T H E N A T I O N A L U N I V E R S I T Y
o f S I N G A P O R E

S c h o o l of C o m p u t i n g
Computing 1, 13 Computing Drive, Singapore 117417

TRC3/2014

Vector Abstraction and Concretization for Scalable Detection of Refactorings (A Technical Report)

Narcisa Andreea Milea, Lingxiao Jiang and Siau-Cheng Khoo

March 2014

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

David ROSENBLUM
Dean of School

Vector Abstraction and Concretization for Scalable Detection of Refactorings (A Technical Report)

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, but have not yet been restructured in a program), or reconstruct historical refactoring (i.e., code restructuring operations that have happened between different versions of a program). However, it remains challenging to apply those techniques to large code bases containing millions of lines of code involving many versions. In this paper, we propose a new *scalable* technique that can be used for *both identifying refactoring opportunities and historical refactoring*. The key of our technique is the design of *vector abstraction and concretization* operations that can capture the essential patterns of the code changes induced by various refactoring operations in the form of 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. We have applied the prototype 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 refactoring opportunities for all Eclipse projects, taking 25 minutes on average for each type. We have also applied the prototype to 14 versions of 3 smaller programs (JMeter, Ant, XML-Security), and detected (1) more than 2.8K refactoring opportunities within individual versions with an accuracy of about 87%, and (2) more than 190 historical refactorings across consecutive versions of the programs with an accuracy of about 92%.

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, 36], 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, but have not yet been restructured, and thus to reduce "bad smells" in code and improve the design of the code [7, 20, 32, 54, 55]; 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, 28, 45, 51, 53, 58].

An approach, which is currently unavailable, that provides consistent detection of refactoring opportunities and historic refactorings is desirable, as it enables developers to measure more accurately the refactoring efforts and progress through software evolution. Also, scalability of a refactoring detection technique still remains challenging for large code bases containing millions of lines of code. Figure 1 helps to illustrate some of the challenges in detecting refactoring opportunities and historical refactorings. The code fragments (a), (b), and (c) are detected in a program named Apache JMeter version 1, and still exist (with small variants) in the latest version 2.11 (https://jmeter.apache.org/). The code fragments (a) and (b) look similar to each other, however (a) contains extra variable declarations and a method call to `instantiate` (the underlined red parts); they may be considered as code "clones" (i.e., code fragments similar to each other, [8, 24, 26, 29, 56]) under very relaxed similarity conditions. It will be far-fetched to consider (a) and (b) as refactoring. When we also take into consideration the code fragment (c) which invokes `getVectors`, it becomes clearer that (a) can be refactored in a manner as exhibited by bundling (b) and (c) together. Specifically, if we inline in (c) the call to method `getVectors` shown in (b), the inlined code (obtained by replacing the formal parameters with the simple actual argument and removing the `return` statements) will become more similar to (a) than (c) itself.

Let us denote this inlined code by (c^I). It continues to be challenging for many code clone techniques to detect the similarity between (a) and (c^I) without incurring a significant number of false positives in their outcome [2, 11, 23, 25, 27, 42, 47]. This is so because these techniques cannot flexibly omit or emphasize specific program elements (e.g., the variable declaration and assignment) when computing similarity. The additional variable declarations and assignments in (a) do not exist in (c) or (c^I), which is in fact another kind of refactoring operation, named "inline temp." This indicates that a desired

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2014 National University of Singapore.

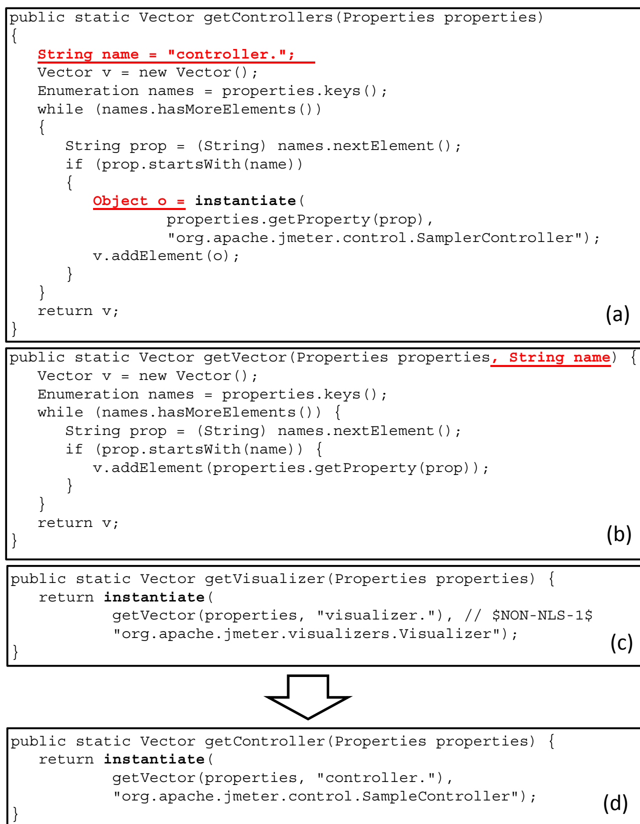


Figure 1: Sample refactoring opportunity detected from JMeter.

refactoring detection tool should possess knowledge about, as well as work flexibly with, various kinds of refactoring operations.

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 manipulates the characteristic vectors flexibly based on code change patterns induced by known refactorings. By using vectors as the representation of code and code changes, our approach reduces the problem of detecting refactorings to the problem of finding vectors matching certain conditions. Since vector-based operations can be performed in almost linear time with respect to the total number of vectors and the dimension of each vector, it becomes the a key to the scalability of our approach.

In addition to just detecting a code fragment c for refactoring, our approach also identifies the possible refactoring operation that may be applied to c by reporting a set of code fragments that may have been refactored via the same kind of refactoring operations so that a user of our approach can understand better, from the samples, how to refactor c. For the example shown in Figure 1, our approach identifies (a) as a refactoring opportunity, and at the same time, it reports both (b) and (c) as a sample to show how (a) may be refactored. Then, a user could proceed to refactor (a) in a way similar to (c) and transform (a) into (d).

We have implemented our approach for Java, generating vectors for both source code and bytecode, and extracted vector abstraction and concretization operations for 21 common types of refactoring operations. Our tool takes in the source code of a Java program, compiles it to get bytecode, inlines non-recursive method calls that invoke methods defined in the program itself for one level, and generates characteristic vectors for both the original code and inlined code. Then, for every type of refactoring operations γ, the tool applies the corresponding vector abstraction to every generated vector, uses hash-based search to cluster vectors that are identical under abstraction γ, and concretizes the vectors within clusters to identify ones that match the effect of γ.

The tool is both scalable and accurate in detecting refactorings. In a large code base comprising of more than 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 refactoring opportunities for all Eclipse projects, taking 25 minutes on average for each type. In a smaller code base containing 14 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 both automated and manual validation by 4 graduate students, we find that the detected refactorings are of high accuracies, 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 represent essential code changes needed for various types of refactoring operations as abstraction and concretization operations of vectors, which encode syntactic features of code and code changes;
- Our vector-based encoding of refactoring operations enables detection of refactoring both within the same version and across different versions of a program, so that we can detect both refactoring opportunities and historical refactorings;
- Our vector-based encoding and similarity queries for abstracted and concrete vectors enable scalable detection of refactorings;
- We have evaluated our approach on large code bases with millions of lines of code, and show scalable and accurate detection results.

The rest of the paper is organized as follows. Section 2 presents related work on refactoring. Section 3 presents our detection approach. Section 4 presents specific vector abstraction and concretization operations used in our detection approach. Section 5 presents the results of our empirical evaluation Section 7 concludes with future work.

2. RELATED WORK

This paper aims for *scalable* detection of refactoring, and 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 a number of general introduction and surveys for software refactoring and related tools [9, 22, 31, 36, 39]. Negara et al. [40] recently infer refactoring operations from continuous code changes and compare manual and automated refactorings. Some surveys and tools investigate the relations between refactoring and code clones [8, 19, 52, 56], and code clone detection has been touted as an important way to detect refactoring opportunities (Extract or Pull-Up Method in particular). Fontana et al. [8] manually refactor code clones detected by three different clone detection tools and find that certain code quality metrics are improved after the refactoring. Higo et al. [19] define several metrics for code clones and demonstrate a tool that can suggest refactoring operations for code clones. Tairas [52] visualize clones so that it may become easy to select candidates for refactoring. Van Rysselberghe and Demeyer [56] investigate three different kinds of clone detection techniques (simple line matching, parameterized matching, and metric fingerprints) and find that clones detected by different techniques may be suitable for different kinds of refactoring.

Many studies aiming for automatic detection of refactoring (besides the studies on clone detection). Many of these studies 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 [58] define various signatures based on code clone detection results to look for refactoring. Prete and Kim et al. [28, 45] 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 Ref-Finder can detect 63 kinds of refactoring in Fowler's catalog [9]. Taneja and Dig et al. [5,53] present tools (`RefactoringCrawler` and `RefacLib`) to detect refactoring between different versions of libraries. Soetens et al. [51] 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. Our detection technique is vector-based; it 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 are also studies that detect refactoring opportunities, and related to many studies on detecting "bad smells" in code [9,32,38]. Tourwe and Mens [54] use logic programming to encode several types of refactoring operations and detect possible refactoring opportunities. Meng et al. [34] 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 also 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 from their experience combining inter-procedural analysis and the expressiveness of general-purpose edits is a very hard problem. They can thus not detect changes that require moving code from one method to another or coordinating changes to multiple methods in the way our approach does. Our approach is similar to many studies on programming by examples [17, 34, 35, 41] and specification mining [43, 59–61] in the sense that we all learn from examples.

Cider [50] 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 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. [33] identify particular kinds of generalization refactoring opportunities. Our vector-based approach detects different types of refactorings and can complement those studies.

Many 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 [30] 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. [57] propose a domain specific language *JunGL* for expressing dataflow properties on a graph representation of the program. Scäfer et al. [49] improve on this and provides 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 use lambda expressions and functional operations available in Java 8. Our tool currently focuses on scalable detection only. Our tool reports refactorings together with possible refactoring results; so we believe our tool can also be improved to perform identified refactorings automatically and accurately.

3. METHODOLOGY

We explain the main steps of our approach together 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, 23] 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 [48]. Using bytecode or binary code has the benefits that many code differences only applicable to high-level languages (e.g., different syntaxes for writing `for` loops) are unified or eliminated, and potentially can help to detect more refactorings [48]. We also simulate the effect of method lining by manipulating the STs based on call relations and get inlined code, and generate vectors

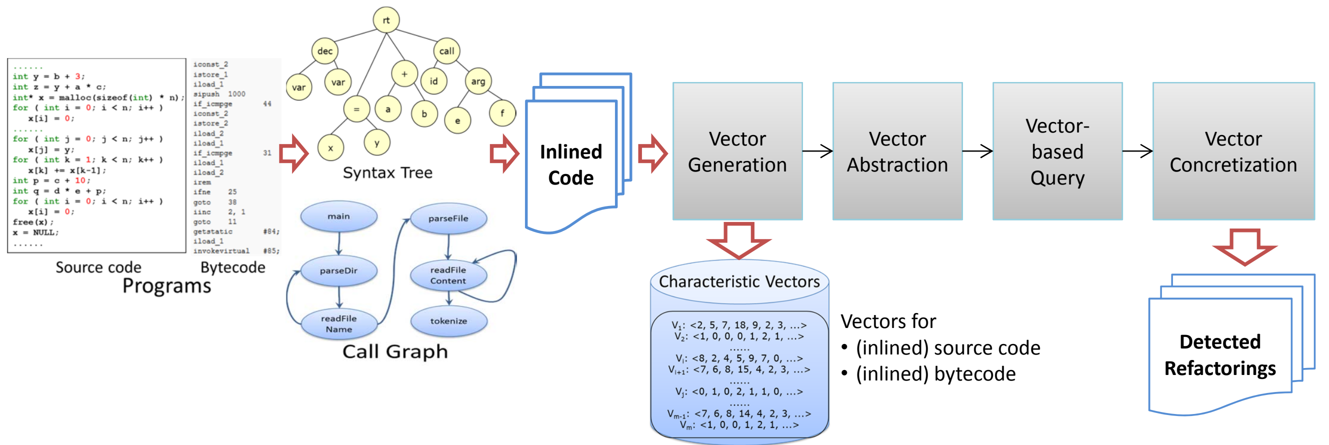


Figure 2: Overview of Our Approach.

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 refactorings γ. The particular code characteristics are semi-automatically extracted from known sample code refactored by γ (see Section 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 refactorings (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.

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, we call these vectors *base-level* characteristic vectors in this paper.

3.1 Vector Generation

In this work we use characteristic vectors for the purpose of refactoring detection. We define vectors as follows.[1]

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 [23], 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.

3.1.1 *For original 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 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 this program; "new invoc." refers to invocations of constructors (e.g., `new Vector()`).

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

Table 1: Sample partial vectors for code fragments in Figure 1

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 particular, we inline methods invoked in c in several different modes: inlining all methods invoked in c all at once, inlining all calls to each method separately, or inlining nothing. We do not inline constructor and api invocations. 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.

[1]We use the term "characteristic vector" and "vector" interchangeably in this paper.

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 `returns` 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 `getVector` 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 certain method extraction and inlining [37]. This paper employs the same idea, but extends 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 byte code too. However, the idea of encoding method inlining as vector operations can be generally applicable to different programming languages.

For easier discussion in the following, 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 change patterns induced by a kind of refactoring operation in the form of vectors as accurate as possible, and abstract away (or eliminate) the changes, while keeping 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 representations.

Each type of refactorings has a different abstraction since they often induce different code changes. We use ${}^{\gamma}\!\mathcal{A}$ to denote the abstraction operation for a type of refactorings γ. Then, ${}^{\gamma}\!\mathcal{A}(v)$ means to apply the abstraction onto a vector v, and ${}^{\gamma}\psi$ denotes the resulting abstracted vector. Conceptually, the relations among base code, inlined code, and various kinds of vectors with respect to a refactoring γ are illustrated by the "Query" portion on the left side of Figure 3. Any piece of base code c can have more than one inlined code c^I; the base vector v_c for c can be abstracted to an abstracted base vector ${}^{\gamma}\psi_c$; and the inlined vector v_c^I can be abstracted to an abstracted inlined vector ${}^{\gamma}\psi_c^I$. It is possible that ${}^{\gamma}\psi_c^I$ may be the same as v_c^I and/or v_c.

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

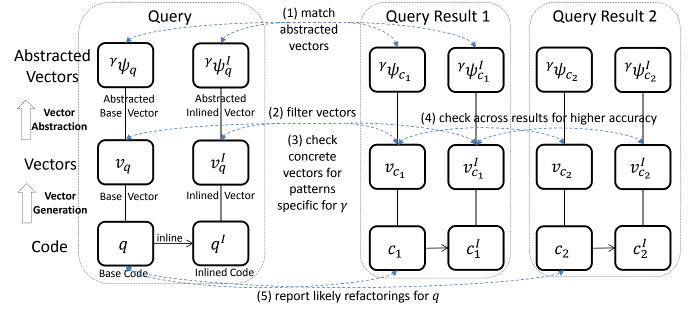


Figure 3: Given a piece of code q, search for code in a code base that is similar to q if q is refactored via a refactoring operation γ.

DEFINITION 3.2 (VECTOR SUBSTITUTION). *Given a vector v and a set of mapping from features to counts ($\mathcal{F} = \{f_j \mapsto n_j\}$), the* vector substitution *is denoted by $v(\mathcal{F})$; it generates a new vector v', such that :*

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

We use either n_j or $\mathcal{F}[j]$ to 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 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 to define the abstraction operation ${}^{\gamma}\!\mathcal{A}$ for γ that can eliminate the feature changes that may be induced by γ into an arbitrary vector v. The abstracted vector for v is denoted by either ${}^{\gamma}\!\mathcal{A}(v)$ or ${}^{\gamma}\psi$. The following pseudo-algorithmic rules describe how ${}^{\gamma}\psi$ is generated, based on $\delta(v_1, v_2)$. These rules are mostly the same for either source code or 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 $\Sigma_{i=1}^{s} n_{d_i} = 0$, 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

$1 \leq d_1 \leq d_2 \ldots \leq d_{s-1} \leq d_s \leq K$, such that $n_{d_1} = n_{d_2} = \ldots = n_{d_s}$, 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.

(IV) there may be multiple subsets of $\mathcal{D}$ satisfying the above conditions; if the subsets are disjoint, we carry out the abstraction as above; if the subsets overlap, we then apply our manual intervention to identify suitable abstractions heuristically.

(V) ${}^{\gamma}\psi[i] = 0$, otherwise.

For example, we can define the vector abstraction for the kind of refactoring operations 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 ${}^{\gamma}\mathcal{A}$ 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.

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

Table 2: Sample abstracted vectors for vectors in Table 1

When more than one pair of sample code are provided for a refactoring operation γ, we can refine the extracted abstraction for γ to represent the most essential code change patterns 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, 59–61] and programming by examples [17, 34, 35, 41], 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 a particular kind 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. Thus, we can use hash-based matching techniques [3] to find vectors that are either matching exactly or very similar to each other [11, 23].

We tail our 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 in a code base?* As illustrated in the step (1) in Figure 3, we perform queries on abstracted vectors for inlined code: each abstracted vector (either ψ_q or ψ_q^I depending on the abstraction defined for γ (see Section 3.2)) is used as a query against all other abstracted vectors from the code base to identify the ones matching the query.

Not all matching vectors can be refactorings; some may be just code that is syntactically similar to each other. Thus, we also apply filters (the step (2) in Figure 3) to remove 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 essentially the same syntactically, and their inlined versions are the same as well. c_1 is simply a clone of q and does 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 filer 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. Vector-based querying and filtering can be performed in almost linear time with respect to the total number of vectors. This is a key to the scalability of our approach.

3.4 Vector Concretization

After steps (1) and (2) as above, we have a set of filtered query results for each piece of code used as query. There must be some syntactic differences between the query and each of the query results and differences must exist in either between v_q and v_{c_1} or between v_q^I and $v_{c_1}^I$ (see Figure 3). This concretization phase corresponds to steps (3) and (4) in Figure 3; it performs several kinds of checks on those concrete vectors to enhance the likelihood for reported query results to be true refactorings.

The first kind of checks is to make sure the *differences* among certain concrete vectors indeed subsume the differences ($\mathcal{D}$, see Section 3.2) that may be induced by a kind of refactoring operations γ. This is useful for reducing false positives since different refactoring operations may in fact change same features in code and thus may have similar

abstractions.

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 swap 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 on the *common parts* among certain concrete vectors indeed subsume the common parts (v^δ, see Section 3.2) that can represent the contexts needed for γ.

We also check between the query and the query results and across the results to improve the reliability of the final results. For certain types of refactorings, we also manually add special checks for the types of refactorings (see Section 4), based on our understanding of the code changes involved in the refactorings, to help remove likely 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 following pseudo-algorithm describes the checks performed by concretization:

(I) Calculate the vector difference between v^I_q and each $v^I_{c_i}$: $\delta(v^I_q, v^I_{c_i}) = (v^{I\delta}_q, m^I_q, \mathcal{D}^I_q)$. Check them against $(v^\delta, m, \mathcal{D})$, and remove the query result c_i if one of the following conditions is true:
  (1) if $\exists(f_i \mapsto n_i) \in \mathcal{D}, s.t.\ (n_i < 0) \wedge (v^I_q[i] < |n_i|)$, 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 $\mathcal{D}^I_q$ does not contain f_i or $|\mathcal{D}^I_q[f_i]| < |n_i|$. This indicates that the changes between v^I_q and $v^I_{c_i}$ may be too few in comparison with the changes induced by γ to be considered as a real case for γ;
  (3) if $\exists i \in 1..K, s.t.\ (v^I_q[i] < v^\delta[i]) \vee (v^I_{c_i}[i] < v^\delta[i])$, 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^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 results against each other when there are still more than one result at this stage. We remove all of the query results if the following condition is true:
  (1) $\exists i, j, s.t.\ i \neq j \wedge v_{c_i} \neq v_{c_j}$;
  Effectively, all base code corresponding to the query results should be syntactic clones of each other. This is to prevent confusing situations where different query results may indicate to users different refactorings. Thus, users can have higher confidence that the refactoring operation indicated by the final results can be applied to the query code.

Finally, the code corresponding to the query and filtered query results is 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."$\mapsto -2$}). The contexts for the case also obviously match the contexts extracted from the case itself. 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. By far, the vectors used in this paper only capture code features related 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, etc.) Thus, our approach by far 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 types of refactorings we can detect are mostly in classical collections [9, 22]. Table 3 lists sample abstraction and concretization operations extracted by our approach for the types of refactoring operations that we can detect. The sample refactored code we use for defining abstraction and concretization is all from the classical collections [9, 22]. Due to space limitation, we use simplified notations and descriptions for illustration, instead of rigorous vector-based operations. Abstraction rules that do not change the values for a feature are not shown; some concretization rules that are the same for all types of refactorings, as described in the various subsections in Section 3 are also omitted.

5. EMPIRICAL EVALUATION

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

<table>
<tr><th>#</th><th>Refactoring</th><th>Abstractions ($^{Y}A(v)$ or simply ψ)</th><th>Concretization Checks</th><th>Descriptions</th></tr>
<tr><td>1.</td><td>Extract Method</td><td>$\psi[load] = \psi[store] = 0$
$\psi[constant] = 0$</td><td>$\nexists$ v_i inlined into v_c s.t. $v_q == v_i$
(remove simple extraction methods)</td><td></td></tr>
<tr><td>2.</td><td>Inline Method</td><td>$\psi[load] = \psi[store] = 0$
$\psi[constant] = 0$</td><td>$\nexists$ v_i inlined into v_q s.t. $v_q == v_i$
(remove simple extraction methods)</td><td></td></tr>
<tr><td>3.</td><td>Inline Temp</td><td>$\psi[load] = \psi[store] = 0$</td><td>$v_q[load] - v_c[load] > 0$ &&
$v_q[load] - v_c[load] == v_q[store] - v_c[store] ==$
$v_q^S[var_declaration] - v_c^S[var_declaration]$</td><td>Remove the declaration of a temporary variable, and replace the use of the variable with the value of the variable.</td></tr>
<tr><td>4.</td><td>Introduce Explaining Variable</td><td>$\psi[load] = \psi[store] = 0$</td><td>$v_q[load] - v_c[load] < 0$ &&
$v_q[load] - v_c[load] == v_q[store] - v_c[store] ==$
$v_q^S[var_declaration] - v_c^S[var_declaration]$</td><td>Extract a complicated expression into a temporary variable.</td></tr>
<tr><td>5.</td><td>Split Temporary Variable</td><td>$\psi[load] = v[load] + \sum v[load_i]$
$\psi[store] = v[store] + \sum v[store_i]$
$\psi[load_i] = \psi[store_i] = 0$
$i \in \{0, 1, 2, 3\}$</td><td>$v_c^S[variable_declaration_statement] >= 2$ &&
$(v_q^S[assignment] - v_c^S[assignment]) ==$
$-(v_q^S[var_declaration] - v_c^S[var_declaration])$</td><td>Transform multiple assignments to a temporary variable into separate variable declarations for each assignment.</td></tr>
<tr><td>6.</td><td>Replace Method With Method Object</td><td>$\psi[load] = \psi[store] = 0$
$\psi[getfield] = \psi[putfield] = 0$
$\psi[new] = \psi[invoke_init] = 0$</td><td>$\exists$ v_i inlined into v_c s.t.
$(\sum v_i[f] - \sum v_q[f]) == (v_i[getfield] + v_i[putfield])$, $f \in \{0,.. v_q.length\}$
$(v_i[getfield] - v_q[getfield] > 0)$
$(v_c^I[new] - v_q[new] > 0)$</td><td>Transform a method into its own object so that all the local variables become fields. Abstraction involved ignoring "new" operators and encapsulation.</td></tr>
<tr><td>7.</td><td>Self Encapsulate Field</td><td>$\psi[aload_0] = 0$

(aload_0 is used for loading "this" on the stack)</td><td>$v_q[getfield] - v_c^I[getfield] > 0$ // $\exists$ 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</td><td>Replace direct accesses to a field with a getter method</td></tr>
<tr><td>8.</td><td>Replace Magic Number with Symbolic Constant</td><td>No abstraction needed as both are represented by the same bytecode.</td><td>$\sum v_q^S[literal] - \sum v_c^S[literal] > 0$ // query has more magic numbers
$literal \in \{ string_literal, boolean_literal, num_literal \}$
$v_q^S[simple_name] - v_c^S[simple_name] < 0$</td><td>Replace constants used in code with symbolic names for easier maintenance</td></tr>
<tr><td>9.</td><td>Replace Magic Number with query method</td><td>No abstraction needed as both are represented by the same bytecode.</td><td>$\exists$ v_i inlined into v_c s.t.
$\sum v_i^S[op] == 1$, $op \in \{ string_literal, boolean_literal, num_literal \}$
$v_i^S[return] == 1$
$v_i^S[totalCount] == 2$</td><td>Replace constants used in code with a getter method that returns the constants.</td></tr>
<tr><td>10.</td><td>Reverse Conditional</td><td>$\psi[eq] = v[eq] + v[neq]$
$\psi[lt] = v[lt] + \sum v[opp]$
$\psi[neq] = \psi[opp] = 0$
$opp \in \{ gt, ge, le \}$</td><td>$\exists$ $cond \in \{ eq, neq \}$ or $opp \in \{ lt, gt, ge, le \}$ s.t.
$v_q[cond] - v_c[cond] != 0$ ||
$v_q[opp] - v_c[opp] != 0$</td><td>Treat "==" the same as "!="
Treat "<=" the same as ">", ">=", and "<"</td></tr>
<tr><td>11.</td><td>Encapsulate Downcast</td><td>$\psi[checkcast] = 0$
(ignore type casts)</td><td>$v_q[methodinvoke_checkcast] - v_c[methodinvoke_checkcast] > 0$</td><td>Encapsulate type cast operations into a separate method returning the casted type</td></tr>
</table>

Table 3: Sample abstraction and concretization operations for various refactorings. In addition to the notations used in Section 3, vectors superscripted with "S" are vectors generated from source code, while vectors without the "S" superscript 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.

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 65KLOC. The projects have been selected in order to compare with state-of-the-art in detecting historical refactorings, RefFinder [46]. Thus, for each of the subject programs we have performed experiments to find both the number of historical refactorings between each consecutive versions and the number of opportunities within each version.

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.).

We also tested our aproach on a set of examples taken from Fowlers catalog and found our aproach can succesufuly detect all defined types of refactorings correctly, thus showing 100% recall.

5.2 Detection Results

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. The accuracy of the historical refactorings is 92% while the accuracy of the opportunities for refactoring is 87%.

The results of the experiments performed in the first case study are shown in Figure 4, Figure 5, and Figure 6. Each row in the figures shows the results obtained for one type of refactoring query. Each column with header name a single number (e.g. 1) shows the refactoring opportunities within a version a the project while each column with header a range of numbers (e.g. 0-1) shows the number of detected historical refactorings between two versions.

As can be seen from the figures, the number of refactoring oportunities detected evolves 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 refactoring that has only been applied to parts of the project. A decrease in the numbers can be indicative of the fact 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 versions 2 and 3 for the refactoring Encapsulate Downcast (Figure 4 row with id 6. columns with ids 2, 2-3, and 3. In versions 0 to 2 of Jmeter a large number of methods invoked

#	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

Figure 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	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 for ant								1	1		1
	Subtotal	71	4	166	3	168	0	167	23	539	99	611

Figure 5: Result Summary for Ant.

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

The experiment results were validated independently by a group of 4 graduate students with good knowledge of Java and refactoring. Out of the total of 2882 refactoring opportunities detected within all versions of the 3 subject programs, we chose for validation the refactoring opportunities within the first version of each subject program. This amounted to 276 refactoring opportunities that were validated. Among these refactoring opportunities, a method can be counted multiple times if it appears in multiple categories and thus different refactoring types can be applied. The report inspectors were required to verify that each of the detected refactorings is correctly classified. A result was counted as a false positive if any of the students considered it as a false positive. A reason used by one ispector to consider a result as false positive is "When they extract the whole method, can that be considered as an Extract Method Refactoring?". Overall, the inspectors showed 35 false positives giving 87% accuracy.

We also validated all the historical refactorings detected. This validation was performed by the authors as well as by the students. The aim was to verify that the classification of an actual code change between two versions is correctly classified by our approach. To this aim we validated 191

#	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

Figure 6: Result Summary for XML-Security.

Type of results	Validated results	Accuracy
Refactoring Oportunities	276	87%
Refactoring Historical	191	92.6%

Table 4: Accuarcy in detecting refactoring

code changes between different versions and found 14 false positives which resulted in 92.6% accuracy.

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, 28, 45] 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 (aka., high number of false positives.) Our approach, on the other hand, can detect a large number of "Self Encapsulate Field" refactoring opportunities with high accuracy. Specifically, we detect more than 50 historical refactorings, that were not detected by state of the art RefFinder, between Ant versions 4 and 5 with 100% accuracy.

In comparison with RefFinder – the state of the art threshold-based approach, 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 get 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.

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.

RefFinder Definition of Self Encapsulate Field

encapsulate_field(fFullName) $\wedge$ there are no access to the field besides the new getter and setter $\rightarrow$ self_encapsulate_field

deleted_fieldmodifier(fFullName, public) $\wedge$
added_fieldmodifier(fFullName, private) $\wedge$
added_getter(mGetFullName, fFullName) $\wedge$
added_setter(mSetFullName, fFullName) $\rightarrow$ encapsulate_field

b) *Detecting complex refactoring patterns*: "Replace Parameter with Method" is a refactoring that involves transforming a method m that invokes a method m_1 then passes the result as an argument for another method m_2, by moving the call to m_1 into m_2. An example of this is shown in Figure 7. Detecting an opportunity for refactoring of this type re-

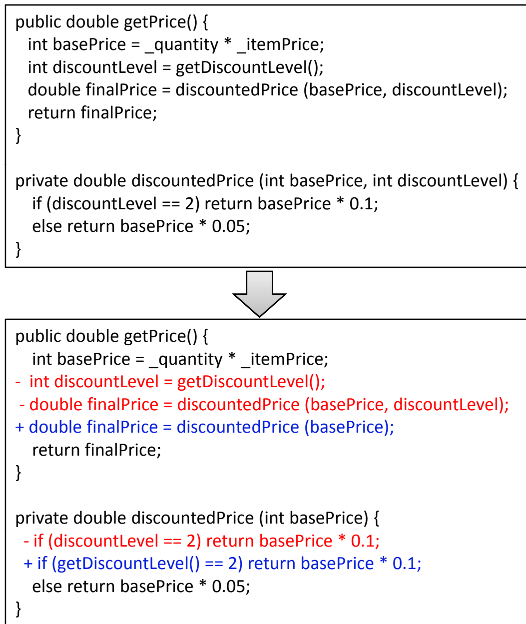


Figure 7: Example of replace parameter with method.

quires a very specific definition of similarity between the caller methods and also between the called methods. Our approach can encode and detect this type of refactoring accurately, by specifying that the difference between the vectors of the caller methods (*getPrice*), in terms of the number of method invocation features, is the reverse of the difference between the vectors of the invoked methods (*discountedPrice*).

The second case study evaluated the scalability of our approach by applying it on a large-scale ecosystem of projects, Eclipse. Given the gigantic size of the system, the number of within version refactoring opportunities detected is expected to be huge. The results are presented in Figure 8. They show that our approach can both scale to a very large project and detect a broad range of refactoring types. For each refactoring type, our system takes about 25 minutes on average to complete. Specifically, we note that around 23K of "Introduce Explaining Variable" refactoring opportunities were discovered in less than 23 minutes. Such a speedy return of results is unattainable by any of the existing refactoring detection systems – in fact, if they were to scale to the level of comparing two versions of Eclipse, then most systems would have taken hours to complete detection of many of these refactorings [45, 46]. Lastly, we note that "Consolidate Conditional" and "Replace nested conditionals" each took about 59 minutes to complete. A plausible reason is that the abstracted vectors created for these refactoring operations are rather coarse, resulting in the formation of large clusters, thus needing a large number of comparisons at the concretization stage.

In order to understand how our system scales we compared the results from our first case study with the results for Eclipse and the results reported in [45]. In our first case study, the average size of the subject programs was 33KLOC and a query took on average 34 seconds. The total time to detect all *opportunities* for refactoring and all *historical* refactoring within and between the 16 versions of Ant/Jmeter/XMLSec took slightly more than 3 hours. This can be faster than [45]

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

Figure 8: Result Summary for Eclipse.

based on the results reported in [45] for projects of similar size to the ones in our case study. [45] can however detect more types of historical refactoring while we can detect opportunities for refactoring. The size of Eclipse on the other hand is 4,500KLOC and each query took on average 25 minutes. The results of comparing Eclipse and the first case study are encouraging as they show that time increases in a sub-linear manner with respect to the increase in project size. Specifically, although Eclipse is more than 100 times larger than the average project in the first case study, each query for Eclipse took less than 50 times more time. A plausible reason for the sub-linear relation is that the overhead of reading the vectors, creating the LSH structure may not increase linearly with respect to the increase in project size.

6. THREATS TO VALIDITY

The main threat to external validity is the generalization of the results. As described in Section 5.1, this paper presents the results of applying our approach to open source programs written in Java. As it has been shown that open source teams work in fundamentally different ways from proprietary software teams [21], we are not certain how many refactoring opportunities there will be in proprietary software, both within and between versions. Nevertheless, the degree of accuracy obtained from our approach should not deviate much. Moreover, our approach works well on a group of methods linked via inlining; we cannot do a simple generalization of this approach to refactoring operations that involve entities beyond methods, such as classes.

The main threat to internal validity is the correctness of the implementation of refactoring operations. Our implementation is based on the number of occurrences of program elements in the code and does not consider the data or control flow dependency between such elements. It makes it difficult to validate the correctness of the abstract and concretized vectors used in defining the refactoring operation. As future work we plan to investigate augmenting the present representation with the use of bag of words as well as data and control dependency information for more precise definitions of refactoring operations.

A threat to construct validity stems from using limited

examples from Fowler's book to define the abstractions and concretizations. This approach might lead to overspecific abstractions and concretizations which might in turn influence the recall of our approach. To mitigate this threat, future work plans to infer from multiple examples the definitions of refactoring operations. A second threat to construct validity stems from the use of Wala to obtain the features of vectors. Specifically, in obtaining the callgraph and program dependence graphs, Wala transforms a stack-based program (Java bytecode) to an intermediate stack-less representation. This results in the exclusion of some statements which are available in the subject program's source code. Specifically, these statements are translated to bytecode that only involves "load on stack" and/or "store on stack" and does not involve method calls, field accesses, or binary operations, etc. Fortunately, we find these statements to have contributed little or none to the core functionality of the code. One such example of statement lost during the transformation from a stack-based program to an intermediate stack-less representation is "a=a;" which is equivalent in bytecode to one "load on stack" and one "store on stack" operation. Although the features of the excluded statements are not captured by the vectors the information in them is however captured by the pointer analysis performed by Wala for the construction of the callgraphs and performing method call resolution.

Yet another threat to validity is the setting of a specific threshold (as defined in 3.3, *FilterSmall*) for Extract and Inline Method refactorings. Setting the threshold to 10 enables filtering of small and irrelevant code, however, we have yet to explore its impact on the accuracy measures.

7. CONCLUSION AND FUTURE WORK

This paper presents a new vector-based approach for scalable detection of refactorings. Our approach builds on the 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 as concrete and abstracted vectors. Thus, we reduce the problem of detecting refactoring to the problem of detecting matching vectors in our approach, which can be efficiently carried out in almost linear time with respect to the size of given code bases.

We have implemented our approach for Java. We have applied the prototype to a large code base containing 200 bundle projects from the Eclipse ecosystem (e.g., Eclipse JDT, Eclipse PDE, Apache Commons, Hamcrest, etc.) and about 4.5 million lines of code. Our prototype detects more than 32K instances of 17 types of refactoring opportunities in the code base in about 7 hours in total. We have also applied our prototype to 16 versions of 3 programs used in previous studies on refactoring detection. Our prototype reports 191 instances of various types of historical refactorings across consecutive versions of the programs, about 92% of them are accurate. Our prototype also detects more than 2.8K instances of refactoring opportunities within individual versions of the programs. Manual validation by 4 graduate students show that the detected refactoring opportunities can have a high accuracy of about 87%.

In near future, we plan to automate the definitions of abstraction and concretization operations with ideas and techniques from programming by examples [1, 34, 35] and also apply detected refactoring opportunities automatically.

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] J. Howison, K. Inoue, and K. Crowston. Social dynamics of free and open source team communications. *International Federation for Information Processing Digital Library*, 21:203(1), 2009.
[22] JetBrains. Refactoring Source Code in IntelliJ IDEA 13. http://www.jetbrains.com/idea/webhelp/refactoring-source-code.html.
[23] 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.
[24] E. Jürgens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE*, pages 485–495, 2009.

[25] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE TSE*, 2002.
[26] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *WCRE*, 2006.
[27] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: memory comparison-based clone detector. In *ICSE*, pages 301–310, 2011.
[28] 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.
[29] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE*, 2005.
[30] R. Lämmel. Towards generic refactoring. In *ACM SIGPLAN workshop on Rule-based programming*, pages 15–28, 2002.
[31] H. Liu, Y. Liu, G. Xue, and Y. Gao. Case study on software refactoring tactics. *IET Software*, 8(1):1–11, 2014.
[32] 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.
[33] H. Liu, Z. Niu, Z. Ma, and W. Shao. Identification of generalization refactoring opportunities. *Automated Software Engineering*, 20(1):81–110, 2013.
[34] N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *ICSE*, pages 502–511, 2013.
[35] 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.
[36] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE TSE*, 30(2):126–139, Feb 2004.
[37] N. A. Milea, L. Jiang, and S.-C. Khoo. Scalable detection of missed cross-function refactorings. In *NUS Technical Report*, 2014.
[38] E. R. Murphy-Hill. Scalable, expressive, and context-sensitive code smell display. In *OOPSLA*, pages 771–772, 2008.
[39] E. R. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, 2008.
[40] 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.
[41] 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.
[42] 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.
[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*, 1992.
[45] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *ICSM*, pages 1–10, 2010.
[46] N. Rachatasumrit and M. Kim. An empirical investigation into the impact of refactoring on regression testing. In *ICSM*, pages 357–366, 2012.
[47] 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.
[48] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *ISSTA*, pages 117–128, 2009.
[49] M. Schäfer and O. de Moor. Specifying and implementing refactorings. In *OOPSLA*, pages 286–301, 2010.
[50] M. Shomrat and Y. Feldman. Detecting refactored clones. In *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 502–526, 2013.
[51] Q. D. Soetens, J. Perez, and S. Demeyer. An initial investigation into change-based reconstruction of floss-refactorings. In *ICSM*, pages 384–387, 2013.
[52] R. Tairas. Clone detection and refactoring. In *OOPSLA*, pages 780–781, 2006.
[53] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *ASE*, pages 377–380, 2007.
[54] T. Tourwe and T. Mens. Identifying refactoring opportunities using logic meta programming. In *CSMR*, pages 91–100, March 2003.
[55] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities. In *CSMR*, pages 119–128, 2009.
[56] F. Van Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *19th ASE*, pages 336–339, 2004.
[57] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *ICSE*, pages 172–181, 2006.
[58] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE*, pages 231–240, 2006.
[59] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.
[60] H. Zhong, L. Zhang, and H. Mei. Inferring specifications of object oriented apis from api source code. In *APSEC*, pages 221–228, 2008.
[61] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language api documentation. In *ASE*, pages 307–318, 2009.

APPENDIX

Examples of refactorings detected

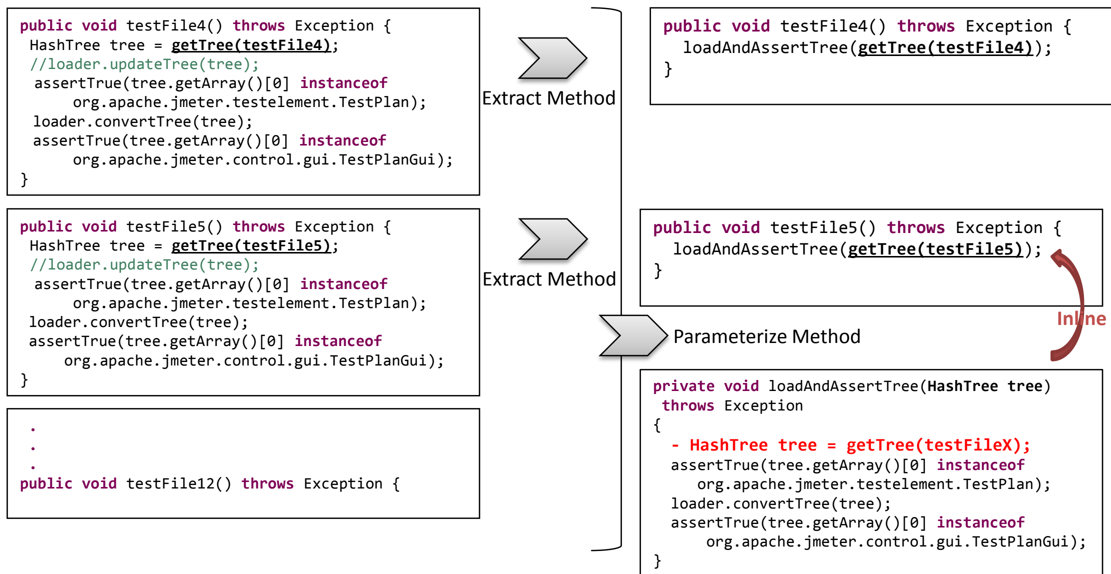


Figure 9: Example of extract method. Example also shows how clustering individual refactorings can provide definitions for complex refactorings such as “Parameterize method”. In a “Parameterize Method” refactoring several methods that do similar things but with different values contained in the method body are transformed by creating one method that uses a parameter for the different values.

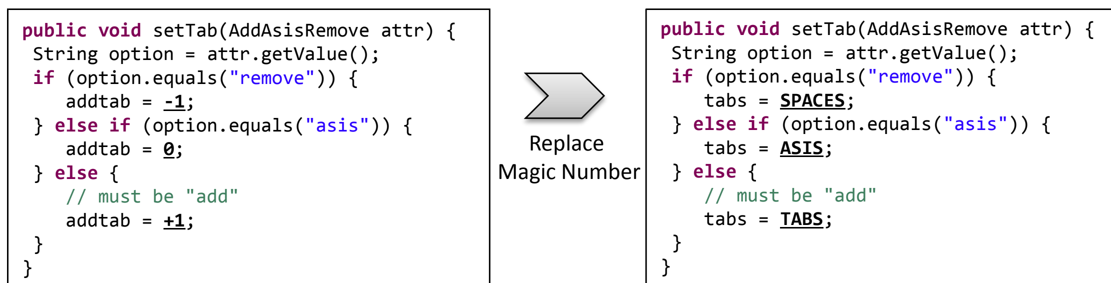


Figure 10: Example of replace magic number with symbolic constant.

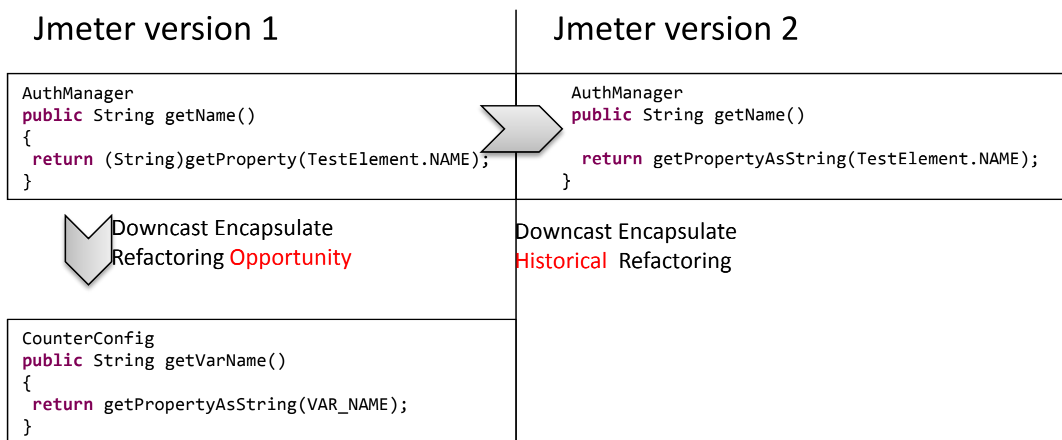
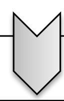


Figure 11: Example of encapsulate downcast.

```
public Key getKey(String alias, char[] password)
 throws NoSuchAlgorithmException, UnrecoverableKeyException {
 try {
  KeyElement keyElement = new KeyElement(
           this.getKeyEntryElement(alias), this._baseURI);
   return keyElement.unwrap(password);
  } catch (XMLSecurityException ex) {
    throw new UnrecoverableKeyException(ex.getMessage());
  }
}
```


```
public Certificate[] getCertificateChain(String alias) {
 try {
   Element keyElement = this.getKeyEntryElement(alias);
   + if (keyElement != null) {
     KeyElement ke = new KeyElement(keyElement, this._baseURI);
     return ke.getCertificateChain(alias);
   }
 } catch (XMLSecurityException ex) {
   ex.printStackTrace();
 }
 return null;
}
```

Figure 12: Example of introduce explaining variable.

```
X509Data
public int lengthUnknownElement() {
  NodeList nl = this._constructionElement.getChildNodes();
  int result = 0;
  for (int i = 0; i < nl.getLength(); i++) {
  Node n = nl.item(i);
  if ((n.getNodeType() == Node.ELEMENT_NODE)
       && !n.getNamespaceURI().
          equals(Constants.SignatureSpecNS)) {
      result += 1;
    }
  }
  return result;
}
```

```
KeyInfo
public int lengthUnknownElement() {
  int res = 0;
  NodeList nl = this._constructionElement.getChildNodes();
  for (int i = 0; i < nl.getLength(); i++) {
  Node current = nl.item(i);
  /**
   * @todo using this method, we don't see unknown Elements
   *  from Signature NS; revisit
   */
  if ((current.getNodeType() == Node.ELEMENT_NODE)
       && current.getNamespaceURI().
          equals(Constants.SignatureSpecNS)) {
         res++;
    }
  }
  return res;
}
```


Figure 13: Example of reverse conditional.