11-2014

Vector Abstraction and Concretization for Scalable Detection of Refactorings

Narcisa Andreea MILEA
National University of Singapore

Lingxiao JIANG
Singapore Management University, lxjiang@smu.edu.sg

Siau-Cheng KHOO
National University of Singapore

Vector Abstraction and Concretization for Scalable Detection of Refactorings

Narcisa Andreea Milea
School of Computing
National University of Singapore
mileanar@comp.nus.edu.sg

Lingxiao Jiang
School of Information Systems
Singapore Management University
lxjiang@smu.edu.sg

Siau-Cheng Khoo
School of Computing
National University of Singapore
khoosc@nus.edu.sg

ABSTRACT

Automated techniques have been proposed to either identify refactoring opportunities (i.e., code fragments that can be but have not yet been restructured in a program), or reconstruct historical refactorings (i.e., code restructuring operations that have happened between different versions of a program). In this paper, we propose a new technique that can detect *both refactoring opportunities and historical refactorings* in large code bases. The key of our technique is the design of *vector abstraction and concretization* operations that can encode code changes induced by certain refactorings as characteristic vectors. Thus, the problem of identifying refactorings can be reduced to the problem of identifying matching vectors, which can be solved efficiently. We have implemented our technique for Java. The prototype is applied to 200 bundle projects from the Eclipse ecosystem containing 4.5 million lines of code, and reports in total more than 32K instances of 17 types of refactoring opportunities, taking 25 minutes on average for each type. The prototype is also applied to 14 versions of 3 smaller programs (JMeter, Ant, XML-Security), and detects (1) more than 2.8K refactoring opportunities within individual versions with a precision of about 87%, and (2) more than 190 historical refactorings across consecutive versions of the programs with a precision of about 92%.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

General Terms: Algorithms, Design, Experimentation, Performance

Keywords: Refactoring Detection, Software Evolution, Vector-Based Code Representation

1. INTRODUCTION

Software development and maintenance tasks often need to change the structure of code without changing the functionality of the code. This kind of code changes are often called refactoring, and have long been recognized as an important way to improve the design of existing code [9, 35], making code easier to understand, maintain, adapt to new requirements. Detecting refactoring has been a topic of long lasting interest in the literature. Some of the studies aim to detect *refactoring opportunities*, i.e., code fragments that can, be but have not yet been restructured, and thus to reduce "bad smells" in code and improve the design of the code [7, 20, 31, 56, 57]; some studies focus on understanding *historic refactorings* that have happened; they reconstruct the refactoring operations used to restructure the code by analyzing different versions of a program to facilitate code maintenance and evolution studies [4, 5, 18, 27, 46, 52, 55, 60].

It is desirable to have an approach that provides scalable, consistent detection of both refactoring opportunities and historic refactorings, as it can enable developers to measure more accurately their refactoring efforts and progresses during software evolution. Figure 1 illustrates some challenges in detecting refactoring.[1] The first challenge is how to identify that code fragment (a) may be refactored? One naive way is to check (a)'s code pattern against every type of refactorings to see whether it may match one refactoring type. Such checks may identify many meaningless refactorings since certain refactorings may in fact be applied to any code. Then, we may ask a more specific question: how can we identify a *likely* refactoring type for (a)? An idea is to utilize refactorings that have happened: we can try to identify another piece of code c that has been refactored yet is similar to (a) before it is refactored, and then we can have high confidence in saying that (a) may be refactored in the same way as c. In this example, the question may become how to decide (a) is similar to a supposedly "before-refactoring" version of either code fragments (b) or (c) shown in the figure?

This can be challenging as well. The code fragments (a) and (b) in Figure 1 look similar to each other, but (a) contains extra variable declarations (the underlined red parts) and a method call to `instantiate` (the bold part); they may not be detected as similar code (a.k.a. code clones [8, 23, 25, 28, 58]) or refactoring, unless relaxed similarity conditions are used. However, relaxed similarities can lead to many false clones to be detected and thus imprecise refactoring detection.

We can see that (c) may in fact be more similar to (a) than (b) if its call to the method `getVector` is inlined; i.e., by replacing the call to `getVector` with the actual method body from (b) and removing one of the `return` statements, the inlined code (denoted as (c^I)) becomes syntactically

FSE '14, November 16–22, 2014, Hong Kong, China

[1] The code fragments (a), (b), and (c) were detected by our approach as a refactoring opportunity in a program named JMeter version 1, and still exist (with small variants) in the latest version 2.11 (`https://jmeter.apache.org/`), where (a) may be refactored into (d) which is syntactically similar to (c) but does not actually exist in the program.

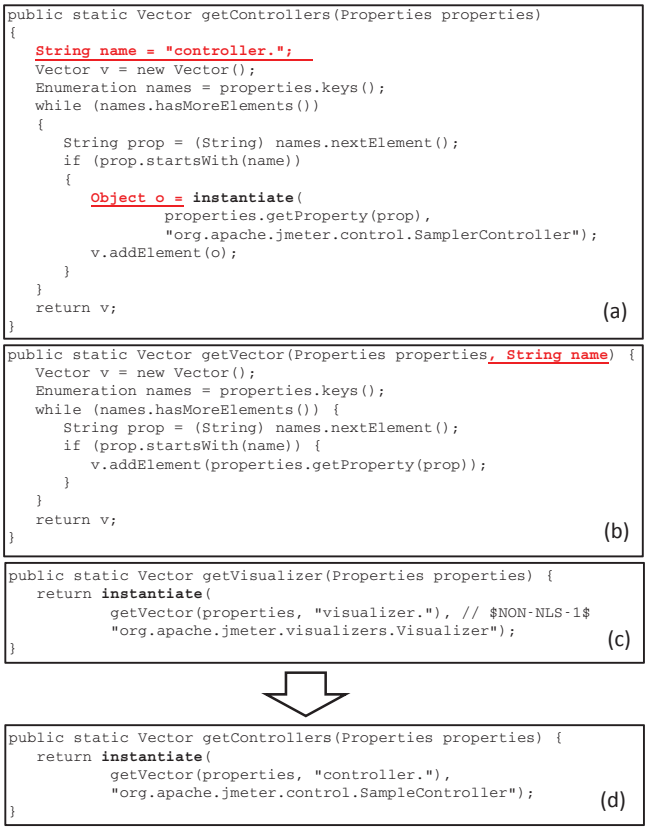

Figure 1: Sample refactoring detected from JMeter.

similar to (a), except for differences in variable declarations (the underlined red parts) in (a). Again, usual code clone techniques [2, 11, 22, 24, 26, 41, 48] may detect (a) and (c^I) as clones only if relaxed similarity conditions are used to tolerate the differences, but these techniques cannot flexibly specify which differences to tolerate when computing similarity, and using a relaxed similarity to force the detection of (a) and (c^I) may produce other false clones that differ in other program elements than the variable declarations.[2] This indicates that similarity-based clone detection is not sufficient for accurate refactoring detection; a desired refactoring detection tool should possess knowledge about various kinds of refactoring operations and work flexibly with specific program elements.

Last but not least, in the context of discovering refactoring opportunities in large code bases, we may need to compare many code fragments against each other, and there is the added challenge to locate suitable refactoring candidates from multitude of code very efficiently.

In this paper, we present a new vector-based approach for scalable detection of both refactoring opportunities and historical refactorings. We first construct characteristic vectors that can be used to encode syntactic features of code, and use such vectors to encode *inlined* code so that the effect of method extraction and inlining, which are commonly performed by various refactoring operations, can be captured as well. Then, we present a novel approach via vector *abstraction* and *concretization* that can manipulate vectors flexibly based on code changes induced by known refactorings. By using vectors and vector operations as the representation of code and code changes, our approach reduces the problem of detecting refactoring to the problem of finding similar vectors satisfying certain refactoring conditions. Since vector-based operations can be performed in almost linear time with respect to the number of vectors and the dimension of each vector, it becomes the key to the scalability of our approach.

For each code fragment identified as a refactoring candidate, our approach also reports the likely refactoring operation applicable to c by reporting a set of sample code fragments that may have been refactored via the same kind of refactoring operations. Such sample refactored code may help users understand better how to refactor c. For the example shown in Figure 1, our approach identifies (a) as a refactoring opportunity, and reports (b) and (c) together as a sample. Then, a user could proceed to refactor (a) in a way similar to (b) and (c), and transform (a) into (d).

We have implemented our approach for Java (for both source code and bytecode) and enabled vector abstraction and concretization operations for 21 common types of refactoring operations. Our prototype is scalable and precise: In a large code base comprising of 200 bundle projects in the Eclipse ecosystem (e.g., Eclipse JDT, Eclipse PDE, Apache Commons, Hamcrest, ObjectWeb ASM, etc.) containing 4.5 million lines of code, the tool reported in total more than 32K instances of 17 types of refactoring opportunities, taking 25 minutes on average for each type; In a smaller code base containing 16 versions of three Java programs (JMeter, Ant, and XML-Security), our tool reported 191 historical refactorings across various versions and more than 2.8K instances of refactoring opportunities. With validation by four graduate students, we find that the detected refactorings are of high precisions, about 92% for historical refactorings and about 87% for refactoring opportunities.

Our main contributions in this paper are as follows:

- We design a systematic way to encode syntactic features of code and code changes needed for various types of refactoring operations as abstraction and concretization operations of characteristic vectors;
- Our encoding of code and code changes does not need to differentiate code within the same version or from different versions of a program, so that we can detect both refactoring opportunities and historical refactorings;
- Our vector-based encoding and similarity queries are efficient and enable scalable detection of refactorings;
- We have evaluated our approach on large code bases with scalable and precise detection results.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 presents our detection approach. Section 4 presents specific vector abstraction and concretization operations used in our approach. Section 5 presents the results of our empirical evaluation and discusses threats to validity. Section 6 concludes with future work.

2. RELATED WORK

This paper is related to many studies in refactoring detection, and software maintenance and evolution in general. The discussion here is by no means complete.

There are many introductions and surveys for software refactoring and related tools [9,21,30,35,38,39]. Some surveys and tools investigate the relations between refactoring and code clones [8,19,54,58], and clone detection has been touted as an important way to detect refactoring opportunities (Ex-

[2]In fact, removing the underlined variable declarations in (a) and replacing the uses of the variables with their right-side expressions is another kind of refactorings, called "inline temp."

tract or Pull-Up Method in particular). Different from clone detection techniques, especially the ones based on vector representations for code [11, 22], our approach extends vector operations with abstraction and concretization, and utilizes various characteristics specific to refactoring operations.

Many studies aiming for automatic detection of refactorings (besides the studies on clone detection) rely on the changes recorded in version control systems. Their focus is to reconstruct historical refactoring operations that have happened. Demeyer et al. [4] define heuristic metrics to search for refactoring between successive versions. Hayashi et al. [18] model the refactoring detection as a graph search representing structural differences between two versions of a program. Weißgerber and Diehl [60] define signatures based on code clone detection results to look for refactoring. Prete and Kim et al. [27, 46] use template logic queries to represent refactoring operations and a logic programming engine to search for refactoring happened between versions of a program; their tool RefFinder can detect 63 kinds of refactorings in Fowler's catalog [9]. Taneja and Dig et al. [5, 55] present tools (`RefactoringCrawler` and `RefacLib`) to detect refactorings between different versions of libraries. Soetens et al. [52] detect refactoring operations as actual changes are happening in an integrated development environment, and thus achieve higher accuracy than previous work. Origin analysis has also been used to detect refactoring [15] by capturing certain kinds of cross-function changes and how call relations change between two versions of a program. Different from these techniques, our detection technique is vector-based and is not limited for changes between versions; it can search whole code bases and detect refactoring opportunities within the same version of a code base as well.

There exist studies that detect refactoring opportunities; they are related to many studies on detecting "bad smells" in code [9, 31, 37]. Tourwe and Mens [56] use logic programming to encode several types of refactoring operations and detect possible refactoring opportunities. Meng et al. [33] create context-aware edit scripts from two or more examples and use the scripts to identify edit locations and transform the code. This approach can be applied to find refactoring opportunities and fully automated, while we have yet to automate some steps in our approach. However, the edit-scripts are so far limited within a single method, as their experience shows that combining inter-procedural analysis and the expressiveness of general-purpose edits is a very hard problem. They cannot detect changes that require moving code from one method to another or coordinating changes to multiple methods in the way our approach does. Some other studies detect refactoring opportunities by searching for user-built code that may be replaced with certain library calls [53] or upgraded with calls to different versions of library APIs [42]. Those techniques mostly rely on graph-based matching and substitution, while we rely on compact and efficient vector-based representations.

Cider [51] is another work that can detect refactoring without code change histories. Their algorithm relies on graph matching and requires initial seeds that are similar code fragments at first, and is limited within individual methods too. Our technique does not need seeds and relies on vector matching, making it more scalable to large code bases where code divergences across functions occur more often. Hui et al. [32] identify particular kinds of generalization refactoring opportunities. Our vector-based approach detects different types of refactorings and can complement those studies.

Many other studies on refactoring focus on the specification and implementation of refactoring operations. A classical work by Opdyke [44], describes a set of refactoring operations for C++ in terms of the preconditions needed to preserve behaviour. Griswold specifies refactoring from the perspective of their effects on program dependence graphs [16]. Lämmel [29] and Garrido [12] use rewriting rules to represent refactoring. Recent studies also aim to allow programmers to script their own refactoring operations. To this end, Verbaere et al. [59] propose a domain specific language *JunGL* for expressing dataflow properties on a graph representation of the program. Schäfer et al. [50] improve on this and provide high-level specifications for many refactoring operations implemented in Eclipse. Ge and Murphy-Hill [13] can automatically validate a manually performed refactoring. Our work complements those studies in that it searches for new refactoring opportunities. As future work, we plan to investigate the development of a language for specifying vector abstraction and concretization that would allow us to more comprehensively and precisely specify the intended refactorings, in addition to learning from examples.

Many of the above mentioned studies can also automatically perform identified refactoring. Modern development environments, such as Eclipse and NetBeans, have automated refactoring capabilities. Concurrencer [6] can identify and convert sequential code that may be benefited from the `java.util.concurrent` supports. LambdaFicator [10], automatically refactors certain anonymous inner Java classes and `for` loops to lambda expressions and functional operations available in Java 8. Our tool currently focuses on detection only. As our tool reports refactorings together with possible refactoring results, it can also be improved to perform identified refactorings automatically.

3. METHODOLOGY

We explain the main steps of our approach along with Figure 2. Given a source code base, we construct its syntax trees (STs), and call graphs (CGs). The STs are used in a way similar to previous studies [11, 22] to generate characteristic vectors for code fragments from the code base. When the code is compilable, we also generate the bytecode (for Java) or binary code, and construct characteristic vectors for the bytecode or binary code as well [49]. Using bytecode or binary code has the benefit that many code differences only applicable to high-level languages (e.g., different syntaxes for writing `for` loops) are unified or eliminated, which can potentially help to detect more refactorings [49]. We also simulate the effect of method linining by manipulating the STs based on call relations and get inlined code, and generate vectors for code fragments in the inlined code as well. Our tailored vector generation is described in Section 3.1.

After vectors are generated, they are *abstracted* to eliminate or unify code characteristics related to a particular type of refactoring γ. The particular code characteristics are semi-automatically extracted from known sample code refactored by γ (see Sections 3.2 and 4).

Then, hash-based search (simple hash and locality-sensitive hashing (LSH, [14])) is used to query for similar *abstracted* vectors efficiently so that we can identify candidates for refactoring (see Section 3.3). Not all candidates can be true refactorings. We then apply vector *concretization* to check whether the characteristics in the *concrete* vectors indeed match the code characteristics of a particular type of refac-

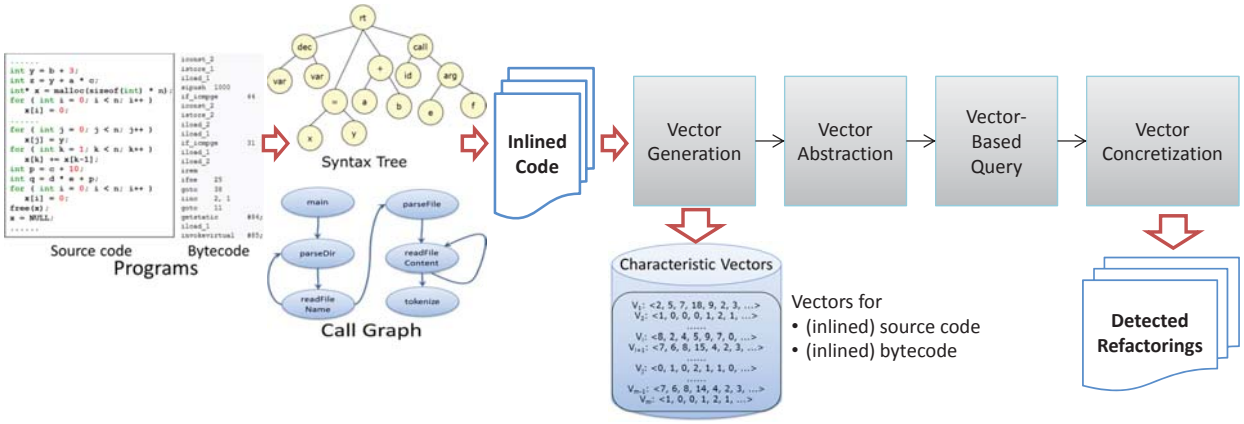

Figure 2: Overview of our approach.

toring (see Section 3.4). We can afford to do more detailed checks during concretization since the number of candidates is much smaller than the original code sizes. Finally, the code fragments corresponding to the candidates that are likely to be true refactorings are reported to users.

3.1 Vector Generation

In this work we use characteristic vectors for the purpose of refactoring detection. We define vectors as follows.[3]

Definition 3.1 (Characteristic Vector). *Given a sequence of K unique features denoted by $[f_1, \ldots, f_K]$, a characteristic vector v for a code fragment c is an array $[n_1, \ldots, n_K]$ of size K such that $n_1, \ldots, n_K \geq 0$ and for each i, n_i is* the number of occurrences *of the feature f_i in c.*

An entry in a vector v can be referred to by either an index i or a feature f_i, denoted by $v[i]$ or $v[f_i]$ respectively. In principle, the features can be anything in the code of interest to an application. For example, they can be different types of nodes in the syntax tree of c to represent the *syntactic* characteristics of the code [22], or be certain parts of the syntax tree that match slices of the program dependence graph of the code [11]. Following the previous work, we use the types of the nodes in syntax trees as features for this paper. Note that node types for source code and bytecode can be different and thus corresponding vectors can be different.

Since the vectors are generated according to the number of occurrences of program elements in code, they themselves do not capture various specific information about each element (e.g., the specific name of an identifier, the specific value of a constant, etc.) or relational information between elements (e.g., the containing class of a method or a field, the parent class of a child class, a statement appearing before another statement, etc.). Nevertheless, such vectors have been shown to be effective for code clone detection [11, 22].

3.1.1 For Original not-yet-Inlined Code

Given a code fragment c from a code base, we can identify the nodes of the syntax tree that match the location of c and then count the number of occurrences of different node types. For example, sample heavily simplified vectors for the code fragments Figure 1(a), (b), and (c) are shown in Table 1; the table headers indicate the sample features used for the vectors; rows 1–3 are the vectors for each of the three methods. Separated from the usual method invocations ("mth invoc."), "API invoc." refers to invocations of methods not defined in the subject program; "new invoc." refers to invocations of constructors (e.g., `new Vector()`). The actual number of features in the vectors depend on the number of different types of nodes in (either abstract or concrete) syntax trees for a programming language. Our prototype implementation relies on abstract syntax trees generated by Eclipse JDT for Java, which has more than 80 node types.

Table 1: Sample partial vectors for code in Figure 1

row ID	Code	Features									
		simple name	string literal	var. decl. stmt.	cast	if	return	while	mth invoc.	new invoc.	API invoc.
1	getControllers	29	2	5	1	1	1	1	1	1	6
2	getVector	23	0	3	1	1	1	1	0	1	6
3	getVisualizer	3	2	0	0	0	1	0	2	0	0
4	getVisualizer (inline getVector)	26-1	2	3	1	1	2-1	1	2-1	1	6

These vectors only capture characteristics of the code inside the same function: if a method is invoked in a code fragment, the vector for the code fragment does not capture any characteristic of the code inside the invoked method, except the method invocation expression and actual parameters. Thus, in this paper we call these vectors *base-level* characteristic vectors, or simply *base* vectors. Although refactoring can happen on arbitrary pieces of code, our implementation only generates base vectors for method bodies, as the bodies likely contain refactorings in smaller scopes too and we do not aim to detect *minimal* code fragments that may be refactored. This differs from the vector generation strategy for clone detection [11, 22], and helps to reduce our search space.

3.1.2 For Inlined Code

Refactoring may involve different ways of extracting or inlining methods. To encode various possible changes induced by method inlining or extraction, we also consider different ways to inline methods for a given code fragment c. In general, if c invokes n methods, there could be up to 2^n ways to inline the n methods in combination. To reduce the search space in this paper, we inline methods invoked in c in mainly three different modes: inlining all methods invoked in c all at once, inlining all calls to each distinct method separately, or inlining nothing. We do not inline constructor and API invocations. In this way, the number of inlined versions C^I of c may equal to two plus the number of distinct methods defined in the program and called in c. An inlined version c^I for c can be the same as c when the mode of "inlining nothing" is applied or when no method is called in c.

[3] We use the term "characteristic vector" and "vector" interchangeably.

We simulate the effect of method inlining by summing up the vectors for the caller and the callee and manipulating the features in the vectors that are related to method declarations and invocations, i.e., the features for the invocations, `returns`, and formal and actual parameter substitutions. Specifically for the features shown in Table 1, we reduce the occurrence counters for "mth invoc." and "simple name" each by one for each method called ("simple name" is a child node of "mth invoc.", representing the method name in the syntax trees generated by Eclipse JDT for Java), and remove all counts for `return`s from the callee. We assume each actual argument is only evaluated once and the corresponding formal parameter somehow automatically receives its value, and thus the vector manipulations do not need to consider the effect of parameter substitution. For example, when we inline `getVector` into `getVisualizer`, the vector for `getVisualizer` is changed as the row 4 in Table 1. The red parts of the row indicate the manipulations applied to the sum of rows 2 and 3 to simulate the inlining. Such simulated method inlining via vector manipulation has been shown in our previous work to be effective for detecting method extraction and inlining [36]. This paper employs the same idea of vector inlining, but extends it to define vector abstraction and concretization for detecting more types of refactorings.

We note that the manipulation of vectors to simulate method inlining may be language-specific; it depends on the structure of syntax trees as well; it can be different for source and bytecode too. However, the idea of encoding method inlining as vector operations can be generally applicable to different programming languages.

In the following discussion, we use the following terms and notations: given a code fragment c, we call it *base code*, and its vector is called *base vector* and denoted as v_c. The set of all possible inlined versions of c is denoted as C^I, while an instance in the set is denoted as c^I. The vector for the *inlined code* c^I is called *inlined vector* and denoted as v_c^I.

3.2 Vector Abstraction

Our objective here is to encode code changes induced by a kind of refactoring operation in the form of vectors as precisely as possible, and abstract away (or eliminate) the changes from the vectors representing code, while maintaining essential code features, so that the *abstracted* vector representations for the code before and after it is refactored can be the same. Then, the problem of searching for refactorings can be reduced to the problem of finding code with the same abstracted vector representation.

Each refactoring type has a different abstraction since they often induce different code changes. We use $\mathcal{A}$ to denote the abstraction operation for a refactoring type γ. $\mathcal{A}(v)$ means to apply the abstraction onto a vector v, and ${}^{\gamma}\psi$ denotes the resulting abstracted vector. The "Query" portion on the left in Figure 3 illustrates the conceptual relations among base code, inlined code, and various kinds of vectors with respect to a refactoring γ: A piece of base code used as a query q can have more than one inlined code q^I; its base vector v_q can become an abstracted base vector ${}^{\gamma}\psi_q$; and its inlined vector v_q^I can become an abstracted inlined vector ${}^{\gamma}\psi_q^I$. It is possible that ${}^{\gamma}\psi_q^I$ may be the same as v_q^I and/or v_q.

In this paper, we use a semi-automated mechanism to extract differences from sample code refactored by a type of refactoring γ and define the abstraction for γ systematically based on the differences. We introduce our definitions:

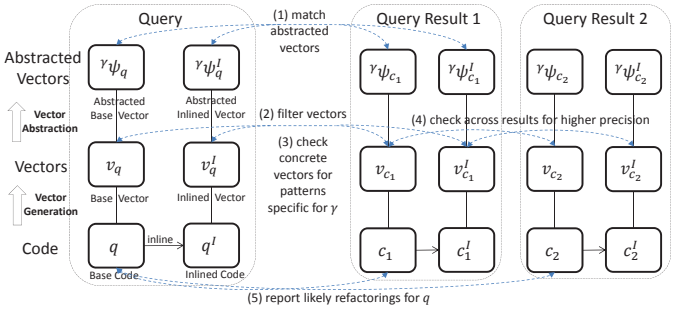

Figure 3: Given a piece of code q, search for code similar to q if q may be refactored via a refactoring operation γ.

Definition 3.2 (Vector Substitution). *Given a vector v and a set of mappings from features to counts ($\mathcal{F} = \{f_i \mapsto n_i\}$), the* vector substitution *is denoted by $v(\mathcal{F})$; it generates a new vector v', such that :*

$$\forall i \in 1..K,\ v'[i] = \begin{cases} n_i & \text{if } \{f_i \mapsto n_i\} \in \mathcal{F} \\ v[i] & \text{otherwise} \end{cases}$$

Both n_i and $\mathcal{F}[i]$ denote the mapping result for a feature f_j.

Definition 3.3 (Vector Difference). *Given two vectors v_1 and v_2, the* vector difference *operation δ for v_1 and v_2 is defined as $\delta(v_1, v_2) = (v^\delta, m, \mathcal{D})$ where*

1. *v^δ is a vector called* assimilation vector *between v_1 and v_2: $\forall i \in 1..K,\ v^\delta[i] = min(v_1[i], v_2[i])$;*
2. *$0 \le m \le K$;*
3. *$\mathcal{D}$ is a feature mapping set of size m: $\forall i \in 1..K,\ (f_i \mapsto (v_2[i] - v_1[i])) \in \mathcal{D}$ iff $v_1[i] \neq v_2[i]$.*

Such vector difference operations $(v^\delta, m, \mathcal{D})$ encode both "common" parts (in v^δ) and differences (in $\mathcal{D}$) between two vectors. When v_1 and v_2 correspond to two sample pieces of code c_1 and c_2, and c_2 is the result of applying a certain refactoring operation γ onto c_1, the feature mapping set $\mathcal{D}$ indicates the features that may be changed by γ, and can help us define the abstraction operation $\mathcal{A}$ for γ that can abstract away the changes that may be induced by γ into an arbitrary vector v. The abstracted vector for v is denoted by either $\mathcal{A}(v)$ or ${}^{\gamma}\psi$. The rules below describe how ${}^{\gamma}\psi$ is generated for an arbitrary v, based on a given $\delta(v_1, v_2)$. The rules are conceptually the same for source code and bytecode.

(I) ${}^{\gamma}\psi[i] = v[i]$, if $\mathcal{D}$ does not contain a mapping for f_i.

(II) if there is a subset of $\mathcal{D}$, denoted as $\mathcal{D}_f = \{f_{d_1} \mapsto n_{d_1}, f_{d_2} \mapsto n_{d_2}, \ldots, f_{d_s} \mapsto n_{d_s}\}$ where $2 \le s \le K$ and $1 \le d_1 \le d_2 \ldots \le d_{s-1} \le d_s \le K$, such that $\Sigma_{i=1}^{s} n_{d_i} = 0$,[4] then we consider the features in $\mathcal{D}_f$ as inter-exchangeable and we merge their counts in v all into a unique conceptual feature as follows:
- ${}^{\gamma}\psi[d_1] = \Sigma_{i=1}^{s} v[d_i]$;
- $\forall i \in 2..s, {}^{\gamma}\psi[d_i] = 0$.

For example, various relational operators (`<`, `>`, `<=`, and `>=`) in code are in fact inter-exchangeable, since a refactoring operation can reverse the condition in an `if` statement and swap the branches of `if`. Such a refactoring would induce changes in the counts for the individual operators, but the total sum of the counts for these inter-exchangeable code features should remain the same.

(III) if there is a subset of $\mathcal{D}$, denoted as $\mathcal{D}_f = \{f_{d_1} \mapsto n_{d_1}, f_{d_2} \mapsto n_{d_2}, \ldots, f_{d_s} \mapsto n_{d_s}\}$ where $2 \le s \le K$ and $1 \le d_1 \le d_2 \ldots \le d_{s-1} \le d_s \le K$, such that $n_{d_1} = n_{d_2} =$

[4] $\forall (f_{d_i} \mapsto n_{d_i}) \in \mathcal{D}, n_{d_i} \neq 0$ because of the rule 3 in Def. 3.3.

$\ldots = n_{d_s}$, then we consider the features in $\mathcal{D}_f$ should be changed together in the same way by γ and we set all their counts to 1 as follows:

- $\forall i \in 1..s, {}^{\gamma}\psi[d_i] = 1$.

This condition helps the refactoring cases when it is not important to count the actual number of occurrences of a code feature as long as the feature exists in the code. For example, the refactoring type "Consolidate Duplicate Conditional Expression" consolidates more than one conditional expression into one, so the features essential to the conditional expressions were all set to 1.

(IV) there may be multiple subsets of $\mathcal{D}$ satisfying the above conditions; if the subsets are disjoint, we perform the abstraction for each subset separately; if the subsets overlap, we manually identify a subset to abstract. In our experiments, vector differences $\mathcal{D}$ are generated from sample refactored code in classical collections (see Section 4) and of small sizes, and it was easy to find suitable subsets as above efficiently.

(V) ${}^{\gamma}\psi[i] = 0$, otherwise. The intuition for this rule is that if a feature f_i can be changed by the refactoring γ but not in "conjunction" with other features, f_i may in fact be changed *arbitrarily* by γ, and it is non-essential for γ, and thus we abstract it away. As one can see, we define "conjunction" as the subsets satisfying the rule (II) or (III), and we found the simple rules are sufficient for the refactoring types detected in our experiments. Also, since the rules do not yet consider the semantics of the features, the mapping $\mathcal{D}$ generated from sample refactored code may contain more features than what are really needed for γ. So, we also manually verify each inferred abstraction in our experiments.

For example, we can define the vector abstraction for the kind of refactoring operation in Figure 1. Even though those code snippets are detected by our tool, here we use them as sample refactored code to illustrate how we define the abstraction for a refactoring operation based on sample refactored code. For this case, the vector in row 1 in Table 1 is v_1 and the other in row 4 is v_2; the vector difference $\mathcal{D}$ is {"simple name"$\mapsto -4$, "var. decl. stmt."$\mapsto -2$} which indicates the removal of two variable declaration statements containing four simple names (two are for the variable names; the other two are for the variable types). The above abstraction rule (V) applies, so the abstraction $\mathcal{A}$ would set the counts for both "simple name" and "var. decl. stmt." to zero. Table 2 shows the abstracted vectors if the abstraction is applied to the concrete vectors in Table 1.

Table 2: Sample abstracted vectors for vectors in Table 1

row ID	Code	Features									
		simple name	string literal	var. decl. stmt.	cast	if	return	while	mth invoc.	new invoc.	API invoc.
1	getControllers	0	2	0	1	1	1	1	1	1	6
2	getVector	0	0	0	1	1	1	1	0	1	6
3	getVisualizer	0	2	0	0	0	1	0	2	0	0
4	getVisualizer (inline getVector)	0	2	0	1	1	1	1	1	1	6

When more than one pair of sample code is provided for a refactoring operation γ, we can refine the extracted abstraction for γ to represent the most general code changes induced by γ. To achieve this, we can calculate the vector difference $(v^{\delta}, m, \mathcal{D})$ for every pair, and look for the "maximum common difference" among all those $(v^{\delta}, m, \mathcal{D})$. In this paper, we still employ manual efforts to use appropriate thresholds and refine the extracted abstraction if necessary. As interesting future work, we plan to automate the extraction of abstraction from given sample code based on vector arithmetics. Such automation may be in spirit similar to studies on specification mining [43, 61–63] and programming by examples [17,33,34,40], but it will use a significantly different technique based on vector representation and arithmetic of the characteristics of code and code changes.

3.3 Vector-Based Query

When we want to find instances of a type of refactoring operations γ in a large code base, we apply the abstraction for γ to all vectors generated from the code base. The code difference induced by γ should thus be eliminated, and the abstracted vectors of either refactored or non-refactored code should appear the same. Then, we can use hash-based matching techniques [3] to find vectors that are either matching exactly or very similar to each other [11, 22].

We tailor the queries in our approach to answer the question: *can a piece of code q be refactored via a refactoring operation γ so that it becomes similar to some other code?* As illustrated in the step (1) in Figure 3, we perform queries on abstracted vectors for either base or inlined code, or both: abstracted vectors (either ψ_q or ψ_q^I or both) are used, each as a query against all other available abstracted vectors to identify the ones matching the query. Depending on the vectors used, we can identify candidates for both historical refactorings and refactoring opportunities:

(1) When we use every abstracted vector generated for one version of code as a query to search for matching ones in the set of abstracted vectors for another version of the code, we can detect historical refactorings happened between two versions of the code.

(2) When we use every abstracted vector generated for a code base as a query to search for matching ones in all abstracted vectors for the same code base, we can detect refactoring opportunities that may be similar to some refactorings that have happened in the code base.

The types of refactorings under investigation would affect whether the corresponding queries and matching outcomes are drawn from either base vectors or inlined vectors or both. For example, for the detection of the opportunity of method inlining shown in Figure 1(a), we used the abstracted vector for the *base* vector of (a) as a query and search for its matches among the abstracted vectors for all other *inlined* code.

Not all matched vectors can be refactorings; we apply heuristic filters (Figure 3, step (2)) to reduce unlikely ones:

FilterSmall: When a piece of code is too small (e.g., smaller than the number of elements involved in the abstraction for γ or the sizes of the sample code used to define the abstraction), it may not be useful to refactor it. We can use a threshold (e.g., 50% of the sizes of the sample code or 10 program elements or 1 functioning statement) to remove code that is too small.

FilterClones: When comparing the concrete vectors for both the query code and the result code, if both $v_q = v_{c_1}$ and $v_q^I = v_{c_1}^I$, q and c_1 are very likely the same syntactically, and their inlined versions are the same as well. c_1 is simply a clone of q and may not indicate how to refactor q, and thus can be removed.

FilterNames: Many refactoring operations would maintain various names (e.g., some variable names in the code and the name of the method/class/file containing the

code) the same before and after the refactoring. We can remove a query result if its fully qualified method name does not match that of the method containing the query code. This can be useful for detecting and reconstructing historical refactorings happened between versions, where the query code and the result code are in different versions of a program and often share same name. We only turn on this filter for across-version refactoring detection.

For the code fragments in Figure 1, (a) and (c) inlined with (b) can be detected as likely refactorings since their abstracted vectors (rows 1 and 4 in Table 2) are the same.

3.4 Vector Concretization

After above steps, we have a set of filtered query results for each piece of code used as a query. The following concretization phase performs several kinds of checks on the concrete vectors corresponding to the query and the query results to improve the precision of refactoring detection. This phase corresponds to steps (3) and (4) in Figure 3.

The first kind of checks is to make sure the *differences* among the concrete vectors indeed subsume the differences ($\mathcal{D}$, see Section 3.2) that may be induced by a kind of refactoring operation γ; i.e., γ may really be applied to the corresponding code. This is useful for reducing false positives since different refactoring operations may in fact change same features in code and having similar abstracted vectors may not mean the corresponding concrete vectors satisfy requirements needed by γ (see the concretization rule (I) below).

The second kind of checks is to make sure the reported query results indeed have the contexts in which the refactoring operation γ can be carried out. For example, the refactoring "Reverse Conditional" reverses the relational operator in an `if` statement and swaps the branches of the `if`, and thus the refactoring can only happen when the code contains at least one `if`, even though the feature representing `if` itself is not changed by the refactoring. So we perform checks that the *common parts* among the concrete vectors indeed subsume the common parts (v^δ, see Section 3.2) that represent the contexts needed for γ (see the rule (II) below).

For certain types of refactorings, we manually add special checks for them (see Section 4), based on our understanding of the code changes involved in the refactorings, to help reduce false positives. For example, a refactoring operation may simply replace the *whole* body of a method with a call to a newly extracted method containing the replaced body. Although such a refactoring may be classified as "Extract Method", it may be too simple to be useful. Thus, we filter such cases during concretization (the rule (III)).

We also cross-check query results to improve their credibility. Intuitively, the query results should be syntactically different from the query; otherwise, they are likely clones only, not refactorings. Also, when there are more than one query result that are syntactically different from each other, they may indicate more than one way to refactor the query code, which may appear confusing for users. We thus choose to remove such cases so that users can have higher confidence that the refactoring operation indicated by the final query results can be applied to the query code (the rule (IV)).

The rules below describe the above checks more rigorously:

(I) Calculate the vector difference between v_q^I and each $v_{c_i}^I$: $\delta(v_q^I, v_{c_i}^I) = (v_q^{I\delta}, m_q^I, \mathcal{D}_q^I)$. Check them against the vector difference $(v^\delta, m, \mathcal{D})$ for γ, and remove the query result c_i if one of the following conditions is true:
  (1) if $\exists(f_i \mapsto n_i) \in \mathcal{D}, s.t.\ (n_i < 0) \wedge (v_q^I[i] < |n_i|)$, it means γ would need to remove $|n_i|$ instances of the code feature f_i but q^I does not contain enough;
  (2) if $\exists(f_i \mapsto n_i) \in \mathcal{D}, s.t.$ either $\mathcal{D}_q^I$ does not contain f_i or $|\mathcal{D}_q^I[f_i]| < |n_i|$. This indicates that the changes between v_q^I and $v_{c_i}^I$ are too few in comparison with the changes induced by γ to be a real case of γ;
  (3) if $\exists i \in 1..K, s.t.\ (v_q^I[i] < v^\delta[i]) \vee (v_{c_i}^I[i] < v^\delta[i])$, it means γ would need to be carried out in a context containing $v^\delta[i]$ instances of the code feature f_i but q^I or c_i^I does not contain enough;

(II) Check v_q and v_{c_i} against $(v^\delta, m, \mathcal{D})$, and remove the query result c_i if the following condition is true:
  (1) if $\exists i \in 1..K, s.t.\ (v_q[i] < v^\delta[i]) \vee (v_{c_i}[i] < v^\delta[i])$, it means γ would need to be carried out in a context containing $v^\delta[i]$ instances of the code feature f_i but q or c_i does not contain enough;

(III) Check all base and inlined vectors against code change rules specific for γ to remove possibly more query results;

(IV) We finally check the query results against each other if there are still more than one result at this step. We remove all of the results if the following condition is true:
  (1) $\exists i, j, s.t.\ i \neq j \wedge v_{c_i} \neq v_{c_j}$;

Finally, the code corresponding to the query and checked query results are reported as refactorings. For the code fragments in Figure 1, one of the differences among their concrete vectors (Table 1, rows 4 and 1) indeed match the vector difference operation ({"simple name"$\mapsto -4$, "var. decl. stmt."$\mapsto -2$}). Their contexts are also matched. Thus, (a) and (c) inlined with (b) are reported as refactorings.

4. REFACTORING AS VECTOR ABSTRACTION & CONCRETIZATION

Our approach is based on abstraction and concretization of characteristic vectors that capture various code features before and after certain refactorings. The effectiveness of our approach is dependent on how well the vectors can represent code features. As mentioned in Section 3.1.1, the vectors used in this paper only capture code features related to the number of occurrences of program elements in code. Thus, our approach is tailored to detect refactoring operations that would change the number of occurrences of various program elements in code. Some refactoring operations can induce code changes that are not represented by the vectors. For example, "Pull Up Method" moves a method from a child class to its parent class. The moved method itself is the same before and after the refactoring, but its containing class is changed. "Rename Method" changes the name of a method. The characteristics of such changes are not captured in the vectors, and thus are not yet detectable by our approach. It will be our future work to extend the capabilities of vectors to encode and index programs more comprehensively.

In this paper, the sample refactored code used for constructing abstraction and concretization rules is all from classical collections [9, 21]. We have rules for 21 types of refactorings, although not every type has detection results in our experiments. Due to our page limit, Table 3 only lists 11 types, and uses simplified notations and descriptions, instead of rigorous vector-based operations. Abstraction rules that do not change the values for a feature are not shown. Some concretization rules are the same for all types of refactorings (as described in Section 3.4); they are also omitted.

Table 3: Sample abstraction and concretization operations. In addition to the notations used in Section 3, vectors superscripted with "S" are generated from source code, while others are generated from bytecode. Many features used in the operations characterize bytecode instructions in our implementation, but we use more high-level names for the features here for illustration purposes. Due to space limitation, we rely on the feature names to convey their meaning.

#	Refactoring	Abstractions (${}^{Y}A(v)$ or simply ψ)	Concretization Checks	Descriptions
1.	**Extract Method**	ψ[load] = ψ[store] = 0 ψ[constant] = 0	$\nexists$ v_i inlined into v_c s.t. v_q== v_i (remove simple extraction methods)	
2.	**Inline Method**	ψ[load] = ψ[store] = 0 ψ[constant] = 0	$\nexists$ v_i inlined into v_q s.t. v_q == v_i (remove simple extraction methods)	
3.	**Inline Temp**	ψ[load] = ψ[store] = 0	v_q [load] - v_c [load] > 0 && v_q [load] - v_c [load] == v_q[store] - v_c [store] == v_q^S[var_declaration] - v_c^S[var_declaration]	Remove the declaration of a temporary variable, and replace the use of the variable with the value of the variable.
4.	**Introduce Explaining Variable**	ψ[load] = ψ[store] = 0	v_q [load] - v_c [load] < 0 && v_q [load] - v_c [load] == v_q[store] - v_c [store] == v_q^S[var_declaration] - v_c^S[var_declaration]	Extract a complicated expression into a temporary variable.
5.	**Split Temporary Variable**	ψ[load] = v[load]+$\sum$v[load_i] ψ[store] = v[store]+$\sum$v[store_i] ψ[load_i] = ψ[store_i] = 0 i $\in$ { 0, 1, 2, 3 }	v_c^S[variable_declaration_statement] >=2 && (v_q^S[assignment] - v_c^S[assignment]) == - (v_q^S[var_declaration] - v_c^S[var_declaration])	Transform multiple assignments to a temporary variable into separate variable declarations for each assignment.
6.	**Replace Method With Method Object**	ψ[load] = ψ[store] = 0 ψ[getfield] = ψ[putfield] = 0 ψ[new] = ψ[invoke_init] = 0	$\exists$ v_i inlined into v_c s.t. ($\sum$ v_i[f] - $\sum$ v_q[f]) == (v_i[getfield] + v_i[putfield]), f $\in$ {0,.. v_q.length} (v_i[getfield] - v_q[getfield] > 0) (v_c^I[new] - v_q[new] > 0)	Transform a method into its own object so that all the local variables become fields. Abstraction involved ignoring "new" operators and encapsulation.
7.	**Self Encapsulate Field**	ψ[aload_0] = 0 (aload_0 is used for loading "this" on the stack)	v_q [getfield] - v_c^I[getfield] > 0 // $\exists$ an extra field to encapsulate $\exists$ v_i inlined into v_c s.t. v_i [getfield] == 1 && v_i [return] == 1 && v_i [aload_0] == 1 v_i [op] == 0 where op!= getfield && op!= return && op!= aload_0	Replace direct accesses to a field with a getter method
8.	**Replace Magic Number with Symbolic Constant**	No abstraction needed as both are represented by the same bytecode.	$\sum$ v_q^S[literal] - $\sum$ v_c^S[literal] > 0 // query has more magic numbers literal $\in$ { string_literal, boolean_literal, num_literal } v_q^S[simple_name] - v_c^S[simple_name] < 0	Replace constants used in code with symbolic names for easier maintenance
9.	**Replace Magic Number with Query Method**	No abstraction needed as both are represented by the same bytecode.	$\exists$ v_i inlined into v_c s.t. $\sum$ v_i^S[op] == 1, op $\in$ { string_literal, boolean_literal, num_literal } v_i^S[return] == 1 v_i^S[totalCount] == 2	Replace constants used in code with a getter method that returns the constants.
10.	**Reverse Conditional**	ψ[eq] = v[eq] + v[neq] ψ[lt] = v[lt] + $\sum$ v[opp] ψ[neq] = ψ[opp] = 0 opp $\in$ { gt, ge, le }	$\exists$ cond $\in$ { eq, neq} or opp $\in$ { lt, gt, ge, le } s.t. v_q [cond] - v_c [cond] != 0 || v_q [opp] - v_c [opp] != 0	Treat "==" the same as "!=" Treat "<=" the same as ">", ">=", and "<"
11.	**Encapsulate Downcast**	ψ[checkcast] = 0 (ignore type casts)	v_q [methodinvoke_checkcast] - v_c[methodinvoke_checkcast] > 0	Encapsulate type cast operations into a separate method returning the casted type

5. EMPIRICAL EVALUATION

This section presents our evaluation on four aspects: how many historical refactorings are detected, how many refactoring opportunities are detected, how accurate are the identified refactorings and how scalable is our approach.

5.1 Experimental Setting

In order to provide answers to the evaluation questions we have performed two case studies. All experiments related to these studies were performed on a PC running Ubuntu 10.04 with Intel Xeon at 2.67GHz and 24GB of RAM.

In the first case study we looked at three Java programs from the Software Infrastructure Repository (SIR): JMeter, XMLSecurity, and ANT. For JMeter, we performed experiments on 6 versions (0 to 5), for Ant on 6 versions (0 to 5), and for XMLSecurity on 4 versions. The size of these subject programs ranges from 17KLOC to 80KLOC. The projects were selected for comparing our prototype with the state-of-the-art in detecting historical refactorings—RefFinder [47], and for measuring the effectiveness of detecting refactoring opportunities by our prototype.

In the second case study we aimed to explore the scalability of our system. As such we have applied the prototype to a large code base containing 4.5 million lines of code and 200 bundle projects from the Eclipse ecosystem (e.g., Eclipse JDT, Eclipse PDE, Apache Commons, Hamcrest, etc.).

To evaluate the precision of the results detected by our approach, a group of four graduate students with good knowledge of Java and refactoring were invited to inspect the results independently. Due to the large number of results, we do not evaluate every one of them. For refactoring opportunities detected for the three subject programs, we chose to inspect the results for the first version of each program only. Each result inspector was required to verify that each of the detected refactorings is correctly classified. A result was counted as a false positive if any of the inspectors considered it as a false positive. For historical refactorings detected, we chose to inspect all of them due to a more manageable number.

It is not our focus to evaluate the recall of our approach due to lack of ground truths. However, we tested our approach on a set of examples taken from Fowlers catalog and found our approach can successfully detect all defined types of refactorings in the example set.

5.2 Detection Results

The results of the experiments performed in the first case study are shown in Table 4, 5, and 6. Each row in the tables shows the results obtained for one type of refactoring query; the types having no detection results are not shown in the tables. Each column having a single number (e.g. 1) as the header name shows the refactoring opportunities within a version of the project, while each column having a number

Table 4: Result summary for JMeter.

#	Program Versions	0	0-1	1	1-2	2	2-3	3	3-4	4	4-5	5
1.	Extract Method	11		7	1	6	15	6	1	7	1	7
2.	Inline Method	20		16		15		48		56		60
3.	Introduce Var	13		17		20		42		44		46
4.	Inline Temp	7		9		10		21		20	4	19
5.	Replace Assignment with Initialization	4		1		1	1	1		1		1
6.	Downcast Encapsulate	34		37		38	32	10		10		10
7.	Reverse Conditional							1		1		1
8.	Replace Magic Number with Symbolic Constant	1		5		5	2	6		6		6
9.	Self Encapsulate Field	17		17		18	3	17		18		18
10.	Replace Parameter with Method							10		10		10
	Subtotal	107	0	109	1	113	53	162	1	173	5	178

Table 5: Result summary for Ant.

#	Program Versions	0	0-1	1	1-2	2	2-3	3	3-4	4	4-5	5
1.	Extract Method	18	1	31		23		23	12	78	49	106
2.	Inline Method	18		31		29		29	2	53		80
3.	Introduce Var	1		21		17		17		52		53
4.	Inline Temp	2		6		9		9	1	14		21
5.	Replace Assignment with Initialization			3		3		3	1	4		4
6.	Downcast Encapsulate			10		4		4				
7.	Reverse Conditional	3		5		10		10		14		15
8.	Replace Magic Number with Symbolic Constant	1	3	11	3	17		16		179		195
9.	Self Encapsulate Field	28		47		55		55	6	137	50	128
10.	Introduce Parameter Object			1		1		1		7		8
11.	Split Temp								1	1		1
	Subtotal	71	4	166	3	168	0	167	23	539	99	611

range (e.g. 0-1) as the header shows the number of detected historical refactorings between two versions.

The number of historical refactorings we detected between versions ranges from 0 to 99 while the number of opportunities within each version ranges from 70 to 611. Out of total 2882 refactoring opportunities detected within all versions of the three subject programs,[5] the students inspected 276 and identified 35 false positives, giving a precision of *87% for our approach in detecting refactoring opportunities*.

We inspected all of the 191 historical refactorings detected. This validation was performed by the authors and the students, to verify that the classification reported by our approach for an actual code change between two versions is correct. We found 14 false positives, which resulted in a *92.6% precision for detecting historical opportunities*.

The tables show that the numbers of refactoring oportunities detected evolve from one version to another in a non-monotonous manner. An increase in the numbers of reported refactorings may imply that the size of the project increased due to code copy-paste operations or refactorings that have only been applied to parts of the project. A decrease may indicate that code was deleted, that previously similar code has diverged in shape or that the opportunities were applied. An example of the latter situation is exhibited by JMeter between the versions 2 and 3 for the refactoring Encapsulate Downcast (Row # 6 and Columns # 2, 2-3, and 3 in Table 4). In the versions 0, 1, and 2 of JMeter, a large number of methods invoked method `getProperty` from class `Task` and downcasted its result to obtain a string. Another category of methods invoked method `getPropertyAsString` from class `Task` which had the downcast pushed inside the method. The similarity between the methods that invoked `getProperty` and those that invoked `getPropertyAsString`

[5]A piece of code can be counted multiple times if it appears in multiple refactoring types in our results.

Table 6: Result summary for XML-Security.

#	Program Versions	0	0-1	1	1-2	2	2-3	3
1.	Extract Method	4	1	3		3		3
2.	Inline Method	24	1	6		6		6
3.	Introduce Var	41		41		36		43
4.	Inline Temp	12		12		12		10
5.	Reverse Conditional	1		1		1		1
6.	Replace Magic Number with Symbolic Constant			4		4		4
7.	Self Encapsulate Field	14		8		8		4
8.	Introduce Parameter Object	2		2		2		
	Subtotal	98	2	77	0	72	0	71

resulted in a number of refactoring opportunities detected by our approach. Between versions 2 and 3, some of these opportunities were applied and the methods that invoked `getProperty` were changed to invoke `getPropertyAsString`. These cases were captured by our approach by comparing the two versions, but not detected by RefFinder.

We now discuss a few refactoring types that highlight the strength of our approach as opposed to using either clone detection to detect refactoring opportunities, or tools in the literature [4, 18, 27, 46] that detect historical refactorings:

a) *Classifying refactoring involving small changes precisely*: "Self Encapsulate Field" is a refactoring that manifests itself in terms of changes to method bodies by a change from a direct field access to a call to a getter method. This small change between the before and after methods can cause a large number of similar methods to be returned by traditional threshold-based similarity approaches. Unfortunately, most of the returned results are irrelevant to the "Self Encapsulate Field" refactoring, leading to high numbers of false positives. Our approach, on the other hand, can detect a large number of "Self Encapsulate Field" refactorings with a high precision. Specifically, we detect more than 50 historical refactorings, that were not detected by RefFinder, between the Ant versions 4 and 5 with a 100% precision.

In comparison with RefFinder, we note that RefFinder has a different refactoring definition for "Self Encapsulate Field"; its definition is based on changes between two versions of a program focusing on the creation of a getter method, and does not capture the manifestation of self-encapsulate field in the methods that access it. This makes comparison between RefFinder and our approach impractical.

RefFinder Definition of Self Encapsulate Field [45]

encapsulate_field(fFullName) ∧ there are no access to the field besides the new getter and setter → self_encapsulate_field

deleted_fieldmodifier(fFullName, public) ∧
added_fieldmodifier(fFullName, private) ∧
added_getter(mGetFullName, fFullName) ∧
added_setter(mSetFullName, fFullName) → encapsulate_field

Moreover, we note that RefFinder cannot be applied within the same version, thus is unable to discover many refactoring opportunities that occur within a version.

b) *Detecting complex refactoring patterns*: "Replace Parameter with Method" transforms a method m (which invokes a method m_1 and passes its return value as an argument for another method m_2) by moving the call to m_1 into a modified version of m_2. Figure 4 shows an example. Detecting an instance of this refactoring type requires a specific definition of similarity among the caller and the callees. Our approach can achieve the precision by specifying that the difference in the numbers of method calls in the two versions of vectors for the caller (`getPrice`) is the additive inverse of that in the two versions of vectors for the callee (`discountedPrice`).

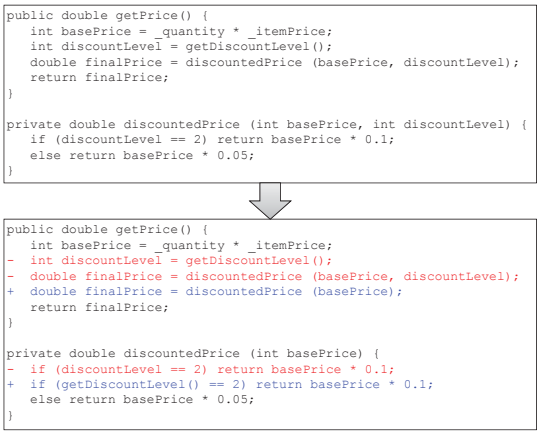

Figure 4: Example of "Replace Parameter with Method."

Table 7: Result summary for Eclipse.

Refactoring Type	No. of Results	Time
Extract Method	1310	17m21
Inline Method	527	14m7
Self Encapsulate Field	2948	17m6
Downcast Encapsulate	664	18m36
Introduce Var	22942	22m43
Inline Temp	2013	22m22
Reverse Conditional	1021	21m21
Split Temp	26	26m56
Remove Assignment to Initialization	50	26m56
Replace Magic Number	1577	24m44
Consolidate Conditional	52	59m3
Replace Nested Conditional with Guard Clauses	60	59m29
Introduce Parameter Object	325	18m57
Replace Parameter with Method	228	18m35
Hide Delegate	13	19m44
Remove Middleman	10	19m14
Replace Method with Method Object	2	18m8

The second case study evaluated the scalability of our approach by applying it on a large-scale ecosystem of projects. The results are presented in Table 7 and show that our approach can efficiently detect a broad range of refactoring types in Eclipse projects. Queries for each type of refactoring finished in about 25 minutes on average. The exceptions were "Consolidate Conditional" and "Replace nested conditionals which took 59 minutes as the complex concretizations rely on comparing features from both bytecode and source-code vectors. As a contrast, we note that for the biggest project in the first case study, Ant version 5, the queries for each type of refactorings took at most 40 seconds.

5.3 Threats to Validity & Future Work

Our approach relies on characteristic vectors, which results in a threat to construct validity stemming from whether the vectors can really represent refactorings. Although vectors ignore various information in code (e.g., the ordering or relations among program elements, the specific names of identifiers), they have been shown to be effective for code clone detection [11, 22], and the abstraction and concretization operations take the features of each type of refactorings into consideration, making the vectors more tailored for refactoring detection. However, since the vectors only count the occurrences of basic program elements and do not encode features needed for many other refactoring types in classical collections [9, 21], we will need to encode more features in the vectors to go beyond the types of refactorings we can detect now. The features encoded in the vectors are language-dependent, so are some refactoring types; so our approach may need adjustments for different languages.

Our vector abstraction and concretization operations are heuristic and learned from sample refactored code; so their accuracies are dependent on the "quality" of the sample code. We used sample code from classical collections, and aimed to ensure precise encoding of the most essential changes for each refactoring type; our results show good precisions. However, there may also be biases in the evaluation of the accuracy of the reported refactorings, since we relied on manual investigation on selected samples by students, had only compared with one previously existing tool for detecting historical refactorings, and we need a better way to evaluate the recall rates of our approach.

In the near future, we plan to extend vector operations for more refactoring types, apply our approach with more sample refactored code to more subject programs, port our implementation to other languages, conduct more systematic user studies to alleviate the above threats, and make our tool and evaluation data available for other researchers. Also, we plan to automate the definitions of abstraction and concretization operations with ideas and techniques from programming by examples [1, 33, 34] and apply detected refactoring opportunities automatically.

6. CONCLUSION

This paper presents a new vector-based approach for scalable detection of refactorings. Our approach builds on top of characteristic vectors that encode various code features. Most importantly, it extends vectors with abstraction and concretization operations to capture the features of the code changes that may be induced by a refactoring operation. Such abstraction and concretization operations can be extracted and refined based on known refactored code samples. Both refactoring opportunities (i.e, code fragments that may be restructured according to a refactoring type) and historical refactorings (i.e., code fragments that have been restructured according to a refactoring type) can be encoded via concrete and abstracted vectors. Thus, our approach reduces the problem of detecting refactorings to the problem of detecting matching vectors, which can be solved efficiently in almost linear time with respect to vector numbers.

We have implemented our approach for Java and applied the prototype to a large code base containing 200 bundle projects from the Eclipse ecosystem and about 4.5 million lines of code. Our prototype detects more than 32K instances of 17 types of refactoring opportunities in about 7 hours. We have also applied our prototype to 14 versions of 3 programs used in previous studies on refactoring detection, and found 191 instances of various types of historical refactorings across consecutive versions of the programs, with a 92% precision. Our prototype also detects more than 2.8K instances of refactoring opportunities within individual versions of the programs, with a 87% precision.

7. ACKNOWLEDGMENTS

We appreciate the students' time and efforts in evaluating our results: Joseph Chan Joo Keng, Ta Quang Trung, Liu Yang, and Zhiqing Zuo. This research is also partially supported by a NUS research grant R-252-000-553-112.

8. REFERENCES

[1] A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[2] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie. XIAO: tuning code clones at hands of engineers in practice. In *ACSAC*, pages 369–378, 2012.

[3] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *20th ACM Symposium on Computational Geometry (SoCG)*, pages 253–262, 2004.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA*, pages 166–177, 2000.

[5] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, pages 404–428, 2006.

[6] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *ICSE*, pages 397–407, 2009.

[7] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. JDeodorant: identification and application of extract class refactorings. In *ICSE*, pages 1037–1039, 2011.

[8] F. A. Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti. Software clone detection and refactoring. *ISRN Software Engineering*, online open access, 2013.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[10] L. Franklin, A. Gyori, J. Lahoda, and D. Dig. LAMBDAFICATOR: from imperative to functional programming through automated refactoring. In *ICSE*, pages 1287–1290, 2013.

[11] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE*, pages 321–330, 2008.

[12] A. Garrido and J. Meseguer. Formal specification and verification of java refactorings. In *6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 165–174, 2006.

[13] X. Ge and E. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *ICSE*, page To appear, 2014.

[14] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[15] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE TSE*, 31(2):166–181, 2005.

[16] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.

[17] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328, 2011.

[18] S. Hayashi, Y. Tsuda, and M. Saeki. Detecting occurrences of refactoring with heuristic search. In *APSEC*, pages 453–460, 2008.

[19] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. ARIES: refactoring support tool for code clone. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–4, 2005.

[20] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance*, 20(6):435–461, 2008.

[21] JetBrains. Refactoring Source Code in IntelliJ IDEA 13. `http://www.jetbrains.com/idea/webhelp/refactoring-source-code.html`.

[22] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[23] E. Jürgens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE*, pages 485–495, 2009.

[24] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE TSE*, 2002.

[25] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *WCRE*, 2006.

[26] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: memory comparison-based clone detector. In *ICSE*, pages 301–310, 2011.

[27] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *FSE*, pages 371–372, 2010.

[28] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE*, 2005.

[29] R. Lämmel. Towards generic refactoring. In *ACM SIGPLAN workshop on Rule-based programming*, pages 15–28, 2002.

[30] H. Liu, Y. Liu, G. Xue, and Y. Gao. Case study on software refactoring tactics. *IET Software*, 8(1):1–11, 2014.

[31] H. Liu, Z. Ma, W. Shao, and Z. Niu. Schedule of bad smell detection and resolution: A new way to save effort. *IEEE TSE*, 38(1):220–235, 2012.

[32] H. Liu, Z. Niu, Z. Ma, and W. Shao. Identification of generalization refactoring opportunities. *Automated Software Engineering*, 20(1):81–110, 2013.

[33] N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *ICSE*, pages 502–511, 2013.

[34] A. K. Menon, O. Tamuz, S. Gulwani, B. W. Lampson, and A. Kalai. A machine learning framework for programming by example. In *30th International Conference on Machine Learning (ICML)*, pages 187–195, 2013.

[35] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE TSE*, 30(2):126–139, Feb 2004.

[36] N. A. Milea, L. Jiang, and S.-C. Khoo. Scalable detection of missed cross-function refactorings. In *ISSTA*, pages 138–148, 2014.

[37] E. R. Murphy-Hill. Scalable, expressive, and context-sensitive code smell display. In *OOPSLA*, pages 771–772, 2008.

[38] E. R. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, 2008.

[39] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *ECOOP*, pages 552–576, 2013.

[40] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *ICSE*, pages 69–79, 2012.

[41] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE TSE*, 38(5):1008–1026, 2012.

[42] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. In *OOPSLA*, pages 302–321, 2010.

[43] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/SIGSOFT FSE*, pages 383–392, 2009.

[44] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign (UIUC), 1992.

[45] K. Prete, N. Rachatasumrit, and M. Kim. A catalogue of template refactoring rules. Technical Report ECE-TR-04161, Department of Electrical and Computer Engineering, University of Texas at Austin, 2010.

[46] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *ICSM*, pages 1–10, 2010.

[47] N. Rachatasumrit and M. Kim. An empirical investigation into the impact of refactoring on regression testing. In *ICSM*, pages 357–366, 2012.

[48] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pages 172–181, 2008.

[49] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *ISSTA*, pages 117–128, 2009.

[50] M. Schäfer and O. de Moor. Specifying and implementing refactorings. In *OOPSLA*, pages 286–301, 2010.

[51] M. Shomrat and Y. Feldman. Detecting refactored clones. In *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 502–526, 2013.

[52] Q. D. Soetens, J. Perez, and S. Demeyer. An initial investigation into change-based reconstruction of floss-refactorings. In *ICSM*, pages 384–387, 2013.

[53] C. Sun, S.-C. Khoo, and S. J. Zhang. Graph-based detection of library api imitations. In *ICSM*, pages 183–192. IEEE, 2011.

[54] R. Tairas. Clone detection and refactoring. In *OOPSLA*, pages 780–781, 2006.

[55] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *ASE*, pages 377–380, 2007.

[56] T. Tourwe and T. Mens. Identifying refactoring opportunities using logic meta programming. In *CSMR*, pages 91–100, March 2003.

[57] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities. In *CSMR*, pages 119–128, 2009.

[58] F. Van Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *19th ASE*, pages 336–339, 2004.

[59] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *ICSE*, pages 172–181, 2006.

[60] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE*, pages 231–240, 2006.

[61] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.

[62] H. Zhong, L. Zhang, and H. Mei. Inferring specifications of object oriented APIs from API source code. In *APSEC*, pages 221–228, 2008.

[63] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE*, pages 307–318, 2009.