are still far ahead from them. We should also note that our results corroborate with the findings of Soares et al. [22] in a study with JHotDraw and Apache Common Collections, in which Ref-Finder achieved precision of 0.35 and recall of 0.24.

It is interesting to note that the precision achieved in the calibration process was inferior to the one achieved in the evaluation, specially considering that the thresholds were optimized to that data. However, such behavior is not surprising, because the calibration oracle is composed of real refactorings performed in those systems, possibly interleaved with all kinds of code changes. Such scenario is undoubtedly more challenging for refactoring detection tools than refactoring-only commits.

B. Execution Time

Besides evaluating precision and recall, we also designed a study to evaluate the execution time and scalability of RefDiff and Refactoring Miner, in the context of mining refactorings from open-source software repositories. Ref-Finder and Refactoring Crawler were removed from the comparison because they are Eclipse-based plug-ins, which are not suitable for automation. Specifically, there are two issues that hinder their application. First, they require user interaction through the Eclipse UI to select their input and trigger the refactoring detection. Second, they require each pair of versions under comparison to be imported and configured as Eclipse projects. Thus, such tasks cannot be reliably automated. In contrast, both RefDiff and Refactoring Miner are able to detect refactorings directly from commits in git repositories, comparing the revisions of the source code before and after the changes.

To run the study, we selected the ten most popular Java repositories from GitHub that met the following criteria: (i) the repository was not used in the studies from Section IV-A and Section III; (ii) the repository contains a real software component (i.e., not a toy example or tutorial); (iii) the repository contains at least 1,000 commits; and (iv) the repository contains commits pushed in the last three months. Then, we employed RefDiff and Refactoring Miner to analyze each commit found in the default branch of the repositories, in a time window ranging from January 1, 2017 to March 27, 2017. For simplicity, merge commits were excluded from the analysis, as the code changes they contain are usually devised from other commits.

Table VIII shows the selected repositories, the number of analyzed commits, the execution time (minimum, maximum, and average) per commit, and the total execution time, for each approach. On average, RefDiff spends 1.96 seconds to detect refactorings in a commit, while Refactoring Miner spends 0.89 seconds. The total execution time to analyze the data set was 3,893 seconds (about 64 min) for RefDiff, against 1,779 seconds (about 29 min) for Refactoring Miner. In the worst case, RefDiff spent 810 seconds (about 13 minutes) in single commit. However, such case happened only in a commit from ReactiveX/RxJava. For all other repositories, the worst execution time was less than a minute. Figure 2 shows a box plot of the distribution of execution time per commit (omitting outliers for readability). We can note that the median is similar for both approaches (close to one second), with a slight advantage to Refactoring Miner. It is clear from the results that Refactoring

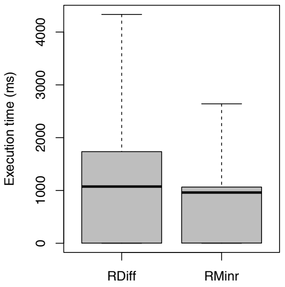


Fig. 2. Boxplot of execution time per commit (ommiting outliers)

Miner achieves lower execution time in most cases, but such differences are relatively small in practice. Thus, the potential gain in precision and recall using RefDiff may still be worth it. Besides, the implementation of RefDiff had as its main objective the evaluation of the approach. Thus, it is possible that its source code can be optimized. In summary, we can conclude that both approaches provide acceptable performance and scalability, enabling their application in large code bases, such as Elasticsearch (922 KLOC) and Spring Framework (1,016 KLOC).

V. Threats to Validity

External validity: The evaluation of precision and recall of RefDiff used refactoring instances injected in seven popular open-source Java projects. We cannot claim that the precision and recall of our approach would be the same for different projects, with distinct characteristics, and with actual refactorings applied by developers. However, such setup was necessary to compute recall, as discussed in Section IV-A1. Besides, the results we achieved in the calibration process (precision of 85.7% and recall of 94.1%), in which we used actual commits from relevant Java repositories, suggest that RefDiff's precision and recall are acceptable in real scenarios. Nevertheless, we plan to extend this study by assessing the precision of RefDiff in a large corpus of commits from open-source repositories.

Internal validity: The evaluation oracle we used in our study is subject to human errors due to the manual task of applying the refactorings and documenting them. However, we addressed that issue by inspecting the source code of the refactored systems to validate all documented refactorings before running our experiment. Besides, the procedures to compare the output of each tool with the entries in the oracle were automated to avoid any mistake.

VI. Conclusion

In this paper, we propose RefDiff, an approach to detect refactorings in version histories of software components. Our approach employs a combination of heuristics based on static analysis and code similarity to detect 13 well-known refactoring types. One key aspect of our algorithm is the employed similarity index, which is an adaptation of the TF-IDF weighting scheme. We have also evaluated RefDiff, comparing it with Refactoring Miner, Refactoring Crawler, and Ref-Finder, using on oracle of 448 known refactorings across seven Java projects. RefDiff achieved the best result among the evaluated tools: precision of 1.00 and recall of 0.88. As an additional contribution, we made publicly available the implementation of RefDiff and all data used in the experiments.

As future work, we intend to explore applications of RefDiff. For example, we could use information of refactorings to build an improved code diff visualization that presents changes in refactored code elements side-by-side with their matching elements in the previous version. Besides, a reliable refactoring detection tool open up possibilities for novel empirical studies on refactoring practices, taking advantage of the vast amount of historical information available in code repositories.

Acknowledgment

Our research is supported by FAPEMIG and CNPq.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] E. R. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[3] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 2013, pp. 132–146.

[4] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *20th Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 50:1–50:11.

[5] ——, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, July 2014.

[6] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of GitHub contributors," in *24th Symposium on the Foundations of Software Engineering (FSE)*, 2016, pp. 858–870.

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

[8] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of API-level refactorings during software evolution," in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 151–160.

[9] P. Weißgerber and S. Diehl, "Are refactorings less error-prone than other changes?" in *3rd Workshop on Mining Software Repositories (MSR)*, 2006, pp. 112–118.

[10] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *12th Conference on Source Code Analysis and Manipulation (SCAM)*, 2012, pp. 104–113.

[11] J. Henkel and A. Diwan, "Catchup!: capturing and replaying refactorings to support API evolution," in *27th International Conference on Software Engineering (ICSE)*, 2005, pp. 274–283.

[12] Z. Xing and E. Stroulia, "The JDEvAn tool suite in support of object-oriented evolutionary development," in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 951–952.

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

[14] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *26th International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

[15] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-Finder: A refactoring reconstruction tool based on logic query templates," in *8th Symposium on Foundations of Software Engineering (FSE)*, 2010, pp. 371–372.

[16] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction," in *5th Working Conference on Mining Software Repositories (MSR)*, 2008, pp. 35–38.

[17] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE software*, vol. 27, no. 4, pp. 52–57, 2010.

[18] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *ACM SIGPLAN Notices*, vol. 35, no. 10, 2000, pp. 166–177.

[19] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *21st International Conference on Automated Software Engineering (ASE)*, 2006, pp. 231–240.

[20] Z. Xing and E. Stroulia, "UMLDiff: An algorithm for object-oriented design differencing," in *20th International Conference on Automated Software Engineering (ASE)*, 2005, pp. 54–65.

[21] O. Chaparro, G. Bavota, A. Marcus, and M. Di Penta, “On the impact of refactoring operations on code quality metrics,” in *30th International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 456–460.

[22] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, “Comparing approaches to analyze refactoring activity on software repositories,” *Journal of Systems and Software*, vol. 86, no. 4, pp. 1006–1022, Apr. 2013.

[23] F. Chierichetti, R. Kumar, S. Pandey, and S. Vassilvitskii, “Finding the jaccard median,” in *21st Symposium on Discrete Algorithms (SODA)*, 2010, pp. 293–311.

[24] G. Salton and M. J. McGill, *Introduction to modern information retrieval*. McGraw-Hill, 1986.