SCALABLE DETECTION OF SOFTWARE REFACTORING

DOCTORAL THESIS

NARCISA ANDREEA MILEA

NATIONAL UNIVERSITY OF SINGAPORE

August 2015

SCALABLE DETECTION OF SOFTWARE REFACTORING

NARCISA ANDREEA MILEA

B. Eng., UNIVERSITY "POLITEHNICA" OF BUCHAREST, ROMANIA

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE, SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

AUGUST 2015

To my beloved partner in crime and my ever lovely sister.

DECLARATION

I hereby declare that the thesis is my original work and it has been written by me in its entirety.

I have duly acknowledged all the sources of information which have been used in the thesis.

No portion of the work reported in this thesis has been submitted in support of an application for another degree of qualification at National University of Singapore or any other institution of learning.

Narcisa Milea, 30 August 2015

Abstract

The process of large-scale software development and maintenance can be unpredictable, with high costs, and a bug-prone and fragile output at times. It is estimated that 90% of the total cost of a software project is spent on evolution and maintenance, causing some authors to call it the legacy crisis (such as [109]). Refactoring is widely believed to have a positive impact on these costs. Without refactoring the gain in development costs can be easily lost in the form of increased maintenance costs [14]. Refactoring can also benefit development costs, as shown by eXtreme Programming [13] which also advocates relentless refactoring throughout a project's life cycle in order to eliminate redundancy, remove unused functionality, and restructure obsolete designs. Refactoring also enables changes during the life cycle of a software system by updating its structure and design and is crucial to the success of a system.

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 consistently apply those techniques to large code bases containing millions of lines of code involving many versions. Besides scale, the main challenge is that refactoring may change the syntactic shape of a program while preserving functional equivalence. One particularly complex change involves moving code fragments across function boundaries for refactorings such as *Extract Method, Preserve Whole Object*, etc. Detecting such changes accurately requires that a refactoring detection tool should possess knowledge about precise code change patterns induced by refactoring and should *flexibly omit or emphasize specific program elements* when computing similarity between two code fragments.

This research thesis aims at addressing this challenge. The thesis asserts that *efficient and effective detection of code fragments related by refactoring operations within large*

software can be achieved by performing abstraction and algebraic operations over high-dimensional vectors representing the code, potentially leading to great saving in software maintenance costs.

To this end, we propose a new *scalable* approach that can be used for *identifying both refactoring opportunities and historical refactoring*. The key technique in our approach is the design of efficient *vector inlining* operations that simulate the effect of method inlining among code fragments, as well as the introduction of *vector abstraction and concretization* operations to capture the essential patterns of the code changes induced by various refactoring operations. Thus, the obnoxious scalability problem can be effectively resolved by reducing the problem of refactoring identification to the problem of vector matching.

We have implemented our technique for Java and have ran the prototype on 200 bundle projects from the Eclipse ecosystem containing 4.5 million lines of code. Our prototype reports in total more than 32K instances of 17 types of 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 (i) more than 2.8K refactoring opportunities within individual versions with an accuracy of about 87%, and (ii) more than 190 historical refactorings across consecutive versions of the programs with an accuracy of about 92%.

Acknowledgements

As I write these lines, I find myself overwhelmed by the prospect of concluding such a defining chapter of my life. A chapter in a country far away that has surprised me and taught me so much. I am grateful to all the people that have walked alongside me in this journey pushing me forward along the way.

I would first like to thank my advisor, Professor Khoo Siau-Cheng, for his courage in undertaking the role of guiding me in the research world. His patience and kindness when I was slowly progressing, his knowledge and insights when trying to tackle elusive targets, his advice on what questions to ask, his advice on how to present my ideas clearly and my slowly improving presenting skills, his enthusiasm and positivity thorough moments of stumbling upon research ideas, and his unique style of motivating me have been fundamental to this work.

I am also forever grateful to Dr. Lingxiao Jiang for a prosperous research collaboration. His advice, knowledge, and fantastic proficiency in writing academic papers have taught so much about formalizing ideas and algorithms. His record speed in writing is a line in the sand that I strive to cross some day. I am indebted for his dedication and late nights before deadlines, his advice and ideas during meetings that many times resulted in me leaving with a head full of research dreams. Moreover Professor's Jiang fantastic

programming skills have motivated me to improve mine. I am also thankful to my Ph.D. committee, Dr. David S. Rosenblum and Dr. Stan Jarzabek, for their helpful comments throughout my Ph.D. candidature.

The Programming Languages Lab at National University of Singapore provided a fantastic learning and social environment. Special thanks to my dear and lovely Peruvian friend Yamilet Rosario Serrano Llerana for her contribution to the sharing component or our smart group. Birthdays and cakes were never forgotten. I also feel privileged to have met Professor Chin Wei Ngan, who taught me to love functional languages(OCaml rocks!), broadened my knowledge of separation logic and formal methods, and introduced me to his research group. I was fortunate to work with amazing colleagues. I am grateful and would like to thank Andreea, Ashankaya, Behnaz, Chanh, Chengnian, Cristi, Cristina, David, Loc, Sandeep, Trung, Zhiqiang for the wonderful time in the lab and the discussions during meetings about research or whatever.

I would like to express my gratitude to the following report inspectors, who have spent much time in validating and evaluating all our experimental results: Chenhong Xie, Joseph Chan Joo Keng, Liu Yang, Lucia, Ta Quang Trung and Zhiqiang Zuo.

To friends in Romania and Singapore I am immensely grateful for your support. To Andrei, Bogdan, Claudia, Cristina, Delia, Dumi, Paew, Marian, Mihai a heartfelt thank you for your friendship, for taking me to the best places in and out of Singapore, and for being my Romanian family away from home. To beloved Andreea, Cristina, Behnaz, Yamilet. I owe you my gratitude for some of the best moments of my time in Singapore, for the memorable talks about nothing in particular, for all the laughter, and for simply helping in making me feel cared for in Singapore. I am also grateful for discovering yoga which has helped me relax my mind, for my yoga friends who have been a source of inspiration, and for the yoga teacher that pushed me to give my best and to see how (im)possibly high we should set our goals.

To my family I love you all! A big thank you to my adorable and bubbly sister,

Diana, for many hours spent on Skype with a slightly neurotic PhD student. I know it was not easy. Your positivity and laughter helped keep my morale high even in dark hours when nothing seemed to be working. I would also like to thank my parents and my beloved grandmother for raising me in a nurturing and creative environment, for the gift of many warm and fuzzy memories that motivated me to stay on track, and for allowing me to find my way.

And last but not least I am grateful to beloved Cristian Andrei Gherghina for his love, his stronger than the world belief that we can do this, and his immeasurable contribution to the success of this chapter. This would not have been possible without you.

Thank you all.

List of Publications

- Narcisa Andreea Milea, Lingxiao Jiang, Siau-Cheng Khoo: Scalable detection of missed cross-function refactorings. *In Proceedings of 22nd International Symposium on Software testing and Analysis, pages:138-148, San Jose, US, July 2014.*

- Narcisa Andreea Milea, Lingxiao Jiang, Siau-Cheng Khoo: Vector abstraction and concretization for scalable detection of refactorings. *In Proceedings of 22nd International Symposium on Foundations of Software Engineering, pages:86-97, Hong Kong November 2014.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Programs are meant to be read by humans and only incidentally for computers to execute."

– H. Abelson and G. Sussman

– Structure and Interpretation of Computer Programs

Programming is informally defined as the process that takes in the formulation of a computing problem and transforms it into an executable program. Elements that factor in the process are existing software, methodologies and even hardware. In the early days of programming, the lack of methodologies and attention to code quality, combined with constraints on the size of the executable program, often led to code being only meant to be read by the machine which had "the same clean logical structure as a plate of spaghetti" [16]. The difficulty posed by reusing, *changing*, or reading such code led Dijkstra to argue that GOTO statements should be abolished. Later on, the development of the so-called "waterfall" programming methodology followed. This too did not lack criticism. The "waterfall" basically describes a sequence of phases such as design, implementation, verification, maintenance, whose key problem is that when completed their results cannot change and are frozen. The computing problem must thus

be specified fully before its implementation with no back loop to revise the specification based on changing needs or fresh insights.

Contemporary programming models, in contrast, inherently allow for changes. Software products are released in versions or iterations with each version bringing an increase in functionality. However, in order to add a new feature or port a software system to a new environment, the structure and design of the software needs to be sufficiently flexible to *enable change*.

Refactoring

> *Refactoring is the process of improving the design of existing code while preserving its functionality [38].*

Refactoring is central to the development and maintenance process. In particular, the study of Xing and Stroulia found refactoring to be very common in Eclipse's evolution history, amounting to 70% of structural changes [129]. Refactoring helps to restructure obsolete designs, adapt to new requirements, eliminate redundancy or remove unused functionality. While for communicating with other components the *interface* exposed by software is critical, for the developers that read or need to maintain and evolve software, *looks* certainly matter. Thus *code modularity and clarity are imperative*. The code structure also influences our code comprehension. Software that is easy to read also makes it easy to detect bugs and possible logical errors. A study by Kim et al., [69], on Windows 7 version history, found that refactored modules had higher reduction in the number of post-release defects. The goal of many refactoring transformations is thus to improve code readability and modularity.

Many modern development environments, such as Eclipse, Microsoft Visual Studio, have built-in support for various kinds of *refactoring* operations, such as *Rename Variable, Encapsulate Field, Move Method, Extract Method, and Extract Interface* [40].[1] These tools, however, usually require *refactoring opportunities* (i.e., program fragments

[1] This chapter uses "method" and "function" interchangeably.

that can be, but have not yet been refactored) to be identified first.

1.1 The Gap: Weak Refactoring Support

Detecting *refactoring opportunities* – program fragments that can be, but have not yet been restructured – has been a topic of long lasting interest in the literature [34, 51, 78, 120, 121]. The theme for many of these works is reducing duplication in code or so called *code clones*. Specifically, duplicate code in sibling classes might gain from *generalization refactoring*, that involves creating an abstract superclass for a set of concrete classes, migrating the common functionality to that superclass, and *inheriting* from that class, as defined by Opdyke in his thesis [93]. Therefore, some developers might find *generalization refactoring opportunities* from clone detector results. The Aries system, proposed by Higo et al. [49–51], exploits this idea by classifying duplicated code reported by CCFinder according to the refactoring operations that can remove duplication. Their approach is based on a number of metrics that characterize the coupling of the duplicated program fragments with their surroundings. For example, if duplicated methods appear in classes that extend a common base class, the *Pull Up Method* would be suggested.

However, removing duplication by generalization or inheritance tends to be less of a popular choice compared to employing delegation (Kegel et al. [65]). Hui Liu et al., [79], showed in their evaluation that more than 95% of clone sets were not accepted as generalization refactoring opportunities by developers. Moreover, replacing generalization or inheritance with *delegation* is a standard refactoring operation and a recurrent theme in programming textbooks.

Despite the large body of work and the presence of many proposed engines for finding opportunities and automated refactoring, recent studies show that developers often perform real-world refactoring manually [69, 85, 87, 125]. Specifically, [85] shows

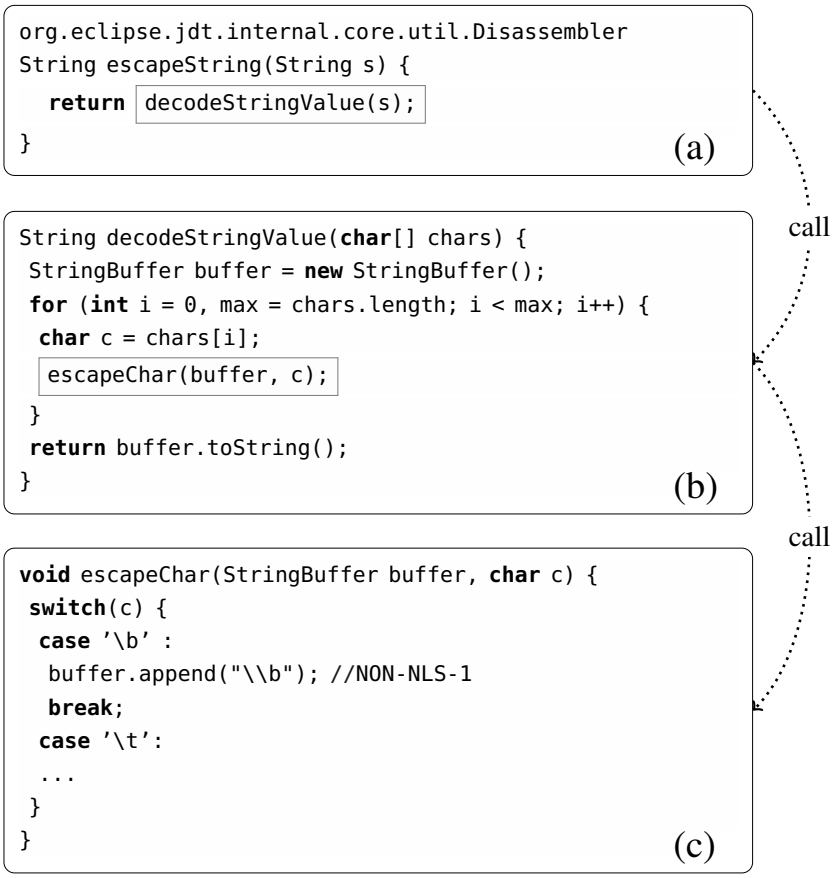
that almost 90% of refactoring is performed manually, without the help of tools. The problem with this is that manual refactoring is error prone. According to a field study at Microsoft [69], more than 70% of developers find it difficult to correctly perform refactoring. Moreover, Weißgerber and Diehl found evidence that a significant fraction of the bugs discovered are caused by incomplete refactoring [127]. However none of the current works provides developers with support for finding refactoring opportunities similar to the one performed manually. In what follows we illustrate the gap to be filled by this thesis.

1.1.1 Sample Scenario

A developer realizes that a method in one of the many subprojects of Eclipse is too big and that part of the functionality of that method is needed by other functions. The method in question deals with escaping a string and the partial functionality deals with escaping a character. She decides to increase code modularity by restructuring the method, which involves extracting that functionality into a new method, that can then be invoked as needed. The result is shown in Figure 1. It depicts three methods from a class named `Disassembler` in the `JDT` project. Method `escapeString` invokes `decodeStringValue` which in turn invokes method `escapeChar`. The change history of Eclipse shows that the method `escapeChar` was created by means of an *Extract Method* Refactoring.

Meanwhile the code in Example 2 is found in class named `Disassembler` in the `Equinox` project. It depicts one method, `escapeString`, that is the same as `escapeString` in Example 1 before refactoring in Eclipse 3.5.2. While both `escapeString` methods have the same functionality, they are structurally different. By simply looking at Example 2 alone it may not be clear whether it has refactoring opportunity. However, by observing how the code in Example 1 is structured, a developer can easily detect a missed cross-function refactoring opportunity for the method

Example 1. *(Extract Method Refactoring from Eclipse)*



`escapeString` in Example 2 as well. Usual clone detection may consider parts of the code in the two examples as code "clones" (i.e., code fragments similar to each other, [62, 71, 103] [36, 59]) under some very relaxed similarity conditions. Detecting the similarity between the two `escapeString` methods on the other hand requires an inter-procedural analysis and answering questions such as 1) what to compare: naively searching through all possible combinations of different code fragments is not likely to be scalable and 2) how to represent code: a representation such an AST or program dependence graph (PDG) will deem an inter-procedural analysis expensive.

The example also indicates that not all refactoring opportunities would be detected and capitalized on by developers at the same time. Refactoring *parts of a large code base*

Example 2. *(Refactoring Opportunity from Eclipse)*

```
1  org.eclipse.equinox.p2.internal.repository.comparator.
       java.Disassembler
2
3  String escapeString(String s) {
4   StringBuffer buffer = new StringBuffer();
5   for (int i = 0, max = s.length(); i < max; i++) {
6    char c = s.charAt(i);
7    switch (c) {
8     case '\b' :
9      buffer.append("\\b"); //NON-NLS-1
10     break;
11    case '\t':
12     buffer.append("\\t"); //NON-NLS-1
13     break;
14    case '\n':
15     buffer.append("\\n"); //NON-NLS-1
16     break;
17    ...
18   }
19  }
20  return buffer.toString();
21 }
```

(a)

of related programs often causes initially similar code fragments in different projects to diverge. For code bases that have long evolution histories, such divergences can in time cause difficulties in finding those missed refactoring opportunities again. Usual refactoring detection based on clone detection (e.g., [12, 36, 50, 103, 116]) would not report two code fragments, one of which is a refactored copy of the other, as clones, and would miss many refactoring opportunities. As for this example, the `switch` statement in Example 1 may be detected as a clone of the body of `escapeChar` in Example 2, but the `for` loops might not be detected, as the for-construct is present in a method `decodeStringValue`, separated from the switch statement. Thus, usual clone detection will typically fail to suggest a refactoring for code in Example 2.

For the remaining of this thesis we will use the term refactoring opportunity or missed refactoring opportunity to denote the following:

Definition 1.1 *(Refactoring Opportunity)*

> *A pair of code fragments (C, C') represents a refactoring opportunity if C can be transformed through refactoring to obtain C'.*

Scalably detecting such missed cross-function refactoring opportunities and helping developers perform refactoring is one of the goals of this thesis. Detecting *missed refactoring opportunities* can help keep code consistent by finding inconsistencies or divergence in the design or structure of code that might be indicative of bugs or *code smells* [38]. Consistency even in the little details, such as when to introduce a new variable or create a new method, matter a lot to developers. How often have you opened up code written by a third party, and before anything else, re-indented it to match your coding style? When everyone is writing code that has the same consistent style, code is easier to understand. One example of refactoring opportunity we can detect that shows code divergence and an inconsistency in naming is illustrated in Example 3.

Example 3. *(Extract Method and Inconsistent Naming)*

```
org.eclipse.jdt.internal.corext.buildpath.ClasspathModifier
public static boolean isIn External Or Archive (IJavaElement element) {
 IPackageFragmentRoot root =
  (IPackageFragmentRoot) element.getAncestor(IJavaElement.PACKAGE_FRAGMENT_ROOT);
 return root != null && (root.isArchive() || root.isExternal());
}
```

```
org.eclipse.jdt.internal.ui.jarpackager.JarPackageWizard
private static boolean isIn Archive Or External (IJavaElement element) {
 IPackageFragmentRoot root =
  JavaModelUtil.getPackageFragmentRoot(element);
 return root != null && (root.isArchive() || root.isExternal());
}
```

call

```
org.eclipse.jdt.internal.corext.util.JavaModelUtil
public static IPackageFragmentRoot getPackageFragmentRoot(IJavaElement element) {
 return
 (IPackageFragmentRoot) element.getAncestor(IJavaElement.PACKAGE_FRAGMENT_ROOT);
}
```

1.1.2 Refactoring Reconstruction

Another theme in refactoring is the detection of *historic refactoring*. Studies that focus on this goal aim to reconstruct the refactoring operations used to transform the code by analyzing different versions of a program to facilitate evolution studies [25, 26, 48, 70, 98, 113, 118, 128].

An approach that provides consistent detection of refactoring opportunities and historic refactoring is needed. Such an approach would enable developers to both improve software maintenance by detecting missed opportunities and measure more accurately the refactoring efforts and progress through software evolution.

This goal presents the following challenges:

1. Scalability of a refactoring detection technique still remains challenging for large code bases containing millions of lines of code.

2. The problem of detecting *refactoring opportunities* is compounded by *cross-function* refactoring opportunities that involve moving code fragments across function boundaries. Naively searching through all possible combinations of different code fragments is not likely to be scalable.

3. It continues to be challenging for many code clone techniques and the refactoring detection techniques based on them to detect the similarity between two code fragments without incurring a significant number of false positives in their outcome [23, 41, 57, 60, 68, 91, 102]. This is a consequence of the fact that these techniques cannot *flexibly omit or emphasize specific program elements* when computing similarity.

Example 1 illustrated the first challenge, posed by cross-function refactoring opportunities. Here, we also illustrate, through an example, the third challenge faced by detecting refactoring. Specifically, the code in Example 4 was detected by our work between two versions of Apache Ant. It shows a category of refactoring operations to

Example 4. *(Apache Ant setEof.v1 Changed to setEof.v2)*

```
1  org.apache.tools.ant.taskdefs.FixCRLF
2
3  public class FixCRLF ... {
4
5  private int ctrlz;     // eof: -1 =>
       remove, 0 => asis, +1 => add
6
7  /* ...
8    * Unix: cr="remove" tab="asis" eof="
       remove"
9    */
10 public FixCRLF() {
11  ...
12  if (System.getProperty("path.separator
       ").equals(":")) {
13   crlz = -1; // remove
14  ...
15 }
16
17 ...
18
19 public void setEof(AddAsisRemove attr)
20 {
21  String option = attr.getValue();
22  if (option.equals("remove"))
23   ctrlz = -1;
24  } else if (option.equals("asis")) {
25   ctrlz = 0;
26  } else {
27   // must be "add"
28   ctrlz = +1;
29  }
30 }
```

```
1  org.apache.tools.ant.taskdefs.FixCRLF
2
3  public class FixCRLF ... {
4
5  private static final int ADD = 1;
6  private static final int ASIS = 0;
7  private static final int REMOVE = -1;
8  ...
9
10 public FixCRLF () {
11     ...
12    if (System.getProperty("path.
        separator").equals(":")) {
13          ctrlz = REMOVE;
14    ...
15 }
16
17
18 public void setEof(AddAsisRemove attr)
19 {
20  String option = attr.getValue();
21  if (option.equals("remove")) {
22   ctrlz = REMOVE;
23  } else if (option.equals("asis")) {
24   ctrlz = ASIS;
25  } else {
26   // must be "add"
27   ctrlz = ADD;
28  }
29 }
```

improve code clarity that involve small code transformations. Checking the correctness of method `setEof` after refactoring, denoted here by `setEof.v2`, is much easier. For method `setEof.v1` a developer must first check that the constant values used are correct. In order to alleviate this problem, the code contains comments that document the constant values used.

```
1 private int ctrlz;     // eof: -1 => remove, 0 => asis, +1 => add
```

This however is error prone. In particular, Ant revision history shows that a bug related

to confusing numerical values in `FixCRLF` was fixed in commit 6231c77 shown below.

```
240 240 log("options:" +
241     " cr=" + (addcr==-1 ?  "add" :  addcr==0 ?  "asis" :  "remove") +
242     " tab=" + (addtab==-1 ?  "add" :  addtab==0 ?  "asis" :  "remove") +
243     " eof=" + (ctrlz==-1 ?  "add" :  ctrlz==0 ?  "asis" :  "remove") +
    241 " cr=" + (addcr==1 ?  "add" :  addcr==0 ?  "asis" :  "remove") +
    242 " tab=" + (addtab==1 ?  "add" :  addtab==0 ?  "asis" :  "remove") +
    243 " eof=" + (ctrlz==1 ?  "add" :  ctrlz==0 ?  "asis" :  "remove") +
244 244 " tablength=" + tablength,
245 245 Project.MSG_VERBOSE);
```

Later on, in commit a794b2b between versions 1.3 and 1.4, the *Replace Number with Constant* refactoring was performed which resulted in method `setEof.v2`. The refactoring operation performed is defined as follows:

Replace Number with Constant

> *A transformation that takes in a code fragment containing a literal number with a particular interpretation and creates a constant, names it after the interpretation, and* replaces *the number with it is called Replace Number with Constant.*

Detecting code fragments that *differ exactly* by the small transformation induced by *Replace Number with Constant* can prove difficult. In particular it may result in a large number of *almost* identical code fragments to be returned by traditional approaches that use a *threshold to find similar code*; all of which are clones but irrelevant to the *Replace Number with Constant* refactoring (aka., high number of false positives.). This indicates that a desired refactoring detection tool should possess knowledge about the precise code change patterns induced by refactoring and should *flexibly omit or emphasize specific program elements* when computing similarity, as well as work flexibly with various kinds of refactoring operations.

The state of the art in detecting historical refactoring, RefFinder, detects the *Replace Number with Constant* refactoring by the definition below. The definition however is very coarse-grained. It checks that the transformation involves a) the creation of a constant, ($added\_field$) and b) an access to the new constant by a method ($added\_accesses \wedge$

$before\_method$). It does not however check that the transformation replaces a number by a constant thus also erroneously identifying as a true positive a transformation that completely rewrites a method and happens to access a new constant. Moreover it will not detect the opportunity posed by `setTab` shown in Example 5 to be refactored in the same way as `setEof`.

RefFinder Definition of Replace Number with Constant

added_field(fFullName, X, X) $\wedge$
added_fieldmodifier(fFullName, final) $\wedge$
added_accesses(fFullName, mFullName) $\wedge$
before_method(mFullName, X, X) $\rightarrow$ replace_magic_number_with_constant(mFullName, fFullName)

Example 5. *(Apache Ant setTab.v1 Refactoring Opportunity)*

```
1  public void setTab(AddAsisRemove attr)
2  {
3   String option = attr.getValue();
4   if (option.equals("remove")) {
5    addtab = -1;
6   } else if (option.equals("asis")) {
7    addtab = 0;
8   } else {
9    // must be "add"
10   addtab = +1;
11  }
12 }
```

```
1  public void setEof(AddAsisRemove attr)
2  {
3   String option = attr.getValue();
4   if (option.equals("remove")) {
5    ctrlz = REMOVE;
6   } else if (option.equals("asis")) {
7    ctrlz = ASIS;
8   } else {
9    // must be "add"
10   ctrlz = ADD;
11  }
12 }
```

1.2 Thesis Statement and Contributions:

Refactoring Support via Vector Operations

Thesis

> *Efficient and effective detection of code fragments related by refactoring operations within large software can be achieved by performing abstraction and algebraic operations over high-dimensional vectors representing the code.*

In this work, we present a new vector-based approach for scalable detection of both refactoring opportunities and historical refactorings. The vectors we construct, as the first step of our approach, were first proposed by [41]. These *characteristic vectors* encode syntactic program features and each of them can be considered an additive program representation. This property of the representation is essential to our work and can be defined as follows:

Additive representation

> *Given two non-overlapping code fragments c_1 and c_2 from method m and their vectors v_1 and v_2, $v_1 + v_2$ exactly represents code fragment $c_1; c_2$.*

We use such vectors to encode *inlined* code so that the effect of method extraction and inlining, which are common transformations induced by various refactoring operations, can be captured. 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 invocations, return operations, and formal and actual parameter substitutions.

Then, we employ a novel approach via vector *abstraction* and *concretization* for manipulating the characteristic vectors flexibly based on code change patterns induced by known refactorings. For Example 4, both the *abstraction* and *concretization* operations manipulate the features in the characteristic vectors related to “number literals” and “simple variable names holding constants”. The abstraction will “massage” them into a single feature “number literals or simple names holding constants” thus making the vectors equal. The concretization will reverse the effect of the abstraction and constrain the code change patterns between two vectors to be just those induced by the sought after refactoring *Replace Number with Constant*. Specifically the concretization will check whether the difference between the features "number literals" is the reverse of that between the features "simple names". Since vector-based operations such as additions

can be performed in almost linear time with respect to the total number of vectors and the dimension of each vector, it becomes the key to the scalability of our approach.

In addition to detecting a code fragment c that may be refactored, our approach also reports the code fragment r, that either shows how c was refactored, when detecting historical refactoring, or provides an example of how c may be refactored, when detecting opportunities for refactoring. For Example 2, our approach identifies `escapeString` in the Equinox project as a refactoring opportunity, and it reports the code fragments shown in Example 1 to illustrate how it may be refactored. Then, a user could proceed to refactor the code in Example 2 in a way similar to Example 1. For Example 4, our approach identifies method `setEof`.v1 on the left as an actual refactoring opportunity that was exploited resulting in `setEof`.v2 on the right.

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.

Let us briefly outline the key points in which our proposal differs from previous approaches. Some previous studies focus on the detection of refactoring operations that have happened and recorded in the version history of a project (e.g., [25, 70, 113, 118, 128]), so as to reconstruct those refactoring operations. Some other studies focus on formal definitions of refactoring operations (e.g., [107, 108, 119]), so as to help ensure semantic equivalence or correctness of code refactoring. Some tools, such as LAMBDAFICATOR and CONCURRENCER, can automatically perform certain refactoring

operations (e.g., converting sequential code to use `java.util.concurrent` support, changing anonymous class to lambda expression, replacing certain `for` loops with functional operations, etc.). Other tools only perform a refactoring operation if the code that needs the operation is identified first with sufficient relevant information (e.g., [40, 50]). Our study addresses a different problem of scalable and precise identification of both refactoring opportunities and historical refactoring; results from our tool can be used to facilitate other tools for performing and validating refactoring operations. Similar to our study, Cider [112], a recent study on detecting refactored clones, can also detect code clones that have diverged due to refactoring. However, Cider's detection algorithm works on a graph representation of a program, which can be computationally expensive and has limited ability in detecting cross-function refactoring. Also, Cider requires initial seeds for its search algorithm, while our approach does not need seeds. Another study by Meng et al. [83] can also detect refactoring opportunities. They create context-aware edit scripts from two or more examples and use the scripts to identify edit locations and transform the code. However, edit-scripts are also limited within a single method, and are yet not scalable to detect changes across methods.

Our main contributions in this thesis are as follows:

- We propose a new technique called *vector inlining* to simulate the effect of method inlining, which enables scalable detection of cross-function refactoring;
- 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 abstract 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.

1.3 Overview and Organization

This dissertation is principally positioned in the domain of automated refactoring detection. Hence we start with a classification and discussion of the different refactoring operations in Chapter 2. Chapter 3 then surveys the related works on the detection of refactoring opportunities and the reconstruction of historical refactoring. We also discuss works in the field of clone detection due to the similarity of the techniques we propose in this thesis to works in the field of clone detection and the contribution of this thesis on pushing the state of the art in finding cross-function similarity or "clones".

In Chapter 4 we introduce vector inlining, which enables the scalable detection of cross-function refactoring opportunities. In Chapter 5 we introduce vector abstraction and concretization to detect a large number of both refactoring opportunities and historical refactoring. We conclude this dissertation with a discussion on future directions of research in Chapter 6 and with a summary of the contributions in Chapter 7.

Chapter 2

Preliminaries

Refactoring techniques induce a spectrum of code transformations, ranging from renaming a method, to more complex and ambitious transformations such as extracting a class, changing to an object-oriented design from a procedural one, or restructuring a whole class hierarchy by introducing polymorphism. The refactoring.com website maintains a catalog of refactoring techniques [37] and describes around 100 refactoring techniques.

This chapter presents some refactoring techniques and a classification of the different *classes of refactoring operations*, that are the subject of existing works, according to their effect on code structure. Existing works are concerned with either the implementation of refactoring or support in detecting software refactoring. For instance, modern software development environments, e.g. Eclipse and MS Visual Studio, offer interactive support for implementing a number of refactoring techniques [2]: *Rename Method/Field/Class, Extract Method, Inline Method, Pull-Up/Push-Down Method, Move Method, Self-Encapsulate Field, Add Parameter, and Extract Interface*. By interactive support we refer to the fact that the developer needs to first select a refactoring operation and the code location that needs refactoring, and the tool will implement the transformation, prompting the developer to provide any required information. Another

[2] In this thesis we use the terms refactoring techniques and refactoring operations interchangeably.

goal of refactoring works is to automatically find code locations that need refactoring and thus speed up the process of refactoring and improve the productivity of software development. Detecting *refactoring opportunities* is a highly active area of research. Current investigations in this area can be broadly classified according to the type of approaches they employ and the *refactoring operations they detect*.

2.1 Refactoring Operation Types

One class of refactoring operations, that many of the existing works focus on, is meant to optimize the class structure of object-oriented systems. Such an optimization can, for example, involve *moving features across classes*. Detecting refactoring opportunities from this class requires an analysis of relations between classes, methods, and even class attributes. *Extract Class* and *Move Method*, defined below, fall into this category.

Definition *(Extract Class)*

Given P, a program in which the functionality of one class can be divided across two classes, an Extract Class refactoring is a code transformation that creates a new class and moves the relevant fields and methods into the new class, thus yielding a new program P', $P \overset{Extract\ Class}{\rightarrowtail} P'$.

Definition *(Move Method)*

Given P, a program in which a method is using or is used by more features of another class than the class in which it is defined, a Move Method refactoring is a code transformation that creates a new method with a similar body in the class it uses most and transforms the old method into a simple delegation or even completely removes it, thus yielding a new program P', $P \overset{Move\ Method}{\rightarrowtail} P'$.

Generalization Refactoring, also known as *Refactoring for Generalization* (Opdyke [93], Fowler et al. [38]), represent another class of refactoring operations. *Generalization*

refactoring aim to restructure object-oriented code to fully benefit from generalization. They basically involve creating an abstract superclass for a set of concrete classes, and migrating the common functionality to that superclass, as defined by Opdyke in his thesis [93]. Examples of refactorings in this class are: *Pull Up Field*, *Pull Up Method, Pull Up Constructor Body, Push Down Method, Push Down Field, Replace Inheritance with Delegation*, *Extract Interface*, etc.

Definition *(Pull Up Field)*

Given P, a program in which two subclasses have the same field f, a Pull Up Field refactoring is a code transformation that moves f to the superclass, thus yielding a new program P', $P \overset{Pull\ Up\ Field}{\rightarrowtail} P'$.

Definition *(Extract Interface)*

Given P, a program in which several clients use the same subset of a class's interface or in which two classes share part of their interfaces, an Extract Interface refactoring is a code transformation, that extracts the subset into a new interface, thus yielding a new program P', $P \overset{Extract\ Interface}{\rightarrowtail} P'$.

Another class of refactoring operations deals with composing methods in order to correctly bundle code. Methods that are too long are very often a source of problems. The key refactoring in this category is *Extract Method*. In order to detect refactoring operations that fall in this class an inter-procedural analysis is needed.

Definition *(Extract Method)*

Given P, a program in which a method contains a code fragment c which is (fairly) loosely coupled with its context, an Extract Method refactoring is a code transformation that extracts fragment c into a new method m and replaces the fragment with a call to m, thus yielding a new program P', $P \overset{Extract\ Method}{\rightarrowtail} P'$.

Another class of refactoring operations deals with *Organizing Data, Making Method Calls Simpler, and Simplifying Conditional Expressions*. Out of this class of operations, a number of refactorings induce only simple code transformations: transformations to method signatures for instance (renaming a method) or simple transformations to how data is organized in classes, *Replace Data Value with Object* shown in Example 6. Though these simple transformations can help improve the readability of code, our focus in this thesis is more on modularity of code and the subclass of refactoring operations that induce *structural changes on method bodies* (e.g. extracting a method, introducing a new variable). One instance of this subclass of operations is *Consolidate Conditional Expression*. This refactoring can be described as changing a sequence a conditionals into one conditional and extracting the new conditional into a new method. As opposed to detecting this instance of structural change the detection of *Replace Data Value with Object or Pull Up Field* requires an analysis and decision procedure that takes into account more than method structural information. Consequently operations such as *Replace Data Value with Object, Rename Method, or Pull Up Field* are out of the scope of this thesis.

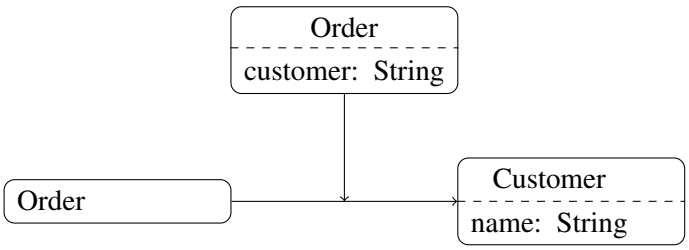
Definition *(Rename Method)*

Given P, a program in which the name of a method does not reveal its purpose, a Rename Method refactoring is a code transformation that changes the name of the method, m, thus yielding a new program P', $P \overset{Rename\ Method}{\rightarrowtail} P'$.

Definition *(Replace Data Value with Object)*

Given P, a program which contains a data item that needs additional data or behavior, a Replace Data Value with Object refactoring is a code transformation that transforms the data item into an object, thus yielding a new program P', $P \overset{Replace\ Data\ with\ Object}{\rightarrowtail} P'$.

Example 6. *(Replace Data Value with Object)*



Definition *(Consolidate Conditional Expression)*

Given P, a program in which a method contains a sequence of conditional tests with the same result, a Consolidate Conditional Expression refactoring is a code transformation that combines them into a single conditional expression c, extracted/refactored into a new method m, thus yielding a new program P', $P \overset{Consolidate\ Conditional}{\rightarrowtail} P'$.

Existing state-of-the-art techniques with respect to refactoring opportunity detection handle only a small number of refactoring operations. Most of the existing techniques are specifically tailored to one operation and cannot be applied to other operations. According to [20], *Extract Class, Move Method and Extract Method are targeted by most of the approaches*.

In contrast, this thesis proposes a general approach for detecting a significant number of *method level* refactoring opportunities. By method level refactoring we refer to those refactorings that induce structural transformations reflected at the method level and only optionally at the level of the class hierarchy or at how methods are bundled in classes. One such example is the *Extract Method* refactoring operation, as shown in Example 1. The transformation at the method level induced by *Extract Method* involves replacing a code fragment c with a call to a method that has the same functionality as c. Another

example would be *Replace Method with Method Object*, defined below and illustrated in Example 7. This refactoring operation involves transforming the local variables in a method into fields.

Definition *(Replace Method with Method Object)*

> *Given P, a program which contains a long method that uses local variables in such a way that you cannot apply Extract Method, an Replace Method with Method Object refactoring is a code transformation that turns the method into its own class so that all the local variables become fields of that class, thus yielding a new program P'.*

Example 7. *(Replace Method with Method Object)*

```
1 class Order{
2  int price (int i, int q) {
3   double primaryBasePrice;
4   double secondaryBasePrice;
      // long computation;
5     primaryBasePrice = (i * q) + i%q;
      secondaryBasePrice = (i * q%i);
6  }
```

```
1 class Order{
2  int price (int i, int q) {
3   return new OrderComputation(this).
       price(i, q);
4  }
5 }...
6 class OrderComputation{
7  double primaryBasePrice;
8  double secondaryBasePrice;
9  int price (int i, int q) {
     // long computation;
10   primaryBasePrice = (i * q) + i%q;
     secondaryBasePrice = (i * q%i);
11 }
```

In this thesis we efficiently and accurately define structural transformations reflected at method level by abstraction and algebraic operations over high-dimensional vectors representing the code. For instance *vector inlining*, which will be presented in Chapter 4, makes it possible to scalably detect refactoring operations that involve transformations that span across method boundaries such as those from the class that deals with composing methods (e.g., *Extract Method*). Furthermore, in Chapter 5 we will show that the *precise code change patterns* induced by refactoring, from classes such as those that deal

with simplifying conditional expressions or making method calls simpler, can be learned from examples, using a semi-automatic approach, and then detected by employing a flexible approach based on *abstractions* and *concretizations*.

Moreover, though this thesis is concerned with refactoring (our experiments in Chapter 5 target 21 refactoring operations), the techniques it describes are general and can in theory be applied to detect code transformations from a more broader scope such as bug fixes or other systematic code edits.

Chapter 3

Related Work

The purpose of this chapter is to provide a survey on the state-of-the-art research in refactoring detection. In Section 3.1 we survey the related work on detecting refactoring opportunities. Section 3.2 presents the related work on refactoring reconstruction, which is critical for code evolution and API understanding. In Section 3.3, we present an overview of the studies which define refactoring operations and automate the code transformations induced by refactoring. As refactoring detection is related to clone detection, Section 3.4 investigates the use of clone discovery in detecting refactoring opportunities. Furthermore, Section 3.5 reviews an emerging area of clone detection research aimed at finding code fragments matching an input code fragment in fractions of a second. Finally, Section 3.6 surveys works that automate program transformations based on examples.

3.1 Refactoring Opportunities

Refactoring opportunities are informally defined as code fragments that could benefit from refactoring. Over the last decade a number of works [49–51, 121, 124] have been proposed to automate the detection of such code fragments. Based on their underlying

techniques, these works can be classified as illustrated in Figure 3.1. Some of these works suggest refactoring based on checking if a set of conditions is satisfied by the code. Others look at software metrics such as the structural and semantic similarity between methods in a class, or propose techniques that involve clustering code fragments, representing code as a graph, or code slicing. In terms of refactoring operations to which they can be applied, most of the works are specifically tailored to one specific refactoring operation and can be hard to generalize to other operations. In contrast, our approach is more general and can be used to detect a broad range of refactoring opportunities. Moreover, our proposal can be used to complement most of the existing techniques, especially those that target opportunities such as *Extract Class and Move Method* whose aim it to optimize class hierarchies and the bundling of methods in classes. In the following subsections we review these existing techniques.

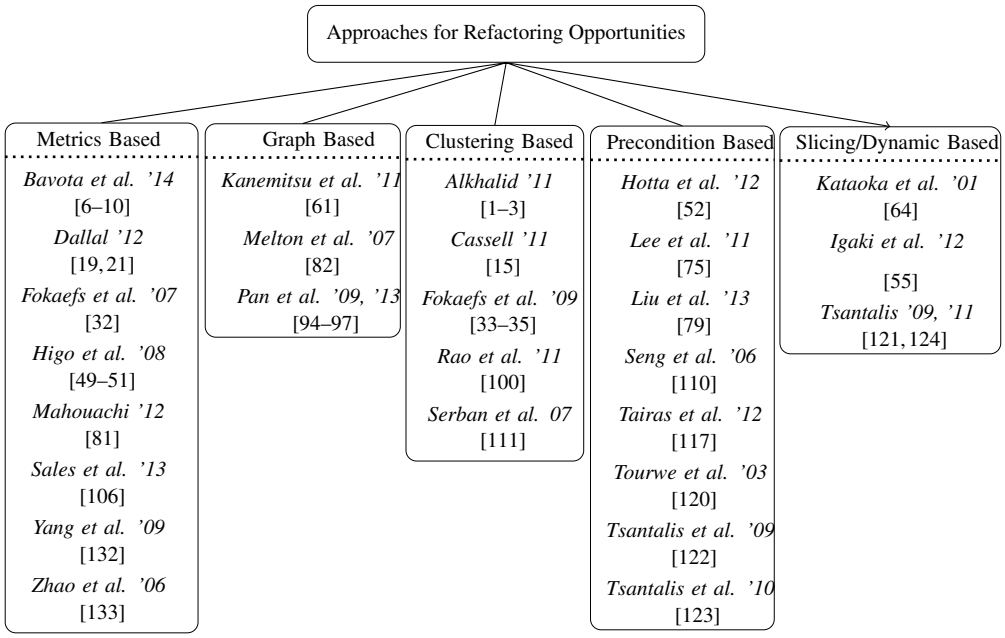
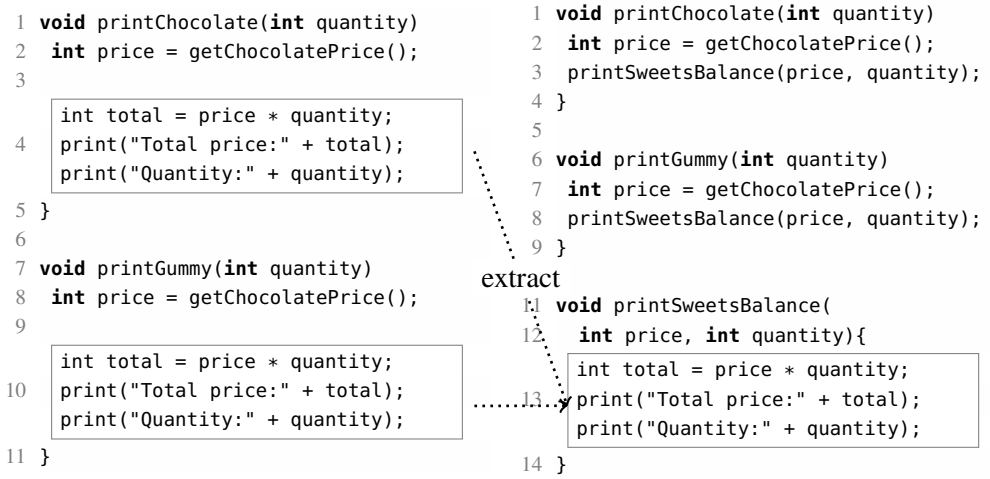


Figure 3.1: Approaches for Detecting Refactoring Opportunities

3.1.1 Metrics-Based Approaches

The Aries system, proposed by Higo et al. [49–51], is based on a number of *metrics* that characterize the coupling of duplicated program fragments with their surroundings. Based on these metrics, Aries classifies duplicated code reported by CCFinder according to the refactoring operations that can remove duplication. For example, if duplicated methods appear in classes that extend a common base class, the *Pull Up Method* would be suggested. Other refactorings it can detect include: *Extract Class, Extract Superclass, Extract Method, Pull Up Method, Form Template Method, Move Method, Parameterize Method, Pull Up Constructor*. The perfect type of duplicate code the approach of by Higo et al. targets, also called *merge clones* or *generalization clones*, is shown in Example 8. For this example Aries would suggest an Extract Method refactoring.

Example 8. *(Perfect Sample Code for Merge Clone Refactoring)*



However not all duplicate code can benefit from merge or generalization refactorings. Hui Liu et al., [79], showed in their evaluation that more than 95% of clone sets were not accepted as generalization refactoring opportunities by developers. Generalization refactorings basically involve creating an abstract superclass for a set of concrete classes, migrating the common functionality to that superclass, and inheriting from that class.

An example of duplicate code that should not be merged is shown in Example 9. Each of the methods in Example 9 sets the value of a class field based on checking if a set of conditions is abided by. As these methods access different fields they cannot benefit from merge or generalization refactorings. In general, removing duplication by means of generalization, or inheritance, tends to be less of a popular choice compared to employing delegation (Kegel et al. [65]). Consequently, most clone sets might not be good generalization refactoring opportunities.

Example 9. *(A False Positive Case for Merge Clone Refactoring)*

```
1  public void setTab(AddAsisRemove attr)
2  {
3   String option = attr.getValue();
4   if (option.equals("remove")) {
5    tabs = SPACES;
6   } else if (option.equals("asis")) {
7    tabs = ASIS;
8   } else {
9    // must be "add"
10   tabs = TABS;
11  }
12 }
```

```
1  public void setEof(AddAsisRemove attr)
2  {
3   String option = attr.getValue();
4   if (option.equals("remove")) {
5    ctrlz = REMOVE;
6   } else if (option.equals("asis")) {
7    ctrlz = ASIS;
8   } else {
9    // must be "add"
10   ctrlz = ADD;
11  }
12 }
```

A number of other proposals aim to predict *Extract Class* refactoring opportunities. In [7] Bavota et al. approach the problem by analyzing the structural and semantic similarity between methods in a class, *class cohesion*. In [11] they combine class cohesion with *class coupling* (similarity between one method and methods in other classes) and propose an approach based on game theory. Zhao et al. [133] take on a different approach and propose the use of a set of complexity metrics to predict *Extract Class*. The work of Dallal, [19, 21], predicts *Extract Subclass* using a statistical model based on size, cohesion, and coupling metrics. Variants of class cohesion and coupling are also used to predict *Move Method* opportunities [10], [32], [106].

The work proposed by Mahouachi et al. [81] is claimed to be applicable to all refactoring activities. Their approach is based on a genetic algorithm, that considers knowledge from a set of defect examples and their corrections, in order to generate a set

of rules. The format of the rules and an example of a rule are as follows:

IF “Combination of metrics with their threshold values” THEN “Combination of Refactorings to apply”

IF No. of attributes >= 10 AND No. of methods >= 20 THEN MoveMethod >=6 AND EncapsulateField < 18

As can be seen from the example, the metrics used in [81] refer to properties of classes. They do not capture method functionality, do not aim to *improve code within methods*, and essentially classify classes as bad smells[3]. However detecting classes that can refactored can be a bit coarse grained. This thesis does not aim to supersede techniques that focus on class-level optimizations such as those detected by [81] or [7, 10, 11, 19, 21, 32, 106, 133]. Instead, our proposal aims to accurately and efficiently capture and specify fine-grained structural information and code transformations induced by refactoring on methods thus *improving code within methods*. The resulting *vector abstraction and concretization* technique, to be presented in Chapter 5, is generic enough that it can be used to capture a broad range of refactorings.

3.1.2 Graph-Based Approaches

The majority of proposals that use graph based program representations to find opportunities for refactoring are mostly targeted towards optimizing the class structure of a program. In [96, 97], Pan et al. detect *Move Method* refactoring opportunities. Their approach is based on a graph that encodes the relations between class attributes and methods. This graph is processed in order to compute an optimized class structure by an evolutionary algorithm [96] or a community detection algorithm [97]. The *Move Method* refactoring opportunities are then identified by comparing the initial and optimized graphs. A similar approach is presented in [94, 95], where the graphs are created to

[3]A code smell is defined according to Fowler as a “surface indication that usually corresponds to a deeper problem in the system". Code smells have also been defined as being certain structures in the design that indicate violations of fundamental design principles and negatively impact design quality [115].

encode classes and their dependencies and the work focuses on *Move Class*.

The work of Melton et al., [82], focuses on *Extract Interface* refactoring opportunities. They construct a graph where each node is a class and the edges encode the dependencies between classes. An analysis is then employed on the reachability information of each node to detect *Extract Interface* refactoring opportunities.

None of the previously mentioned proposals focuses on *improving code within methods*. This is however the aim of the approach by Kanemitsu et al. in [61]. Their approach constructs the program dependencies graph of a method and analyzes the distances between the nodes of this graph in order to identify *Extract Method* opportunities.

3.1.3 Clustering-Based Approaches

The idea behind clustering based techniques is to use the semantic and structural distances between various software elements to improve the grouping of code, e.g. lines of code in methods, methods in classes, etc. In [111], Serban et al. propose computing the distances between the fields and methods both within a class and across classes to identify *Move Method, Move Field, Inline Class, and Extract Class* opportunities. Similarly, Alkhalid et al. proposed using the similarity between methods within and across classes to predict *Move Method* opportunities. They also proposed using the similarities between classes to detect *Move Class* opportunities and thus refactor the bundling of classes into packages [3].

Clustering is also used to predict if the functionality of one class can be divided across two classes, or basically if an *Extract Class* can be performed, [15, 33–35, 100]. Specifically, Fokaefs et al. and Rao et al. [33–35, 100] consider the distances between class attributes and methods as the basis for a number of clustering techniques algorithms. Similarly, Cassell et al. [15] consider the semantic and structural distances between the elements of a class as the basis for clustering.

Extract Method can also be detected using clustering. In [1], Alkhalid et al. propose

using the similarity between the lines of code in a method as input to an adaptive k-nearest neighbor algorithm in order to predict *refactoring at function level*.

3.1.4 Precondition-Based Approaches

The idea behind precondition-based approaches is to define a set of conditions which, when satisfied by code, indicate the applicability of a specific refactoring operation. Seng et al. [110] propose preconditions to identify *Move Method* refactoring. In [52, 75, 79, 117] the authors propose a set of preconditions for code clones, priorly detected, in order to identify the best refactoring operation to apply. There are also studies that detect refactoring opportunities based on "bad smells" in code [38, 78, 86, 120, 123]. Tourwe and Mens [120] use logic programming to encode several types of refactoring operations and detect possible refactoring opportunities. They define a set of conditions which, if satisfied by priorly detected bad smells, indicate the applicability of refactoring. Tsantalis et al. [123] define a set of conditions to identify refactoring opportunities introducing polymorphism. In [122] they identify *Move Method* refactoring opportunities using an approach based on detecting the *Feature Envy* bad smell and checking the compliance to some preconditions.

Our proposal is complementary to existing approaches in this category. They could be used by developers as a first step to detect code fragments that are suitable candidates for refactoring. Our proposal can then make use of the result of this first step to detect missed refactoring opportunities or to reconstruct the refactorings that have occurred.

3.1.5 Code Slicing and Dynamic Analysis Approaches

Code slicing and dynamic analysis have also been touted as approaches to detecting refactoring opportunities. Tsantalis et al. [121, 124] identify *Extract Method* refactoring opportunities based on slicing the program dependencies graph of a method. Igaki et al. [55] identify *Move Method* refactoring opportunities by analyzing method invocation traces and the relations between methods. Kataoka et al. [64] detect refactoring oppor-

tunities by using program invariants found by Daikon. A specific pattern of invariants indicates the applicability of a specific refactoring operation. For example, the *Remove Parameter* refactoring can be applied if an invariant over the parameters indicates that one parameter can be obtained from the others and is thus not needed. Compared to these works our approach can be applied to detect a broader range of refactorings.

3.2 Historical Refactoring

When a new version of a program, P', is analyzed or tested, we may want to *reconstruct* the transformation from the previous version, $P \rightarrowtail P'$, into constituent transformations that could be analyzed or tested. The transformations that preserve behavior and represent refactoring could have been omitted from testing or could be analyzed for reasons such as understanding API evolution. Many of the existing techniques targeting such transformations or so-called *historical refactoring* rely on the changes recorded in version control systems. Given successive program versions, P and P', these techniques will essentially analyze the changes found between the corresponding program entities:

$$Approach : Program \times Program \rightarrow \{Diff\}$$

Weißgerber and Diehl [128] search for the changed, added, or removed entities (methods, classes, fields) to get refactoring candidates, and then use clone detection to rank these candidates. They use an analysis based on method signatures, as shown below, to detect *local refactorings* that transform the signature of a method, such as *Rename Method, Hide/Unhide Method, Add/Remove Parameter* or *structural refactorings* such as *Move Class/Interface/Field/Method, and Rename Class*.

Example. Sample Condition for Signature-based Refactoring Detection

Given two program versions, P and P', and a tuple, $(c', m', p', r', w') \in P'$, consisting of a class, a method name, a list of parameter types, a return type, and a visibility level, method m' is obtained from its previous version by refactoring Add Parameter, if the following condition holds:

$$\nexists\,(c', m', p', r', *) \in P \;\wedge\; \exists\,(c', m', p, r', *) \in P \;\wedge\; p \subset p'$$

Hayashi et al. [48] construct a graph representation of the *structural differences* between two program versions and model the refactoring detection as a search on this graph. Demeyer et al. [25] propose the use of a set of heuristic metrics to search for refactoring. Taneja et al. [118] and Dig et al. [26] present tools (`RefacLib` and `RefactoringCrawler` respectively) to detect refactoring between versions of libraries. Soetens et al. [113] detect refactoring operations as actual changes are happening in an integrated development environment. Godfrey et al. [45] analyze how call relations are changed between two versions of a program in order to detect certain refactoring operations.

The state-of-the-art techniques for historical refactoring detection are introduced by Prete et al. [98] and Kim et al. [70]. The techniques observe that a logic query engine can be efficiently used to detect a broad range of refactorings. They essentially represent the changes between program versions as logical facts and define a set of template logic queries, based on these facts, to represent refactoring operations. In Example 10, the logical facts would capture that in program version P' method $compute$ was changed, method $print$ was added, a call to $print$ was added in method $compute$, and that $print$ is similar to the old version of $compute$. Based on these facts the query for *Extract Method*, shown below, would return method $print$ as a refactoring candidate.

As opposed to the techniques in this category our approach is not limited to changes between versions; it can search the entire code base and detect refactoring opportunities

Example 10. *($P \twoheadrightarrow P'$)*

```
1 void compute(int a, int b)
2  int c = a + b;
3  print("Result:" + c);
4  log("Info: Result:" + c);
5 }
```

```
1 void compute(int a, int b){
2  int c = a + b;
3  print(c);   Diff 2: Added Function Call
4 }
5
6 Diff 1: Added Function
7 void print(int c){
8  print("Result:" + c);
9  log("Info: Result:" + c);
10 }
```

RefFinder Definition of Extract Method

*added*_method(newmFullName, newmShortName, tFullName) $\wedge$
*added*_calls(mFullName, newmFullName) $\wedge$
similarbody(newmFullName, newmBody, mFullname, mBody) $\wedge$
after_method(mFullName, X, tFullName) $\rightarrow$
extract_method(mFullName, newmFullName, newmBody, tFullName)

within the same version of a code base as well. The precise change patterns we define differ from those employed by works that detect historical refactoring. Many of these works rely on the changes recorded in version control systems and define the patterns used to detect refactoring based on changes between the versions (e.g. introduced new local variable in function m). These patterns cannot however be applied to detect refactoring opportunities within the same version. Moreover, these works rely on coarse measures of similarity between function bodies. Though powerful in terms of the number of refactoring operations it can detect, the state-of-the-art technique in detecting historical refactoring ([98]) uses a threshold-based measure to capture the similarity between function bodies (*similarbody*). Using threshold-based similarity implies that changes permissible within the threshold can be arbitrary and not necessarily only those

induced by the sought after refactoring operation thus leading to inaccurate results. Instead our approach can *flexibly omit or emphasize specific program elements* when computing similarity. Another example of work that suffers from the same drawback is that of Xing and Stroulia [129]. Their work detects refactoring by comparing program versions at the design level, which are packages, classes, interfaces, fields, and blocks. The comparison is however based on names and structural similarities and does not analyze method bodies.

The rules we define to detect refactoring are based on the *differences* in functions and not the *changes* between versions (e.g. *added_method*). In particular, for *Extract Method* the difference lies in having a method call, that has the same functionality as a code fragment c, instead of code fragment c. We efficiently and accurately define and detect these *differences* in functions by two new techniques called *vector inlining* and *vector abstraction and concretization*. Our approach can thus detect the Extract Method in Example 10 by comparing method `compute` from program version P against method `compute` from P' *after inlining* method `print`.

3.3 Specification and Implementation of Refactoring

Another area of works related to this thesis deals with formally specifying the preconditions and code transformations required by refactoring operations.

The classical work by Opdyke [93] describes a set of refactoring operations for C++ in terms of the preconditions needed to preserve behaviour. Griswold specifies refactoring from the perspective of its effect on the program dependencies graph [46]. Lämmel [74] and Garrido [43] represent refactoring operations through the use of rewriting rules. Recent studies also aim to allow programmers to script their own refactoring operations. To this end, Verbaere et al. [126] propose a domain specific language for expressing dataflow properties on a graph representation of the program.

Schafer et al. [107] improve on this and provide high-level specifications for many refactoring operations implemented in Eclipse. Our work complements these studies, in that it searches for refactoring opportunities that could then be implemented by the various approaches mentioned. In future we plan to investigate the development of a query language and of abstractions that would allow us to more comprehensively and precisely specify the refactoring opportunities to search for.

Some studies focus on automatic refactoring. One of the first tools to offer refactoring support was the RefactoringBrowser for Smalltalk, proposed by Don Roberts et al. [101]. Its success was followed by modern development environments, such as Eclipse, IntelliJ's IDEA and Microsoft's Visual Studio, that offer refactoring capabilities. Another tool was proposed by Franklin et al. [39] to automatically refactor certain anonymous inner Java classes and `for` loops by using lambda expressions and functional operations available in Java 8. Our tool currently focuses on scalable detection only. We plan in future work to make our tool implement the detected refactoring automatically.

3.4 Clone Detection

Clone detection is informally defined as the process of identifying similar code. Developers are often interested in finding and tracking similar code (a.k.a. code clones), since clones may help perform various important tasks during development and maintenance, such as understanding code design and API usage [5, 92], investigating software evolution [4, 27, 71, 91, 105, 137], identifying redundant code for refactoring [12, 50, 136], analyzing possibly incorrect reuses of code with incompatible licenses [30, 63], and propagating bug fixes from one location to other locations with similar bugs [42, 58, 59, 76].

Over the last decade, many techniques and tools have been developed to detect clones, such as Deckard [57], Exas [89], CP-Miner [76], CCFinder [60, 80], CloneDR [12], NiCad [17, 102], XIAO [22, 23], etc. Empirical evaluations of these

tools have shown that they can detect many clones with good accuracy and can scale to millions of lines of code. The clones they detect are however mostly *syntactic* clones that reside in the same function, with limited gaps allowed in-between the code elements in each clone.

In order to detect semantic clones, a number of other studies used program dependence relations among code elements [41, 72, 73, 77]. Compared with syntactic clone detection tools, they are more resilient to certain kinds of changes such as reordered statements, relevant statements interleaved with other irrelevant statements. However, program dependencies considered in these techniques are mostly *limited within functions* and are not aimed at detecting *clones across functions mainly due to scalability concerns*. Cider [112] proposes a solution to the scalability concerns by first employing a simple clone detector, to find similar code fragments or so-called initial seeds, and then by using the seeds to analyze an inter-procedural graph of the program.

Example 11. *(General Clone)*

```
} else if (arg.equals("--file")){
 try{
  configuration.add(args[i+1]);
  i++;
 } catch (ArrayIndexOutOfBoundsException e){
  String message = "File must be specified" +
     "when --file is given";
 }
} else if (arg.startsWith("--consoleLog")
   ||  args[i+1].startsWith("--consoleLog")){
 log("Console Logging");
 ...
}
```

```
} else if (arg.startsWith("--threshold")){
 try{
  properties.add(args[i+1]);
  i++;
 } catch (ArrayIndexOutOfBoundsException e){
  String message = "Threshold must be specified "
     "in order to compute LSH" +
     "when --threshold is specified";
 }
} else if (arg.equals("--nothreshold")){
 // Compute based on default values.
 ...
}
```

merge merge

```
try{
 var1.add(args[i+1]);
 i++;
} catch (ArrayIndexOutOfBoundsException e){
 String message = "var2 must be specified "
    "var3" +
    "when var2 is specified";
}
```

Leveraging on code clone detection techniques there have been proposals to rank program refactoring opportunities (*Extract* or *Pull-Up Method* in particular). Fontana et al. [36] manually refactor code clones detected by three different clone detection tools and find that certain code quality metrics have improved after the refactoring. Higo et al. [50] define several metrics for code clones and demonstrate a tool that can suggest refactoring operations for code clones. Tairas [116] visualize clones so that it may become easy to select candidates for refactoring. Van Rysselberghe and Demeyer [103] investigate three different kinds of clone detection techniques (simple line matching, parameterized matching, and metric fingerprints) and determine that clones detected by different techniques may be suitable for different kinds of refactoring.

Code clones are however, not always suitable for refactoring. Example 11 illustrates such a case. The highlighted code fragments are general code clones, which can be detected using a token-based technique. In the code fragment on the left, operations based on the `-file` property are performed whereas in the code fragment on the right the operations are based on the `-threshold` property. The try-catch blocks have a common logic, however there are a few statements and method calls that differ which obstructs refactoring. In other words, extracting the try-catch blocks using an *Extract Method* would not be possible.

Our study differs from the above studies on clone detection refactoring. Our unique contribution to the code clone detection area of research is a *novel similarity measure that leverages on abstraction and concretization mechanisms and can flexibly omit or emphasize specific program elements*. We also propose a scalable approach that makes it practical to detect cross-function similarity on subject programs containing millions of lines of code. This is achieved by encoding syntactic features of code in vector format and by flexibly omitting or manipulating specific elements of the vectors when measuring similarity. This allows us to constrain the dissimilarities allowed between code thus enabling us to encode precise change patterns induced by refactoring.

Our vector inlining technique, presented in Chapter 4, builds on, and extends a previous study on clone detection with the capability to *handle cross-function similar code*. In contrast to Cider, our technique does not rely on initial seeds and uses vector matching, making it much more scalable to large code bases where code divergence across functions occurs more often.

3.5 Real-Time Clone Search

A special category of related works deals with real-time clone search. Real-time clone search is an emerging area of clone detection research aimed at finding code fragments matching an input code fragment in fractions of a second. Existing works in this category are able to scale to systems of millions of lines of code.

The first work to provide a solution to the requirements of real-time clone search, scalability and a short response time, was the work of Hummel et al. entitled "Index-Based Code Clone Detection: Incremental, Distributed, Scalable" [53]. The key technique proposed by the authors involves hashing code fragments, storing the hashes in a database, and then querying the database during clone search. Their proposed index-based approach has the advantage of not needing clustering and expensive pairwise comparisons during clone search.

Keivanloo et al. [66] build on top of the success of [53]. They aim to be more robust to small statement reordering by hashing code fragments at two granularity levels, storing the hashes in a database and querying the database during clone search. The first level of hashing granularity is statement level which is followed by ordering the hashes and then by hashing the first level hashes. An overview of how their technique works is presented in Figure 3.2. [67] further extends [66] and presents a hybrid clone search approach using indexing, information retrieval clustering, and Semantic Web reasoning to attain short response times.

Although these clone search studies are shown to scale to large code bases it is not clear how their work can be naturally extended to handle inter-procedural clones. In this thesis, we circumvent this inter-procedural issue – while maintaining high scalability – by introducing arithmetic operations on vector representation of code.

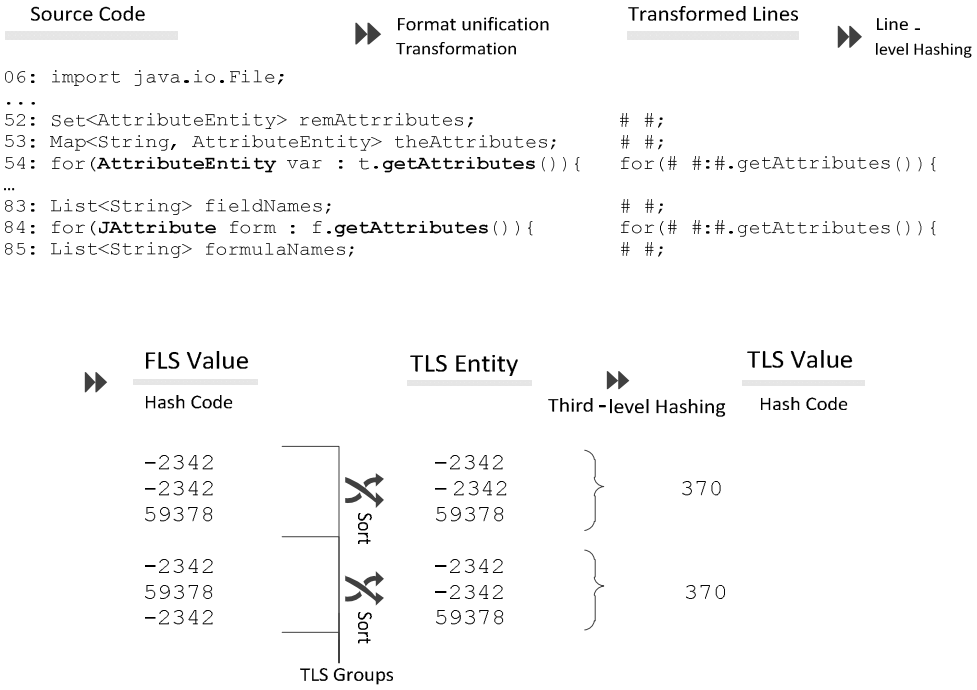


Figure 3.2: Index-Based Clone Search. Taken from [66]

3.6 Automating Program Transformations

Another work similar to ours automates program transformations based on examples of systematic edits [83]. *Systematic edits* are defined by Meng et al. [83] as similar yet not identical changes to multiple code locations usually performed by developers in order to add features or fix bugs. The approach they propose in [83] is based on

code examples containing such systematic edits. Starting from two or more such code examples they create context-aware edit scripts and use the scripts to identify edit locations and transform the code. As refactoring is also a code transformation their approach can also in theory be applied to find refactoring opportunities. 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 thus cannot detect changes that require moving code from one method to another or coordinating changes to multiple methods in the way our approach does. Just like our work, their approach is similar to many studies on programming by examples [47, 83, 84, 88] and specification mining [92, 131, 134, 135] in the sense that we all learn from examples.

Chapter 4

Vector Inlining

4.1 Introduction

The decomposition of a complex method by *extracting a code fragment into a new method* is one of the most extensively performed refactoring operations [85, 124]. Furthermore it can be combined with other code transformations, giving rise to different refactoring operations, in order to address a range of software design problems. This type of refactorings we define as *cross-function refactoring*.

Modern software development environments, e.g. Eclipse, offer support for implementing cross-function refactoring such as *Extract Method*. However the code locations that should benefit from refactoring or *refactoring opportunities* must be manually identified by the developers. This in itself can prove painstaking and nontrivial for large software products. Consequently, a number of works have focused on automating and speeding up the identification process and thus improving the productivity of software development [34, 51, 121]. The identification problem is however challenging for cross-function refactoring.

In this chapter, we present a new scalable approach for identifying cross-function refactoring opportunities (refer to Definition 1.1 for refactoring opportunity). Our

technique builds on top of the success of representing code fragments by characteristic vectors, as defined in previous studies [41, 57, 90]. The salient point of our technique is the design of efficient *vector inlining* operations that simulate the effect of method inlining among code fragments, based on characteristic vector representations of code. Such inlined vectors naturally represent inlined code, taking method extraction and inlining into account. Thus, the problem of scalable identification of cross-function refactoring can be reduced to the problem of scalable identification of similar vectors.

Our investigation is based on the following hypothesis about cross-function refactoring, which we intend to provide evidence of support in this chapter.

Hypothesis for cross-function refactoring

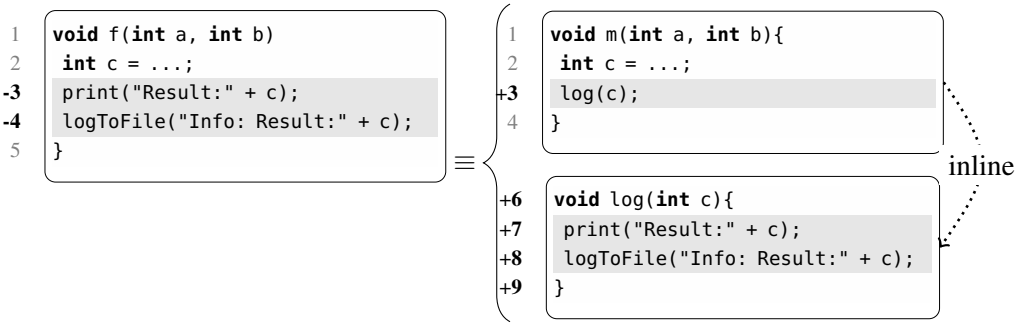
> *If two pieces of code become similar, either syntactically or semantically, after the methods called by them are inlined, they are very likely to indicate a true cross-function refactoring opportunity, especially when the two code pieces are not similar to each other before inlining.*

This means that two pieces of code which have *structural differences related to how this code is bundled in functions* can benefit from applying a cross-function refactoring (e.g. *Extract Method*) to one of the code pieces to make them structurally similar and to increase the modularity of code. Thus, our technique uses a special *vector query and filtering* strategy, which will be presented in Section 4.4.5, to detect pairs of code fragments whose characteristic vectors are *similar to each other after inlining* but have *structural differences related to bundling code in methods before inlining*. Example 12 illustrates such code pieces.

We have implemented our technique for Java in a prototype named REDEX.[4] The tool takes in the source code of a Java program, from which it first creates particular characteristic vectors for every Java method, and then generates inlined vectors by merging the vectors of the methods that have caller-callee relations to simulate the

[4] "ReDex" means refactoring detection in this chapter. We use the name since refactoring operations, especially method extraction/inlining, bear similarity to "reducible expressions" in lambda calculus.

Example 12. *(Extract Method Structural Differences)*



effect of method inlining. It then uses an efficient *vector query* technique, Locality Sensitive Hashing (LSH [24, 44]), together with a set of filters, to search for methods satisfying certain refactoring criteria reporting them as refactoring opportunities. We have applied the tool to 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. REDEX reported 277 refactoring opportunities, and with manual investigation done by 5 students, we found that the detected opportunities are of high accuracy at about 80%, and cover many categories of cross-function refactoring operations from classical collections of refactoring (e.g., [38, 56]), such as *Self Encapsulate Field, Decompose Conditional Expression, Hide Delegate, Preserve Whole Object*, etc.

Our study addresses the problem of scalable identification of missed cross-function refactoring opportunities that have yet to happen; results from our tool can be used to facilitate other tools for performing and validating refactoring operations. Similar to our study, Cider [112], a recent study on detecting refactored clones, can also detect clones that have diverged due to refactoring. However, Cider's detection algorithm works on a graph representation of a program, which is less efficient than REDEX's vector representation and has limited ability in detecting cross-function refactoring. Also, Cider requires initial seeds for its search algorithm, while REDEX works automatically without

the need of seeds. Another study by Meng et al. [83] can also theoretically be applied to detect refactoring opportunities. They create context-aware edit scripts from two or more examples and use the scripts to identify edit locations and transform the code. However, edit-scripts are limited within a single method as, from the author's experience, combining an inter-procedural analysis with the expressiveness of general-purpose edits is a very hard problem.

Our main contributions in this chapter are as follows:

- 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;
- We propose a new technique called *vector inlining* to simulate the effect of method inlining, which enables scalable detection of cross-function refactoring;
- We have evaluated a prototype of our technique on a code base containing 200 projects (4.5M lines of code) from the Eclipse ecosystem, and results show that our prototype can efficiently (in a few seconds) detect more than 200 missed refactoring opportunities with an accuracy of 80%.

The rest of the chapter is organized as follows. Section 4.2 describes more cross-function refactoring examples that can be detected by our technique. Section 4.4 presents our technique in detail. Section 4.5 presents the results of our empirical evaluation and discusses threats to validity. Section 4.6 concludes with future work.

4.2 Cross-Function Refactoring Opportunities

Fowler et al. provide a catalog of refactoring operations [38]. Based on the code transformations they involve, they can be classified as *Extract Method, Inline Method, Replace Temp with Query, Remove Middleman, Introduce Foreign Method, Encapsulate Field, Separate Query from Modifier, or Form Template Method*.

In this thesis, we define *refactoring opportunities* as potential code changes that can fit into classical refactoring categories (i.e., Fowler's categorization [38]) with small variants. Furthermore, lets recall once again that a unique feature of our approach, which aims to increase the chance that a detected refactoring indeed has the potential to improve the design of existing code, is that it targets *missed* opportunities *which are similar to some refactoring that may have happened*. In addition, the term "cross-function refactoring opportunities" in this chapter refers to those categories in Fowler's list that involve method extraction/inlining. In fact in this chapter, all refactoring opportunities mentioned are cross-function refactoring opportunities.

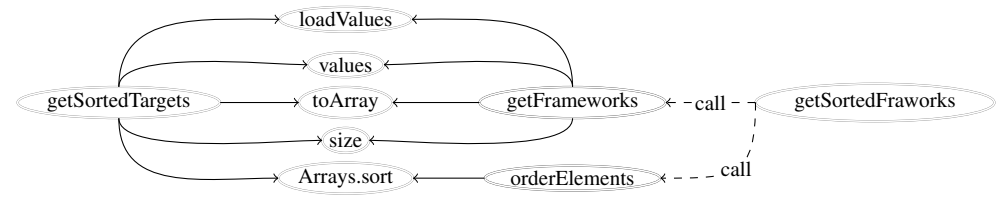
As a first illustrative example let's consider method `escapeString` presented in Chapter 1. According to Eclipse's revision history, method `escapeString` from the JDT package, shown in Example 1, was refactored in version 3.5.2 by an *Extract Method*. Meanwhile, the method with the same name from the Equinox package, shown in Example 2(a), is what we define as a missed *Extract Method* refactoring opportunity. While both methods have the same functionality, they differ in how code is bundled in methods, with the refactored `escapeString` method having enhanced modularity and readability. However, we may also classify the opportunity as *Replace Duplicated Functionality by Existing Method* since `escapeString` from Example 1 was refactored to reuse the functionality of an existing method, `decodeStringValue`. We can also say about the method `decodeStringValue` that it is a *1-way extraction* since compared with its earlier version, it has one method extracted from its body, `escapeChar`.

Another refactoring is illustrated in Example 13, together with the call graph capturing the call relations between the methods illustrated. The single method `getSorted-Targets` is from a class `TargetDefinitionManager` that implements `IRegistry-ChangeListener`. It gets an array of configuration elements and sorts them. The three methods at the bottom belong to a different class, `OSGiFrameworkManager`, that also implements `IRegistryChangeListener`. Although the three methods at

Example 13. *(Separate Query from Modifier Opportunity from Eclipse)*

```
1 public IConfigurationElement[] getSortedTargets() {
2  if (fTargets == null)
3   loadElements();
4  IConfigurationElement[] result = (IConfigurationElement[]) fTargets.values().
      toArray(new IConfigurationElement[fTargets.size()]);
5  Arrays.sort(result, new Comparator() { ... } );
6  return result;
7 }
```

```
1 public IConfigurationElement[] getSortedFrameworks(){
2  IConfigurationElement[] elements = getFrameworks();
3  return orderElements(elements);
4 }
5
6 private IConfigurationElement[] orderElements(IConfigurationElement[] elems)
      {
7  Arrays.sort(elems, new Comparator() { ... } );
8  return elems;
9 }
10
11 public IConfigurationElement[] getFrameworks(){
12  if (fFrameworks == null)
13   loadElements();
14  return
15   (IConfigurationElement[]) fFrameworks.values().toArray(new
       IConfigurationElement[fFrameworks.size()]);
16 }
```



the bottom are spatially apart from each other, they together perform the same functionality as `getSortedTargets`. Based on the enhanced modularity and readability of the code on the bottom, a developer may easily see a refactoring opportunity for `getSortedTargets` as well. Conversely, for reasons such as performance enhancement, a developer may also choose, in reference to `getSortedTargets`, to refactor `getSortedFrameworks` by inlining the methods used in it. We call this example a *2-*

way extraction, since compared with `getSortedTargets`, `getSortedFrameworks` has two methods, `getFrameworks` and `orderElements`, extracted from its body. In general, we could have cross-function refactoring opportunities that are *n-way extraction*.

In Example 14(a), the method `urlDecode` was copied from an earlier version of `urlDecode` in Example 14(b) according to the comments in the code. However, the code in Example 14(b) has gone through refactoring: the `decode` method was introduced to invoke the local method `urlDecode` and the `try-catch` statement was moved from `urlDecode` into `decode`. This indicates the method in (a) has a missed refactoring opportunity. Such a refactoring operation can be classified as *Extract Service Method*. Similar to Example 1, usual clone detection tools may be able to detect parts of the body of both `urlDecode` as clones, but they would *not* be able to link the clones to the additional `decode` method or suggest a concrete way to refactor the code in (a).

Overall, our technique aims to scalably detect missed cross-function refactoring opportunities based on actual refactoring operations that have occurred. REDEX achieves this aim by relying on efficient vector inlining: for every method m in a code base, one or more than one vector is generated to represent m; then REDEX searches for another method m' whose vector(s) can become similar to m's vector(s) if all vectors that are associated by call relations are appropriately *inlined*. The needed similarity search is carried out in the form of a vector query with automated filtering of the results. The results, if any, are presented as a set of pairs of code fragments including the query m and its counter-party, indicating possible ways to refactor m. Section 4.4 has details.

4.3 Background: Characteristic Vectors

Vectors as a representation of code fragments [41, 57, 89] are the key idea of the state-of-the-art code clone detection studies that aim to capture the structure of code yet avoid the high complexity of comparing tree-based (e.g. ASTs) or graph-based structures

Example 14. *(Extract Method Opportunity from Eclipse)*

```
1  org.eclipse.ui.internal.intro.impl.
       model.url.IntroURLParser
2  /* 199
3  * Note: This was copied and adapted
       from
4  org.eclipse.help.internal.util.
       URLCoder
5  200 */
6
7  static String urlDecode(String
       encodedURL){
8   int len = encodedURL.length();
9   ......
10  try {
11   for(int i=0; i<len;) {
12    switch (encodedURL.charAt(i)) {
13    case '%';
14     ......
15    default:
16     os.write(encodedURL.charAt(i++));
17     break;
18    }
19   }
20   return new String(os.toByteArray(),
       "UTF8");
21  } catch(UnsupportedEncodingException
       ex) {
22   return null;
23  }
24 }
```

(a)

```
1  org.eclipse.help.internal.util.URLCoder
2
3  static String decode(String s) {
4   try {
5   return new String(urlDecode(s),"UTF8");
6   } catch(UnsupportedEncodingException uee
        ) {
7    return null;
8   }
9  }
10
11
12 static byte[] urlDecode(
13   String encodedURL) {
14  int len = encodedURL.length();
15  ......
16  for(int i=0; i<len;) {
17   switch (encodedURL.charAt(i)) {
18   case '%':
19
20   default:
21    os.write(encodedURL.charAt(i++));
22    break;
23   }
24  }
25  return os.toByteArray();
26 }
```

calls

(b)

(e.g. PDGs). One of such studies, Deckard [41, 57], uses the occurrence counts of each q-level complete binary subtree in an AST or parse tree to represent the structural information in code fragments. Each dimension of the high dimensional vectors they construct represents the number of occurrences of a particular kind of program element. The advantage of this representation is that efficient near-neighbor search algorithms, found in the database community, such as locality-sensitive hashing [24, 44], can be used to find similar vectors quickly. A formal definition of the characteristic vectors

proposed in the work of Jiang et al [57] is as follows: [5]

Definition 4.1 *(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*.*

Another study, Exas [89], proposes sequences of labels and numbers built from nodes, edges, and paths of various lengths in a generic graph-based representation as dimensions of vectors to represent code. In comparison with these clone detection studies [41, 57, 89] our work explores vectors for the purpose of refactoring detection. Compared to clone detection, refactoring detection is aimed at code fragments that can differ in how statements are ordered or even by how code is split across functions. Our vector representation of choice should thus allow for this. The vectors which our approach is based on are the characteristic vectors proposed by [57]. Though the vectors proposed by Exas capture more complex patterns and structural information about graphs, as we want to perform vector operations that would mimic inlining they would would be more difficult to manipulate. As future work we will investigate the incorporation of our technique into the graph-derived vectors of Exas.

Our characteristic vectors can be generated directly from the abstract syntax tree of a code fragment to represent the *syntactic* characteristics of the code [57]. They can also be generated from certain parts of the abstract syntax tree of the code that match slices of the program dependence. In principle, vectors can be generated from arbitrary combinations of parts of the trees and graphs.

As an illustrating example, Figure 4.1 shows partial ASTs for the code fragments in Example 15. The vector associated with the top "block"-node in Figure 4.1 is the vector for the whole tree shown in Figure 4.1. The elements of this vector indicate

[5] We use the term "characteristic vector" and "vector" interchangeably in this chapter.

the occurrences of nodes of the following types: ⟨`return`, `if`, `for`, `assign`, `init`, `new`, `type`, `funcall`, `.`, `<`, `!=`, `++`, `[]`, `id`, `param`, `const`⟩. Program elements, such as "block" and "parameter" in the boxes with dashed borders in 4.1, are often used to facilitate parsing and considered irrelevant for code semantics, and thus not counted in the vectors. The vectors can be easily generated by traversing the tree from bottom to top and by accumulating counters for various node types. We can also remove certain functionally non-essential code (e.g., simple error-handling code, null-check, assertions, throws, try-catch, etc.) when generating vectors.

Example 15. *(Sample Code: (a) may be refactored as (b))*

```
1 members = getMemberList(...);
2 ArrayList f = new ArrayList<Member>();
3 if ( members!=null ) {
4  IFilter filter = new Filter(...);
5  for(int i=0; i<members.length;i++) {
     if ( filter.satisfy(members[i]) )
       f.add(members[i]);
   }
6 members = f.toArray(new Member[0]);
7 ...
```

(a)

```
1 members = getMemberList(...);
2 if ( members!=null ) {
3  IFilter f = new Filter(...);
4  members = filter(members, f);
5 }
6
7 ......
8 //Code fragment C
9 Member[] filter(Member[] array,
10   IFilter f) {
11  ArrayList r = new ArrayList<Member>();
12  for(int i=0; i<array.length;i++) {
      if ( f.satisfy(array[i]) )
        r.add(array[i]);
    }
13  return r.toArray(new Member[0]);
14 }
```

(b)

Each vector also comes with various meta data (not shown in the figures), such as the name of the method and corresponding file, line ranges of the code fragment, number of tokens contained in the code, etc., to facilitate various postprocessing when needed.

4.4 Methodology

Figure 4.2 illustrates the main steps of our approach. Given a code base, we construct its abstract syntax trees (ASTs), program dependence graphs (PDGs), and call graphs (CGs).

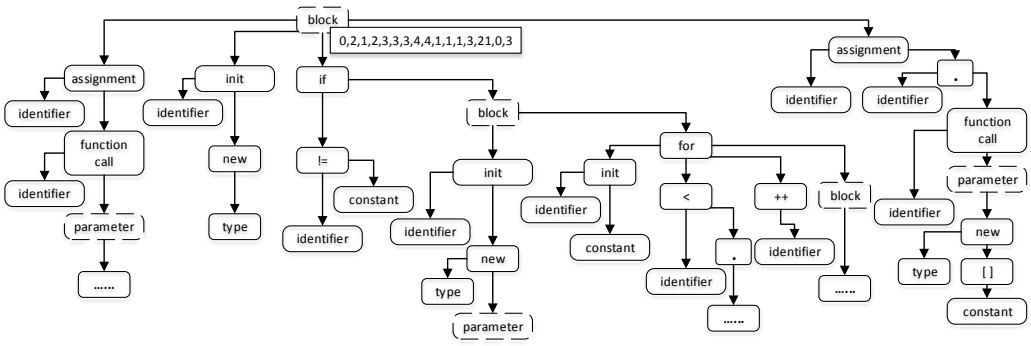


Figure 4.1: AST and Characteristic Vector for Code Fragment (a) in Ex. 15

The ASTs and PDGs are used in a way similar to previous studies [41, 57] in order to generate characteristic vectors for code fragments from the code base (our tailored vector generation is recapped in Section 4.3). These vectors only capture characteristics of the code within a function body: 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 work.

The key novelty of our approach is the use of call relations captured by the call graphs to determine how vectors from different functions can be merged, so that the merged vectors are able to capture cross-function, semantically related code fragments. The merge operation of vectors is in spirit mimicking the effect of method inlining, and thus we call it *vector inlining*, which will be the main subject of Section 4.4.1, and we collectively call such merged vectors *inlined* characteristic vectors.

After vector inlining, an adapted version of Locality-Sensitive Hashing (LSH) [44] is applied to return vectors similar to a vector used as a *query*. Last but not least, all query results are then filtered to identify refactoring opportunities. More details about the query and filtering component will be presented in Section 4.4.5.

In comparison with previous studies, as found in [41, 57], the components in the shaded boxes in Figure 4.2 are new developments of this work. The shaded compo-

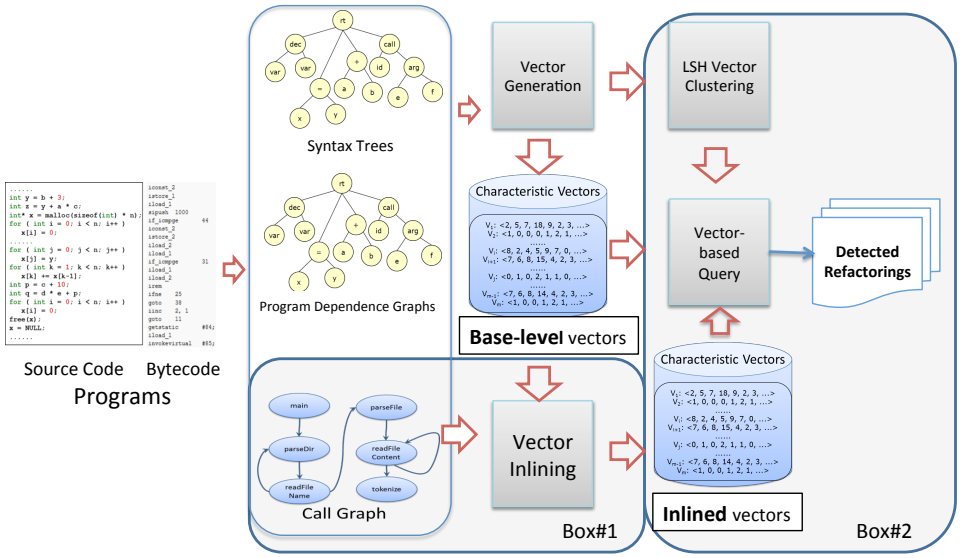


Figure 4.2: Vector Inlining Approach Overview

nent inside box 1 corresponds to vector inlining. The shaded components inside box #2 correspond to vector querying and filtering tailored for cross-function refactoring opportunities.

4.4.1 Inlining Based on Vectors

The key challenge for detecting cross-function refactoring is to efficiently capture the call relations among code and to efficiently search for code having similar functionality in the presence of method calls. Our solution is to use *vector inlining* to simulate the effect of method inlining and extraction.

Given a piece of code c and its corresponding characteristic vector v_c, if c contains a call to a function f, method inlining would replace the function call with the code b in f's body. Intuitively, vector inlining emulating method inlining would replace the parts of v_c corresponding to the call to f with the characteristic vector(s) for the code b in f's body. After the replacement, the changed vector v' would approximately represent the code c inlined with b, taking away the code related to the function call expression.

Algorithm 1 Vector Inlining with Depth 1

1: **Input**: v_c: a vector for a code fragment c that requires inlining
2: **Input**: V: a set of candidate vectors that may be inlined into v_c
3: **Input**: G: a call graph of all code involved
4: **Output**: v_{in}: an inlined vector for v_c
5:
6: Let M_{called} be the *set* of functions invoked by c, which can be obtained from G
7: $V_{M_{called}} := \emptyset$
8: **for all** $m \in M_{called}$ **do**
9: Let V_m be the vector set for m, obtained from V
10: $V_{M_{called}} := V_{M_{called}} \cup V_m$
11: **end for**
12: **if** $V_{M_{called}} = \emptyset$ **then**
13: **return none**
14: **else**
15: v_{in} := `inlineVector`(v, $V_{M_{called}}$)
16: **end if**

Algorithm 1 implements the above idea and accounts for the situations when c may contain zero or more calls. It takes as input a vector v_c, a call graph G, and a set of vectors that can be inlined into others (this set could simply be all available vectors or chosen by users). It identifies all callees of c based on G and collects all vectors for the callees (Lines 7–11), and then calls the `inlineVector` method to inline those callee vectors $V_{M_{called}}$ (could be a very small subset of V) into v_c (Line 15).

The `inlineVector` method transforms the caller vector v_c and all callee vectors in V_{M_c} for inlining: The caller vector is transformed by subtracting the parts from it that represent method invocations and the actual parameters used in the invocations (Line 10); each callee vector is transformed by subtracting the parts from it that represent the return statements (but retaining the expressions actually returned) (Line 12). Then the transformed caller and callee vectors are summed to produce the inlined vector for v_c (Lines 13 and 18).

Algorithm 2 Vector Inlining: Inline direct callees' vectors into caller's

Method `inlineVector`
Input: v_c: a vector from a caller c
Input: V_{M_c}: a set of vectors from c's callees
Output: v: an inlined vector

$v := v_c$; $flag := false$;
$u := \vec{0}$
for all $v_m \in V_{M_c}$ and v_m is a vector from the method m
    and $\neg$`isAPI`$(m, config)$ and `isInlinable`$(v_m, config)$ **do**
  Let v' be the version of v that excludes the call and
      the actual arguments to m
  $v := v'$; $flag := true$
  Let v'_m be the version of v_m that excludes "return"s
  $u := u + v'_m$
end for
if $flag = false$ **then**
  return none
else
  $v := v + u$
end if

4.4.2 Aspects of Inlining

This subsection presents our approach to handle various design considerations related to inlining. These include: handling recursive calls, handling multiple calls to the same function and filtering of methods to be inlined.

Recursive Function Calls

There are pros and cons for inlining the same function into itself or for inlining another function that directly or indirectly calls itself. Recursive inlining may be too expensive, but it may help to capture more "semantic" characteristics of code into the same function, and the following analysis may be more convenient and "accurate." Our vector inlining algorithm provides two capabilities for users to decide how to inline recursive functions.

First, it relies on a control parameter called *inlining depth* (d in Algorithm 3 in Section 4.4.3) to let a user provide a suitable depth of inlining so that we can terminate

(Inlining Example)

Figure 4.3(b) depicts the vectors for the method filter and the code fragment C invoking it. The corresponding source code is given in Example 15. The method call to `filter` is shown by the shaded part in Figure 4.3(b).

By following the algorithm `inlineVector`, the inlining proceeds by first subtracting the vector for the call expression from the vector for the caller, which gives us a new vector:

$$\langle 0,1,0,2,1,1,1,2,0,0,1,0,0,8,0,1\rangle \text{ - } \langle 0,0,0,0,0,0,0,1,0,0,0,0,0,3,0,0\rangle =$$
$$v' = \langle 0,1,0,2,1,1,1,1,0,0,1,0,0,5,0,1\rangle$$

Inlining will proceed according to `inlineVector` at Line 12 by substracting the "return"s from the vector for the function body of the inlined method `filter` (i.e., the vector along with "block" in Figure 4.3(a)), which results in a new vector:

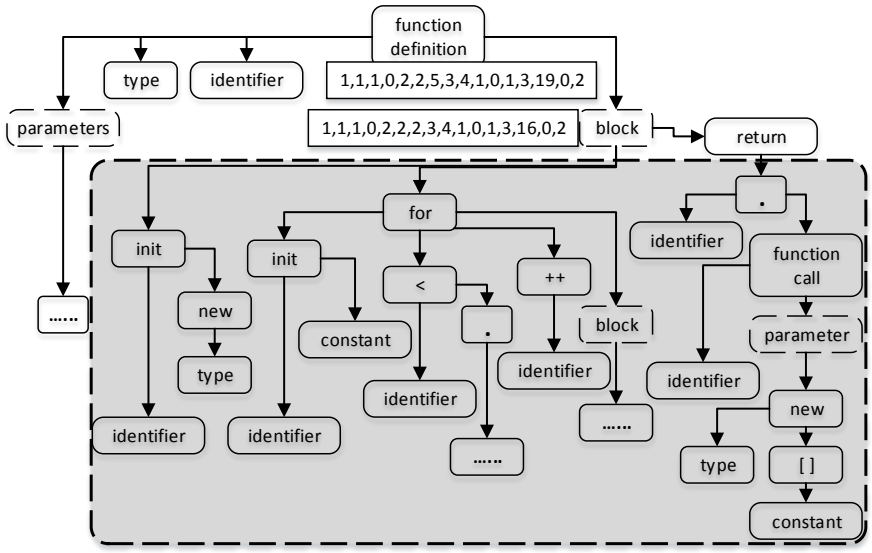
$$v'_f = \langle 0,1,1,0,2,2,2,3,4,1,0,1,3,16,0,2\rangle$$

The last step of the algorithm sums the modified caller and callee vectors together obtaining the vector as earlier shown in Figure 4.1:

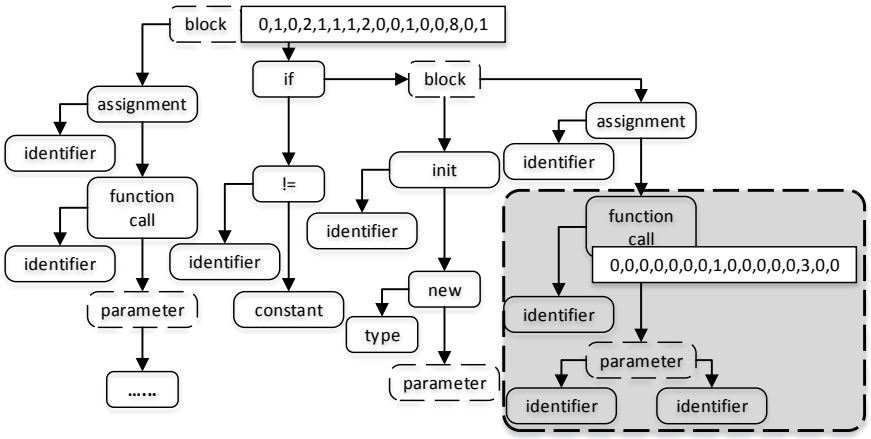
$$v' + v'_f = \langle 0,2,1,2,3,3,3,4,4,1,1,1,3,21,0,3\rangle$$

vector inlining when the depth of inlining is reached. This is an experience-based way to balance the costs and accuracy of cross-function refactoring detection.

Second, it relies on the structure of the given call graphs (used in Algorithm 1) to avoid potential non-terminating inlining. When cycles exist in a call graph and it is requested by a user, we break cyclic call relations in the call graph: Starting from an entry node, we traverse the call graph in a depth-first fashion; if there is no obvious main entry in the call graph, we choose a random one. Then, we remove every back edge



(a) AST and vectors for `filter` method in Example 15



(b) AST and vectors for code C in Example 15

Figure 4.3: Partial, Illustrative ASTs Used for Vector Generation and Inlining

found during the traversal. This back edge removal process is repeated until every node in the graph is traversed. Then, the normal vector inlining is applied.

Multiple Calls to the Same Function

A code fragment may contain multiple calls (with same or different actual parameters) to the same function. For method inlining, the same method is usually inlined multiple times. However, detecting a refactoring may require inlining the same function either once or multiple times. In particular, the code in Example 16 needs inlining of the same method `getSectionName` multiple times to be detected by *Replace Constant with Method Refactoring*. On the other hand, for the refactoring operation shown in Example 17 inlining the same function only once is needed. This is because of the fact that the *Consolidate Duplicate Conditional Fragments* refactoring shown in Example 17 extracts multiple calls to `compute` from the if-then and else branches outside of the if-else statement. Thus we need to only inline once for the vector capturing code on the left to be the same as the vector for the code on the right. In REDEX, we make it an option for users to choose, and by default we inline the same function multiple times.

Example 16. *(Replace Constant with Method Refactoring)*

```
1 IDialogSettings section =
2  master.getSection("pluginsView");
3 if (section == null) {
4  section = master.addNewSection("pluginsView");
5 }...
```

```
1 IDialogSettings section =
2  settings.getSection(getSectionName());
3 if (section == null) {
4  section = settings.addNewSection(getSectionName());
5 }...
6 String getSectionName() {
7  return "EXPRESSION_INPUT_DIALOG";
8 }
```

Example 17. *(Consolidate Duplicate Conditional Fragments)*

```
1 if (discount) {
2  total = price * 0.95;
3  compute();
4 } else {
5  total = price * 0.98;
6  compute();
7 }
```

```
1 if (discount) {
2  total = price * 0.95;
3 } else {
4  total = price * 0.98;
5 }
6 compute ();
```

Inlinable Vectors

Algorithm 1 also allows ignoring certain vectors based on project-specific or user-specific preferences (`isAPI` and `isInlinable` used at Line 9). For example a vector can be deemed as not inlinable if it corresponds to third-party library code; if the code corresponding to the vector is not big enough; or if the code does not contain relevant program elements of interest to the users. Such criteria can be stored in a global configuration file $config$, used to decide whether a vector from a method can be inlined into a vector from the method's caller. In the implementation of REDEX, we check the fully qualified name of each Java method invoked and if the invoked method belongs to certain packages (such as `java.lang.`$*$), then we treat it as an API and do not use it for inlining. Also, if some called methods are interface methods or if some call sites cannot be statically resolved to a unique target method, or if some methods are abstract or refer to native code, we treat them as not inlinable.

4.4.3 Inlining for Multiple Depths

In method inlining, we can choose the inlining depth, from 0, 1, to infinity. Depth 0 effectively means no inlining. Suppose a function f calls another function m: with depth 1, we only inline m's body into f; with depth 2, we inline m's body and also the body of every function called by m into f; and so on, and with depth *infinite*, we risk non-termination by inlining the body of every function called by f, either directly or

Algorithm 3 Vector Inlining with Arbitrary Depths

Input: T: a set of target vectors that may require inlining
Input: V: a set of candidate vectors that may be inlined into vectors in T; V may or may not be the same as T
Input: G: a call graph of all code involved
Input: d: a desired depth of inlining ($d < 0$ means an infinite depth, i.e., to inline as deep as possible)
Output: $I_1, \ldots, I_d$: sets of inlined vectors

Let $i := 0$
Let $I_0 := T$
Let $V_0 := V$
while $|V_i| > 0$ and ($d > i$ or $d < 0$) **do**
    $i := i + 1$
    $I_i := \emptyset$
    $V_i := \emptyset$
    for all $t \in T$ **do**
        $I_i := I_i \bigcup \{\text{Call } Algorithm\ 1(t, V_{i-1}, G)\}$
    end for
    for all $v \in V_{i-1}$ **do**
        $V_i := V_i \bigcup \{\text{Call } Algorithm\ 1(v, V, G)\}$
    end for
end while
$d := i$
return $I_1, I_2, ..., I_d$

indirectly, into f. Similarly, in our vector inlining, we can choose to inline characteristic vectors with various depths.

Algorithm 1 effectively inlines vectors with depth 1. Algorithm 3 extends it to allow arbitrary depths. The correctness of Algorithm 3 can be easily proved based on the correctness of Algorithm 1 and induction on the depth. The complexity of the algorithm is linear with respect to the number of vectors involved and the depth of inlining.

4.4.4 Inlining with Indices for Efficiency

The most time-consuming operations in the above algorithms are related to the repeated lookups in the callgraph for callees in a code fragment (especially when the callgraph is large), and the repeated lookups for vectors corresponding to callee

Algorithm 4 Vector Inlining With Depth 1: With Indices

1: **Input**: v: a vector that requires inlining
2: **Input**: V: a set of candidate vectors that may be inlined into v
3: **Input**: G: a call graph of all code involved
4: **Output**: v_{in}: inlined vector for v
5:
6: Let c be the corresponding code fragment of v
7: Let L be the set of $\langle filename, lineNumber \rangle$ in c
8: Let MV of type: $String \rightarrow SetOfVectors$ the index for vectors
9: Let LM of type: $String \rightarrow lineNumber \rightarrow SetOfStrings$ the index for call relations
10: $M_{called} := \emptyset$
11: **for all** $\langle filename, lineNumber \rangle \in L$ **do**
12: $\quad M_{called} := M_{called} \bigcup LM[filename][lineNumber]$
13: **end for**
14: $V_{M_{called}} := \emptyset$
15: **for all** $m \in M_{called}$ **do**
16: $\quad V_m := MV[m]$
17: $\quad V_{M_{called}} := V_{M_{called}} \cup V_m$
18: **end for**
19: $v_{in} :=$ `inlineVector`$(v, V_{M_{called}})$

signatures, which become particularly expensive when the set of vectors to be inlined is large. However, these operations can be implemented in an efficient way by having various indices to speed up the lookups. The idea is to construct indices among source code locations (file names, method names, and line numbers), methods, and their corresponding vectors. Algorithm 4 optimizes Algorithm 1 and is much more efficient with the introduction of additional indices. If we used multi-sets, instead of sets, to store methods (Lines 9 and 12 in Algorithm 4), we can then inline the same method more than once as discussed in Section 4.4.2.

4.4.5 Vector Query And Filtering

With vector inlining that simulates the effect of method extraction and inlining, the problem of scalable detection of cross-function refactoring can be reduced to finding similarity among base-level and inlined vectors by means of *vector query* and *filtering*.

As mentioned earlier, our intuition, which has been expressed as a hypothesis earlier, is repeated here: *If two pieces of code become similar, syntactically or semantically, only after the methods called in them are inlined, then they are likely to indicate a cross-function refactoring opportunity, especially if the two code pieces are not similar to each other before inlining.*

The purpose of *vector query* is to find vectors, from a given set of candidate vectors, that are similar to a vector used as a *query*. Our vector querying engine takes as input a pair of vector sets: the first is the *query set* containing all query vectors, and the second is the *target set* containing all candidate vectors. The query engine then returns a set of pairs; each pair represents a match between a query vector and a target vector. As an example, consider a query set only containing the base-level vector for the code fragment (a) in Example 2 and a target set only containing the inlined vector for the cross-function code fragment (a) and (b) in Example 1. Running the query engine will return the pair formed by the base-level vector for Example 2(a) and the inlined vector for Example 1(a) and (b), and the corresponding source code would be presented as a potential refactoring opportunity.

Formally, a query execution will return a subset of the cartesian product $A \times B$, where A is the *query set* and B is the *target set*. The results can thus can be derived from the two sets of vectors as follows:

$$\begin{aligned} \otimes &: \mathcal{P}(Vector) \times \mathcal{P}(Vector) \rightarrow \mathcal{P}(Vector \times Vector) \\ Q \otimes T &= \{(v_1, v_2) \mid v_1 \in Q, v_2 \in T, v_1 = v_2, v_1.name \neq v_2.name\} \end{aligned}$$

Similar to previous studies on clone detection [41, 57], we adapt Locality-Sensitive Hashing (LSH) [44], which is designed to efficiently handle nearest-neighbor queries of high-dimensional data, to implement our query engine. Our query engine first stores the

target set into LSH's internal hash tables, then uses every query vector from the query set to get matching target vectors for each query vector via LSH backend, and presents all query results as a set of pairs of matching vectors. The LSH backend from Alex Andoni (http://www.mit.edu/ andoni/LSH) is capable of handling a couple of millions of vectors at a time.

Besides querying for matching code, we also need to identify matching code that may manifest cross-function refactoring. Thus, our query engine also defines a set of *filters* for matching vectors, based on heuristics, to identify more likely cross-function refactoring opportunities. The following defines the query and filters used in REDEX.

Definition 4.2 *(Split Query)*

Given two vector sets B_a and I_n, where B_a contains only base-level vectors and I_n contains only inlined vectors, a Split Query *returns a set of pairs of similar vectors; every pair in the set contains one vector from B_a and another vector from I_n.*

A *split query* uses base-level vectors in the query set and inlined vectors in the target set. It allows us to ask whether code contained in one function is similar to code that spans more than one function. A positive answer may provide an opportunity to create a more modular version of the code used as the query, by means of method extraction.

Results from the above query can be further refined by *filters*. A *filter* defines a set of constraints over a pair of vectors, and *removes the pairs that satisfy the constraints*. Some of the filters we have defined look into the origin of the inlined vectors to make filtering decisions. This lookup can be efficiently implemented (O(1)) by keeping an index of the original base-level vectors and searching the index. To facilitate discussion, let us define several notations. Given a method or code fragment m, I_m denotes the set of methods invoked by m, v_m^0 denotes the base-level vector (no inlining) for m, and v_m denotes the inlined vector when the vectors for all methods in I_m are inlined (with depth

1) into v_m^0. Now, we define the following filters for refining query results.

Definition 4.3 *(Filter Equal)*

> *Given a pair of vectors,* (v_q^0, v_r)*, corresponding to the vectors for the methods* q *and* r *respectively,* Filter Equal *first determines* (v_q^0, v_r^0) *corresponding to the base-level vectors for methods* q *and* r*, and then removes the pair if* v_q^0 *and* v_r^0 *are equal.*

Filter Equal aims to eliminate those pairs where the vectors before inlining are equal: As the code fragments for the two base-level vectors are equal, they are unlikely to indicate a refactoring opportunity.

Definition 4.4 *(Filter Simple)*

> *Given a pair of vectors* (v_q^0, v_r) *corresponding to the vectors for the methods* q *and* r *respectively, and* r *invokes a method* i*, this* Filter Simple *removes the pair if* $|I_r| = 1$ *and* v_q^0 *is equal to* v_i^0*.*

It is obvious that when $|I_r| = 1$, i is the only method invoked by r. In addition, when $v_q^0 = v_i^0$, together with the query premise $v_q^0 = v_r$, we have $v_r = v_i^0$ and can infer that the method r does nothing except invoking i. Thus, *Filter Simple* eliminates those pairs where the possible refactoring opportunity is to simply fold or unfold a method wrapper.

Definition 4.5 *(Filter Size)*

> *Given a pair of vectors* (v_q, v_r) *or* (v_q^0, v_r) *, this* Filter Size *removes the pair if* v_q *or* v_q^0 *contains less than 20 nodes.*

Filter Size filters out query results whose query vectors are too small in terms of numbers of nodes contained so that we only report refactoring opportunities for code of non-trivial sizes to help reduce possible false positives. An example of such a pair of small vectors is shown in Figure 4.4.

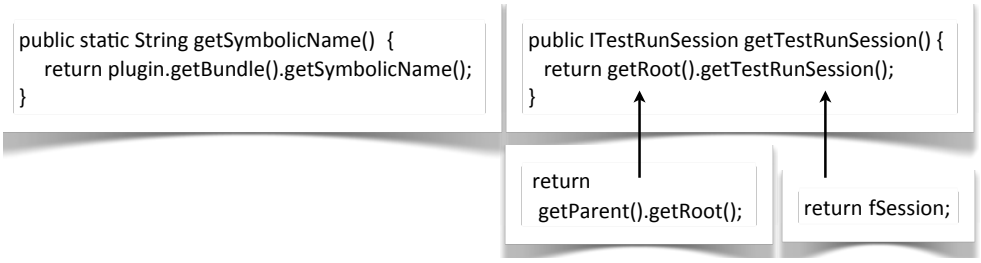


Figure 4.4: Small Vector Inlined

4.4.6 Implementation Remarks

Our approach relies on syntax trees, call graphs, and program dependence graphs of a code base. The general idea of vector inlining is independent of programming languages. However, the tools available for constructing these data structures are often language-specific. Various parts of our algorithms (e.g., method signatures used in Algorithms 4, some program elements –e.g., method invocations, return statements – used in Algorithm 1, 2, etc.) are language-specific. Implementation of some of the data structures may also need adjustments for different languages.

Given a Java code base, we use Eclipse JDT [29] to construct ASTs for the code base, one AST for each Java source file, which is split into smaller ASTs for each function. We then use IBM WALA [54] to construct DDGs and CGs for the code base, one DDG for each function, and a big CG (which may be comprised of multiple disconnected subgraphs) for the whole code base. The ASTs and DDGs are outputted into the .dot format and read by the vector generation component of our system which then proceeds to find for each DDG its matching AST and to construct a number of characteristic vectors. We use fully qualified names and unique identifiers for the .dot files so that we can match DDGs with ASTs easily. We limit DDGs within procedures so as to avoid potential scalability issues for inter-procedural dependence graph construction.

Similar to previous work [114], we also eliminate safety-checking code from the

DDGs. One of the patterns of code that we target is shown below.

```
if(info==null} { throw new Exception(...); }
```

This pattern throws an Exception when a certain property is violated. As such code snippets have little contribution to the main functionality of code, discarding them allows us to detect more clones and ignore associated bug-reporting code.

Due to the large size of our subject programs, the call graph construction poses a challenge. In order to obtain a call graph, WALA first loads all classes into memory, creates the class hierarchy of the classes, obtains the intermediate representations of methods, creates the DDGs and constructs the call graph. The creation of the class hierarchy is needed in order to resolve the possible targets of each virtual method call. As polymorphism is a commonly used feature of Java and a virtual method call can be dynamically dispatched to different method bodies at runtime, we need to determine the number of targets of each call in order to accurately create a callgraph. The method calls that are statically determined to correspond to multiple or no method targets are deemed as not inlinable by our approach.

The implementation of our approach relies on careful memory management and efficient data structures. In particular, our proposed solution to generating the callgraph for large programs is based on trading computation for memory, and basically constructing the DDG for a method m on demand: either when the body of m is analyzed or during the resolution of a virtual method call c if c can correspond to m. Though this approach may lead to the DDG for one method being computed multiple times (caching can be employed to help here) the memory requirements are low and it can accurately handle large programs such as Eclipse.

When an invoked method (e.g., a third-part library) is not part of the subject programs, no vector is generated for that method of course. For consistency and ease of handling, we thus do not inline any method that contains a call to any method without vectors, and leave it future work to explore the effect of allowing partial inlining.

4.5 Empirical Evaluation

Our experiments evaluate the effectiveness and scalability of *vector inlining* in detecting *cross-function refactoring opportunities*. The experiments show that our approach scales to a large code base and detects many refactoring opportunities with high accuracy. Furthermore the results give evidence that *detecting missed refactoring opportunities is indeed needed*.

4.5.1 Setup and Infrastructure

We have experimented using REDEX. REDEX uses a modified version of Deckard, which provides a wide choice of granularity levels at which vectors can be generated: either whole methods, slices of methods obtained from PDGs, or any fragment of code in a method. For our experiments we used the most efficient granularity and only generate vectors that represent whole methods[6]. The vectors we experimented with have 98 features. These features are presented in Figures A.1 and A.2 in Appendix A. The first 84 features are the types of ASTNodes generated by Eclipse JDT [28]. Separated from the usual $method\_invocation$ feature, our vectors also contain the $api\_invocation$ feature that refers to invocations of methods not defined in the subject programs. Furthermore, the last 14 features are $method\_invocation\_paramno$ and $api\_invocation\_paramno$ that denote invocations with the number of actual arguments specified by $paramno$ where $paramno \in \{0, 6\}$.

Also, we focused on detecting refactoring within and across the subject projects and ignored potential refactoring that may span across methods defined in external libraries. Thus, our *vector inlining* algorithm was configured to only inline a method if the method is defined in one of the subject projects (checked by `isAPI` and `isInlinable`

[6]Code in a whole method includes all executable code in the method, but excludes function headers, variable declaration, simple elements unlikely responsible for the main functionality of the code (e.g., simple null-check and return, throw exceptions), and non-executable lines (e.g., comments, blank lines, lines with only curly braces).

in Algorithm 1). In addition, vector inlining was carried out in an *all* mode, i.e., all inlinable and non-API methods called in a code fragment are inlined; if any one of the methods in a code fragment cannot be inlined due to any reason (e.g., missing vectors due to parsing errors, unresolved call targets, etc.), the inlining for the code fragment would be cancelled. Further, we focus on evaluation of depth-1 inlining and the type of queries and filters described in Section 4.4.5.

The experiments on REDEX were performed on a PC running Ubuntu 10.04 with Intel Xeon at 2.67GHz and 24GB of RAM.

The subjects of our evaluation are comprised of 200 bundle projects in the Eclipse 4.2.2 ecosystem, including Eclipse Core, Eclipse JDT, Eclipse PDE, Eclipse Equinox, Apache Commons, Apache Lucene, Hamcrest, ObjectWeb ASM etc. The projects encompass more than 20,000 Java files, 40,000 classes and 7,000 interfaces, and contain about 4.5 million lines of code and a long evolution history.

We chose the subject projects according to two criteria: 1) they represent a variety of projects with a long evolution history that have gone though refactoring (the study of Xing et al. [130] on the structural evolution of Eclipse shows that about 70% of structural changes may be due to refactoring) and 2) they represent a large scale ecosystem of projects and packages that may share common parts (e.g. the `Disassembler` class, illustrated in Chapter 1, contained in both the Eclipse `JDT` project and the Eclipse `Equinox` project in different forms).

4.5.2 Results and Analysis

Scalability

The first question we wanted to answer in this evaluation is related to the scalability of our approach to large projects. We measure three aspects when determining the scalability of REDEX: the time it takes to generate the characteristic vectors, the time it

takes to perform vector inlining and the number of vectors generated.

The most expensive parts in terms of both time and memory consumption are the construction of the callgraph (CG), ASTs, PDGs, and the generation of indices (cf. Section 4.4.4) for these data structures. PDG and CG constructions using WALA [54] took about 44 minutes; vector inlining (including building indices) took 3 minutes: while index generation took most of the time, the actual vector inlining (cf. Algorithm 4) took a couple of seconds for inlining of depth 1. Fortunately, all these constructions are one-time cost, and more optimizations can be performed in future for the constructions.

REDEX generated about 186K base-level characteristic vectors, each of which represents the body (please refer to footnote [6] for details) of a defined method in the Eclipse ecosystem (excluding abstract, native and interface methods or external methods defined outside of Eclipse). Thus, about 186K queries were performed and filtered; they accumulatively took less than 2 minutes to report potential refactoring opportunities which is quite little considering that the subject program is about 5 millions of lines of code and the analysis is inter-procedural. As a very coarse comparison the authors of Cider report in their evaluation more than 300 seconds for a project of 50KLOC.

Accuracy - Cross-Function Refactoring Opportunities

Recall that the hypothesis set forward in this chapter was that if two pieces of code become similar only after the methods they call are inlined a cross-function refactoring may be applicable. In order to verify this hypothesis and the accuracy of *vector inlining for cross-function refactoring* we performed the *Split Query* introduced by Definition 4.2 and the type of filters described in Section 4.4.5.

We have detected many missed refactoring opportunities in the bundle projects of Eclipse. Specifically, REDEX generated 277 reports for the 200 bundle projects in the Eclipse 4.2.2 ecosystem. Each report is a pair of two pieces of code that may span multiple functions: one corresponds to the query generating the report, and the other

corresponds to the target matching the query. Each of the two pieces of code may reveal a refactoring opportunity and could be refactored according to its counter-party.

Validation of the results was done through a user study by 5 graduate students with good knowledge of Java and refactoring. These report inspectors were required to classify each of the reports (i.e., resulting pairs) into one of the following four options: (1) `refactoring`, (2) `not refactoring but clone`, (3) `neither refactoring nor clone`, or (4) `I don't know`. In the case when the `refactoring` option is selected, they were required to classify, additionally, the refactoring operation that might be applied to one of the code fragments in the pair.

Overall, the results after inspection showed a high accuracy at 80% for the cross-function refactoring opportunities detected by REDEX: 80% of all reports were classified as option (1), 16% as option (2), and 4% as options (3) or (4). The validation exercise discovered 223 out of the 277 analysed cases to have true refactoring opportunities. These true refactoring opportunities are each matched to one of many categories and variants of Fowler's catalog. Table 4.1 shows these categories and the number of validated refactoring opportunities. These provide strong evidence to support the ability of REDEX in detecting missed refactoring opportunities. Some examples of refactorings have been shown in Section 4.2.

Practical Usage - Inter-Projects Divergence

One of the reasons we chose the projects in the Eclipse ecosystem as the subject of our evaluation was because they share a long evolution history and possibly common parts that may have evolved differently by refactoring (a.k.a., *diverged*). The third question of the evaluation aims to determine whether this is indeed the case.

Figure 4.5 shows the distribution of the refactoring opportunities in the subject projects. One can see that many projects are covered by the vectors. Figure 4.6 shows

Refactoring Categories	**Occurrences**
Self Encapsulate Field	76
Encapsulate Collection Access and Downcast	19
Downcast Encapsulate	2
Decompose Conditional Expression	2
Substitute Algorithm	23
Extract/Inline method	25
Separate Query from Modifier	2
Introduce Query Method	18
Replace Duplicated Functionality by Existing Method	10
Hide Delegate	23
Preserve Whole Object	3
Introduce Parameter Object	7
Reverse Conditional	3
Replace Temp with Chain	1
Make Method Static	11

Table 4.1: Categories of Detected Refactoring Opportunities [7]

a heatmap of the number of reports (ie., resulting pairs of code) between pairs of projects in the evaluation. Values on the diagonal indicate refactoring opportunities detected within the same project. On the other hand, Figure 4.6 also shows many cross-project refactoring opportunities. This shape of the heatmap results supports our intuition that in large projects initially similar code across different functions and projects is refactored differently or *diverge*, which increases the difficulty for their identification, and techniques that can detect cross-function refactoring opportunities are indeed needed.

[7] The number of refactorings recorded and displayed in Table 4.1 may be bigger than the number of reports returned by REDEX. This is due to the fact that a piece of code may be refactored in more than one way to obtain the shape of its counter-party, providing additional choices for users of REDEX.

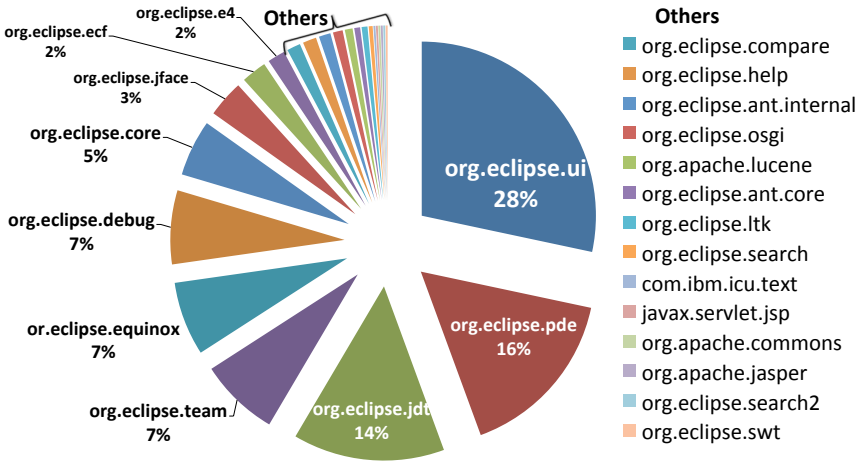


Figure 4.5: Distribution of Covered Code

Furthermore, our evaluation showed that a large number of vectors in the result set, 97%, are the result of inlining one method. This is consistent with the fact that most refactorings that involve cross-function changes in Fowler's catalog commonly only involve extracting/inlining one method.

Analysis

During the investigation, the report inspectors checked if one of the code fragments in a report can be refactored in accordance with Fowler's categorization [38], by comparing its shape with its counter-party. At the same time, we allow their own best judgements and small variants to Fowler's categories as shown in Table 4.1. Some of the reports were validated by multiple inspectors, which resulted in interesting observations. For simple refactorings, such as *Self Encapsulate Field*, the type of refactoring was mostly correctly identified by all. For more complex refactorings, such as *Preserve Whole Object* or *Separate Query from Modifier*, there were variations between the types of refactoring classified by the inspectors. Example 18 was classified as both `I don't know` and *Preserve Whole Object*. The report consists of two methods `convertSeverity` and `convertLevel` that return an integer. Although they have similar functionality,

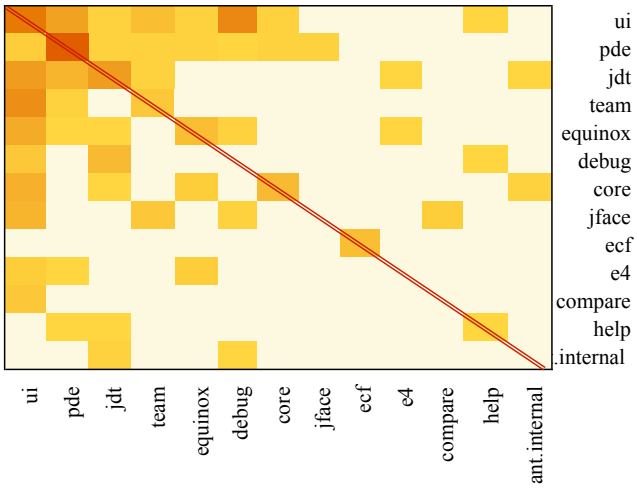


Figure 4.6: Heat Map of Reported Refactoring Opportunities

the two methods differ in their parameters. `convertLevel` receives an object as parameter and calls a member function of the object to access the data needed inside `convertLevel`. `convertSeverity` on the other hand, receives an `int` value obtained by a call to a member method of an object `entry` of the type `LogEntry` before calling `convertSeverity`. Sending the whole object into a method makes it more robust to certain functionality changes and avoids problems when the method needs new data values from the object later. Thus, to make `convertSeverity` more robust against certain changes, we may refactor `convertSeverity` to use an object of the type `LogEntry` as its parameter without affecting its performance, and thus it can be classified as `Preserve Whole Object`. For such cases where the refactoring types varied, we applied our own best judgments and chose from the types selected by the reporters.

An interesting class of reports from the results is represented by code fragments where the *Make Method Static* refactoring available in IntelliJ IDEA [56] can be applied. 40 such reports were classified as `not refactoring but clone` by an inspector with the comment that “one side of the code is not directly refactorable into the other yet

they are similar (possible diverged from one source) and can be made more reasonable with refactoring techniques". These reports are however not counted positively toward the accuracy of our approach. Counting these reports as *Make Method Static* refactoring opportunities would have increased the accuracy to 94%.

Example 18. *(Preserve Whole Object)*

```
1 org.eclipse.core.EclipseLogWriter
2 private static int convertSeverity(int entryLevel) {
3  switch (entryLevel) {
4   case LogService.LOG_ERROR :
5    ....
6    }
7 }
8
9 public void logged(LogEntry entry) {
10  ...
11  convertSeverity(entry.getLevel()) ..
12
13 }
```

```
1 org.eclipse.core.EclipseLogFactory
2 static int convertLevel(FrameworkLogEntry logEntry) {
3  switch (logEntry.getSeverity()) {
4   case FrameworkLogEntry.ERROR :
5    ...
6  }
7 }
```

4.5.3 Discussion and Threats to Validity

Our approach depends on the setting of some parameters. Some of them are related to code similarity metrics and common to most clone detection tools. For example, the minimal number of tokens or nodes that a code fragment needs to contain, and the difference (or similarity) allowed between two code fragments for them to be detected. REDEX is only evaluated with code sizes larger than 20 (cf. Filter Size in Definition 4.5) and similarity 1.0, which denotes that LSH will return for a query vector all vectors

exactly the query.

REDEX is only evaluated with Split Query (cf. Definition 4.2) that uses base-level vectors as the query set and inlined vectors as the target set. It is possible to use different query sets and target sets to find additional cross-function refactoring opportunities. The filters we used are relatively simplistic; a more comprehensive filtering constraints may be developed based on common refactoring operations (e.g., Fowler's categories and other collections [38, 56]) to look for refactoring opportunities more accurately.

A number of other parameters control what can be inlined in our algorithms. For example, the *depth* of inlining in Algorithm 3 and whether to inline a function more than once (cf. Section 4.4.2) affect the number of functions inlined together for refactoring detection. Also, since vectors can be generated for arbitrary code fragments, not just whole methods, inlining can be carried out for vectors of arbitrary code fragments, not just method-level inlining, which may be expected to produce more refactoring opportunities. We currently use an *all mode* to inline all vectors for all methods invoked by a code fragment; we will expect to detect more refactoring opportunities if we allow to inline subsets of the set of invoked methods or *partial inlining*.

We leave it as a future work to explore the large configuration and parameter space to balance the number of refactoring opportunities we can detect with their accuracies.

In our empirical evaluation, we evaluated the accuracy of the reported refactoring via manual investigation by students. This introduces experimenter bias in our study. The students' Java programming skills and knowledge about refactoring may also affect how they label the reports. We also limit our evaluation to Java and thus our results may not be applicable to other programming languages. In the near future, we plan to port our implementation to other languages, extend our evaluation to more programs, and conduct both automated evaluation against historical refactoring operations and more systematic user studies to alleviate the above threats to the validity of our approach.

4.6 Chapter Summary

This chapter presents a novel, automated and scalable technique for identifying cross-function refactoring opportunities that span more than one function (e.g., Extract/Inline Method). The key to the success of our technique is the design of efficient *vector inlining* operations that simulates the effect of method inlining among code fragments, so that the problem of identifying cross-function refactoring can be reduced to finding similar vectors before and after vector inlining.

We have implemented our technique in a prototype tool named REDEX which encodes Java programs to particular vectors. We have evaluated our technique on a large code base (4.5 MLOC) comprising of 200 bundle projects in the Eclipse ecosystem (e.g., Eclipse JDT, Eclipse PDE, Apache Commons, etc.). The results show that ReDex can find 277 cross-function refactoring opportunities in 2 minutes, and 223 cases were labelled as true opportunities by users, and cover many categories of cross-function refactoring operations in classical refactoring books, such as *Decompose Conditional Expression, Hide Delegate, Preserve Whole Object, Introduce Parameter Object*, etc.

Chapter 5

Vector Abstraction and Concretization

5.1 Introduction

In the previous chapter, we presented a technique for efficiently encoding *inlined* code. Our experiments showed that this technique can be used to accurately detect code transformations that involve *moving* code fragments across function boundaries such as those induced by *Extract Method, Preserve Whole Object, Self Encapsulate Field, Replace Temp With Query*, etc. There are however other kind of refactoring operations the detection of which requires knowledge about the precise change patterns induced by refactoring and to *flexibly omit or emphasize specific program elements* when computing similarity between two code fragments. Let's illustrate the challenges as mentioned above by the following refactoring operation.

The code fragments in Example 19 were detected as a refactoring opportunity of type *Introduce Explaining Variable* in a program named XML Security. *Introduce Explaining Variable* is a very common refactoring that involves improving the readability of a complicated expression by means of putting the result of an expression, or parts of an

Example 19. *(Introduce Explaining Variable Refactoring)*

```
1 public Key getKey(String alias, char[] password)
2  throws NoSuchAlgorithmException, UnrecoverableKeyException {
3  try {
4   KeyElement keyElement = new KeyElement(this.
      getKeyEntryElement(alias),this._baseURI);
5   return keyElement.unwrap(password);
6  } catch (XMLSecurityException ex) {
7   throw new UnrecoverableKeyException(ex.getMessage());
8  }
9 }
```

(a)

```
1 public Certificate[] getCertificateChain(String alias) {
2  try {
3   Element keyElement = this.getKeyEntryElement(alias);
4   if (keyElement != null) {
5    KeyElement ke = new KeyElement(keyElement, this._baseURI);
6    return ke.getCertificateChain(alias);
7   }
8  } catch (XMLSecurityException ex) {
9   ex.printStackTrace();
10 }
11 return null;
12 }
```

(b)

expression, in a temporary variable with a name that explains the purpose. This change pattern is present in Example 19. Specifically, as opposed to code fragment (a), code fragment (b) declares a *new variable* at Line 3 and then reads it at Line 5. Usual clone detection may detect (a) and (b) as code "clones" (i.e., code fragments similar to each other [36, 59, 62, 71, 103]). Unfortunately, it may also return a large number of clones that exhibit other differences and not the specific "change pattern" sought after (aka., high number of false positives).

Moreover refactorings can be combined together giving rise to more complex "change patterns". Example 20 shows code which exhibits a "change pattern" induced by the combination of two refactoring operations: *Inline Temporary Variable* (the reverse operation of *Introduce Explaining Variable*) and *Extract Method*. The body of method

`getControllers`, depicted by (a), is similar to `getVisualizer`, depicted by (c), when we inline the method invocation to `getVector`. *Inline Temporary Variable* and *Extract Method* can be performed on `getControllers` to obtain the same structure as `getVisualizer` and to remove duplication since the refactored code would reuse the functionality of an existing method, `getVector`.

Example 20. *(Inline Temp and Extract Method from JMeter)*

```
1  public static Vector getControllers(Properties properties){
2   String name = "controller.";
3   Vector v = new Vector();
4   Enumeration names = properties.keys();
5   while (names.hasMoreElements()){
6    String prop = (String) names.nextElement();
7    if (prop.startsWith(name)){
8     Object o = instantiate(
9      properties.getProperty(prop),
10     "org.apache.jmeter.control.SamplerController");
11    v.addElement(o);
12   }
13  }
14  return v;
15 }
```

(a)

```
1  public static Vector getVector(Properties properties, String name) {
2   Vector v = new Vector();
3   Enumeration names = properties.keys();
4   while (names.hasMoreElements()) {
5    String prop = (String) names.nextElement();
6    if (prop.startsWith(name)) {
7     v.addElement(properties.getProperty(prop));
8    }
9   }
10  return v;
11 }
12 ...
13 public static Vector getVisualizer(Properties properties) {
14  return instantiate(
15    getVector(properties, "visualizer."),
16    "org.apache.jmeter.visualizers.Visualizer");
17 }
```

(b)

(c)

Let us denote the inlined version of method `getVisualizer` in Example 20 (c) 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 [23, 41, 57, 60, 68, 91, 102]. This is so because these techniques cannot flexibly omit or emphasize specific program elements (e.g., the variable declaration and assignment at Line 2) when computing similarity. The additional variable declarations and assignments in (a) do not exist in (c) or (c^I), a difference which can be described as the result of an *Inline Temporary* refactoring operation. This indicates that a *desired refactoring detection tool should possess knowledge about, as well as work flexibly with, various kinds of refactoring operations*.

On top of all these challenges to make refactoring detection flexible and accurate, we also would like to locate suitable refactoring candidates from millions of lines of code very efficiently, as we have been advocating throughout this thesis.

In this chapter, 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. The approach in this chapter employs the same idea as the previous chapter, and extends it to define vector abstraction and concretization for detecting specific types of refactorings.

We have implemented our approach for Java, generating vectors for both source code and bytecode, and performed 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 invocations to 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, and concretizes the abstract 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 of refactoring operations for all Eclipse projects, 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), each version having an average size of 33KLOC, our tool reported 191 historical refactorings across various versions and more than 2.8K instances of refactoring opportunities. Through both automated and manual validation by four graduate students, we find that the detected refactorings are of high accuracies, about 92% for detected historical refactorings and about 87% for detected refactoring opportunities.

Our main contributions in this chapter 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;
- 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 chapter is organized as follows. Section 5.2 presents our detection approach. Section 5.3 presents specific vector abstraction and concretization operations used in our detection approach. Refactoring as Vector Abstraction and Concretization Section 5.4 presents the results of our empirical evaluation, Section 5.4.4 discusses threats to validity. and Section 5.5 concludes with future work.

5.2 Methodology

We explain the main steps of our approach along with Figure 5.1. 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 [41, 57] to generate characteristic vectors for code

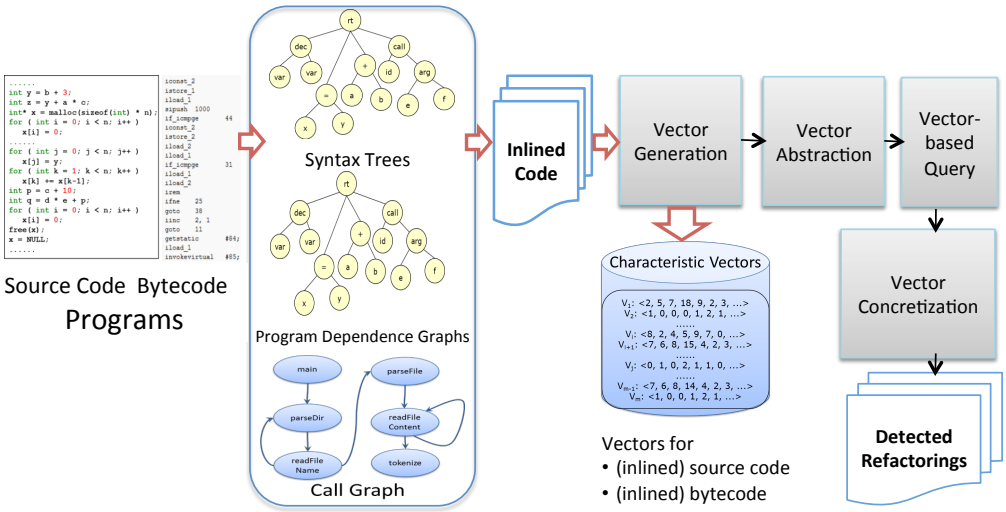


Figure 5.1: Overview of Our Approach

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 [104]. 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 [104]. We also simulate the effect of method inlining 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 5.2.1.

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 Section 5.2.2 and 5.3 for more details).

Then, a hash-based search (simple hash and locality-sensitive hashing –e.g. LSH [44]–) is employed to query for similar *abstract* vectors efficiently so that we can identify candidates for refactoring (refer to Section 5.2.3). 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 refactoring (see Section 5.2.4). 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.

5.2.1 Vector Generation

As the work described in Chapter 4, we represent code fragments as high dimensional vectors in the form of $v = \langle v_1, v_2, ..., v_n \rangle$, where v_i represents the number of occurrences of a particular kind of program element. 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 [41, 57].

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. As an illustration, in Table 5.1, we provide some samples of heavily simplified vectors representing the code fragments depicted in Example 20. The table headers indicate the sample features used for the vectors and rows 1–3 indicate the vectors for each of the three methods. Separated from the usual method invocation ("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.

		Features									
row ID	Code	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 5.1: Sample Partial Vectors for Code Fragments in Example 20

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. As in Chapter 4, we call these vectors *base-level* characteristic vectors, or simply *base* vectors.

For Inlined Code

Refactoring may involve different ways of extracting methods. Simulated method inlining via vector manipulation has been shown in Chapter 4 to be effective for detecting method extraction. As it is important to the understanding of the techniques presented in this chapter we will briefly recap the technique of *vector inlining*.

Vector inlining essentially simulates 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. For the features shown in Table 5.1, vector inlining involves reducing the occurrence counters for "mth invoc." and "simple

name" each by one for each method that is inlined ("simple name" is a child node of "mth invoc.", representing the method name in the syntax trees generated by Eclipse JDT for Java), and removing all counts for `return`s from the callee. For example, when we inline `getVector` into `getVisualizer`, the vector for `getVisualizer` is changed as row 4 in Table 5.1. The red parts of the row indicate the manipulations applied to the sum of rows 2 and 3 to simulate the inlining. 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.

This chapter extends vector inlining to allow for more types of inlining. As opposed to Chapter 4, where we experimented with inlining all method calls, in this chapter we consider different ways to inline methods, for a given code fragment c, in order to encode various possible changes induced by method inlining or extraction. 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 chapter we inline methods invoked in c in mainly three different modes: inlining all methods invoked in c all at once (as in Chapter 4), 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.

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.

5.2.2 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 after the abstraction, the abstract vector representations for the code before and after it is refactored are identical. Then, the problem of searching for refactorings can be reduced to the problem of finding code with the same abstract vector representation.

Each refactoring type defines a different abstraction operation on vectors 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 abstract vector.

The "Query" portion to the left of Figure 5.2 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 abstract base vector ${}^{\gamma}\psi_q$; and its inlined vector v_q^I can become an abstract 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 for example if method q does not invoke any methods or does not contain any of the vector features that are affected by the abstraction.

In this chapter, 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:

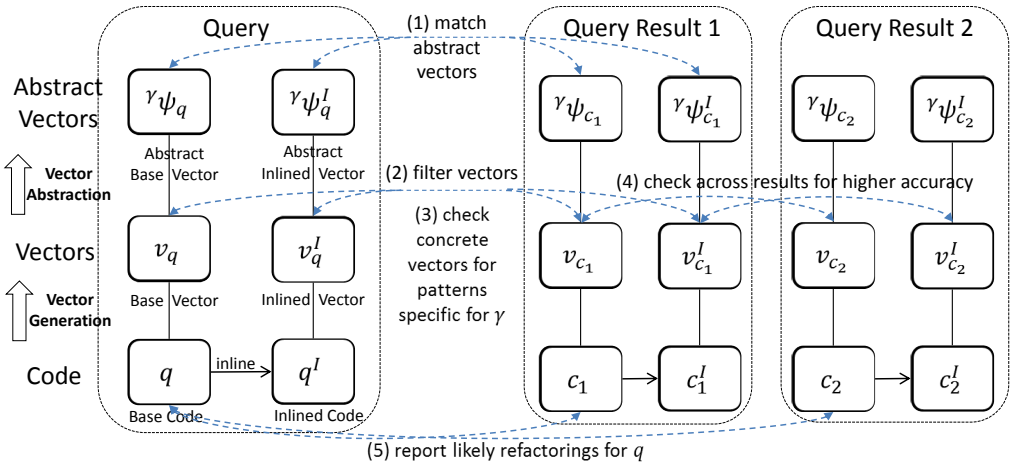


Figure 5.2: Vector Abstraction and Concretization Steps

Definition 5.1 *(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_i*.*

Definition 5.2 *(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. $0 \leq m \leq K$*;*
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 operation $(v^{\delta}, m, \mathcal{D})$ encodes 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 respectively, 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 abstract 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 vectors representing both 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 $\mathcal{D}_s = \{f_{d_1} \mapsto n_{d_1}, f_{d_2} \mapsto n_{d_2}, \ldots, f_{d_s} \mapsto n_{d_s}\}$ where $2 \leq s \leq K$ and $1 \leq d_1 \leq d_2 \ldots \leq d_{s-1} \leq d_s \leq K$, such that $\Sigma_{i=1}^{s} n_{d_i} = 0$,[8] then we consider the features in $\mathcal{D}_s$ as inter-exchangeable and we merge all their counts in v 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 $\mathcal{D}_s = \{f_{d_1} \mapsto n_{d_1}, f_{d_2} \mapsto n_{d_2}, \ldots, f_{d_s} \mapsto n_{d_s}\}$ where $2 \leq s \leq K$ 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 conclude that the features in $\mathcal{D}_s$ should be changed together in the same way by γ and we set all their counts to 1 as follows:

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

[8] $\forall (f_{d_i} \mapsto n_{d_i}) \in \mathcal{D}, n_{d_i} \neq 0$ because of rule III in DEF. 5.2.

This rule 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 $\mathcal{D}$ matching the above rules; if these subsets are pairwise disjoint, we perform the abstraction for each subset separately; if they overlap, we manually identify a subset to abstract. In our experiments, since vector differences $\mathcal{D}$ are generated from sample refactored code in classical collections (see Section 5.3) and are of small sizes, it has been relatively easy to find suitable subsets such 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 rule (II) or (III), and we find that 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 and based on our understanding of the code changes involved by the refactorings, we may remove some features or add new ones.

For example, we can define the vector abstraction for the kind of refactoring operation in Example 20. 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 5.1 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 5.2 shows the abstract vectors if the abstraction rules described above are applied to the concrete vectors in Table 5.1.

		Features									
row ID	Code	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 5.2: Sample Abstract Vectors Corresponding to Vectors in Table 5.1

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 vector differences.

5.2.3 Vector-Based Query

When we want to find instances of a refactoring operation type γ in a large code base, we apply the abstraction for γ to all vectors generated from the code base. This causes the code difference induced by γ to be eliminated, and the abstract vectors of either refactored or non-refactored to be identical. Consequently the abstract vectors corresponding to refactored code can be clustered together and the abstract vectors corresponding to non-refactored code can be used to *query* only against the corresponding cluster of

refactored code. Specifically, we can use Locality-Sensitive Hashing (LSH) [24, 44] to efficiently handle nearest-neighbor *queries* on our high-dimensional vectors and thus to find the cluster of vectors that is matching a query [41, 57]. Afer matching, the comparisons between a query and the vectors in its corresponding cluster can be more complex since the number of pair-wise comparisons is much smaller than the total number of pair-wise comparisons that would be needed for all the vectors.

Different from the vector query presented in Section 4.4.5, that returns pairs of vectors, the results returned by the methodology of this chapter are pairs comprising of a vector q, also called query or candidate that can be refactored via a refactoring operation γ, and a cluster of possibly many vectors, that represent examples of how q could be refactored. Furthermore, as shown by step 5 in Figure 5.2 we check the query results within a cluster against one another to determine if the query can be refactored in a unique way. For instance if the returned vectors for one query q suggest both refactoring via introducing two explaining variables and refactoring via introducing one explaining variable, thus the way to refactor q is not unique, then we exclude q and the example vectors from the results. As illustrated in step (1) in Figure 5.2, we perform queries on abstract vectors of either base or inlined code, or both (either ψ_q or ψ_q^I or both). Depending on the vectors used, we can identify candidates in both the *context* of historical refactoring and the context of refactoring opportunities:

(1) when we use every abstract vector generated from one version of code as a query to search for matching abstract vectors from a version of the code, we can detect historical refactorings that happened between two versions of the code.

(2) when we use every abstract vector generated from a code base as a query to search for matching abstract vectors in the same code base, we can detect refactoring opportunities that may be available among code fragments in the current code base.

The types of refactorings under investigation would affect whether the corresponding

queries and matching results are drawn from either base vectors or inlined vectors or both. For example, for the detection of the opportunity of method inlining shown in Example 20(a), we use the abstract *base* vector of (a) as a query and search for its matches among the abstract vectors for all other *inlined* code.

Not all matched vectors can be subject to refactoring; we apply heuristic filters (Figure 5.2, 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 inlined and baseline query and result code, if $v_q^I = v_{c_1}^I$ and $v_q = v_{c_1}$, then q and c_1 are very likely the same syntactically and do not exhibit structural differences related to *Extract/Inline Method* refactoring. c_1 is simply a clone of q and thus the pair (q , c_1) does not indicate how to refactor q, and 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 Example 20, (a) and (c) inlined with (b) can be detected as likely refactorings since their abstract vectors (rows 1 and 4 in Table 5.2) are identical.

5.2.4 Vector Concretization

After the above steps, we have a set of filtered query results for each piece of code used as query. The following concretization phase performs several kinds of checks on the concrete vectors corresponding to the query and its results to improve the precision of refactoring detection. This phase corresponds to steps (3) and (4) in Figure 5.2.

The first kind of checks is to make sure the *differences* among the concrete vectors indeed subsume the differences ($\mathcal{D}$, see Section 5.2.2) that are induced by a kind of refactoring operation γ. This is useful for reducing false positives since different refactoring operations may in fact change same features in code and having similar abstract 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 5.2.2) 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 5.3), 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 by 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 that many instances;

  (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 neither q^I or c_i^I does not contain that many instances;

(II) Check v_q^I and $v_{c_i}^I$ against $(v^\delta, m, \mathcal{D})$, and remove the query result c_i if the following condition is true:

  (1) if $\exists i \in 1..K, s.t.\ (v_q^I[i] < v^\delta[i]) \vee (v_{c_i}^I[i] < v^\delta[i])$, it means γ would need to be carried out in a *context* containing $v^\delta[i]$ instances of the code feature f_i but neither q or c_i does not contain that many instances;

(III) Check all base and inlined vectors against code change rules specific to γ so as to remove possibly more negative query results [9];

[9] These code rules are based on our understanding of the code changes involved in the refactorings. For a refactoring operation that reverses a conditional, for example, such a rule would involve checking that an if condition is present.

(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 Example 20, one of the differences among their concrete vectors (Table 5.1, 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.

5.3 Refactoring as Vector Abstraction and 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 work 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 can be perceived as tailoring for detecting those 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 captured by the existing 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 has been 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. In future, we intend to enhance the capabilities of vectors to

encode changes made by such refactoring operations.

In this work, the types of refactorings we can detect are mostly defined by classical collections [38, 56]. Table 5.3 lists sample abstraction and concretization operations developed by our approach for the types of refactoring operations that we can detect. The sample refactored code we use for defining abstraction and concretization are taken from the classical collections [38, 56]. In the table, abstraction rules that do not change the values for a feature are not shown (e.g., $\gamma\psi[i] = v[i]$, if the vector difference $\mathcal{D}$ does not contain a mapping for f_i.); some concretization rules that are the same for all types of refactorings, as described in various subsections of Section 5.2 are also omitted. In addition to the notations used in Section 5.2, 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(e.g, *constant* denotes bytecode instructions *iload_0, iload_1*, etc.). We rely on the feature names to convey their meaning. The other refactoring operations that we have experimented with in this chapter and are not defined in the table are the following: *Remove Assignment to Initialization, Consolidate Conditional, Decompose Conditional, Replace Nested Conditional with Guard Clauses, Introduce Parameter Object, Replace Parameter with Method, Hide Delegate, Remove Middleman, Replace Method with Method Object, Replace Temporary with Query*.

5.4 Empirical Evaluation

This section presents the evaluation of *vector abstraction and concretization* on four aspects. They are the effectiveness of our methodology in detecting historical refactoring, the effectiveness in detecting refactoring opportunities, the accuracy in identifying

#	Refactoring	Abstractions ($^{Y}A(v)$ or simply ψ)	Concretization Checks	Descriptions
1.	**Extract Method**	ψ[load] = ψ[store] = 0 ψ[constant] = 0	∄ 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	∄ 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]+∑v[load_i] ψ[store] = v[store]+∑v[store_i] ψ[load_i] = ψ[store_i] = 0 i ∈ { 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	∃ v_i inlined into v_c s.t. (∑ v_i[f] - ∑ v_q[f]) == (v_i[getfield] + v_i[putfield]), f ∈ {0,.. v_q.length} (v_i[getfield] - v_q[getfield] > 0) (v_c[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^1[getfield] > 0 // ∃ an extra field to encapsulate ∃ 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.	∑ v_q^S[literal] - ∑ v_c^S[literal] > 0 // query has more magic numbers literal ∈ { 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.	∃ v_i inlined into v_c s.t. ∑ v_i^S[op] == 1, op ∈ { 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] + ∑ v[opp] ψ[neq] = ψ[opp] = 0 opp ∈ { gt, ge, le }	∃ cond ∈ { eq, neq} or opp ∈ { 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

Table 5.3: Sample Abstraction and Concretization Operations for Refactoring

refactoring, and the scalability of our methodology to large software systems.

5.4.1 Setup and Infrastructure

We have implemented the algorithms presented in Section 5.2 in a tool for Java programs and 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 aimed to explore the effectiveness and accuracy of our system in detecting both historical refactoring and refactoring opportunities. This study analyzed three Java programs from the Software Infrastructure Repository (SIR): JMeter, XMLSecurity, and Apache Ant. We performed experiments on 6 versions (0 to 5) of JMeter, 6 versions of Apache Ant, and 4 versions of XMLSecurity. The

size of these subject programs ranges from 17KLOC to 80KLOC. The projects were selected according to the criteria that they were discussed in related works which allows the comparison of our prototype with the state-of-the-art technique in detecting historical refactorings—RefFinder [99], 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. We 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 each one of them. For refactoring opportunities detected for the three subject programs in the first case study, we chose to inspect the results for the first version of each program only. Each result inspector was required to verify if 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.

5.4.2 Results

The results of the experiments performed in the first case study are shown in tables 5.4, 5.5, and 5.6. 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

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

Table 5.4: Jmeter Results Summary

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 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,[10] the students inspected all 276 for the first version of each subject program

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

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

Table 5.5: Ant Results Summary

and identified 35 false positives, giving a precision for our approach of *87% in detecting refactoring opportunities*.

We inspected all of the 191 historical refactorings detected. This validation was performed independently 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 opportunities detected evolve from

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

Table 5.6: XMLSecurity Results Summary

one version to another in a non-monotonic 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 versions 2 and 3 for refactoring *Downcast Encapsulate* (Row # 6 and Columns # 2, 2-3, and 3 in Table 5.4). In versions 0, 1, and 2 of JMeter, a large number of methods invoke method `getProperty` from class `Task` and downcast their result to obtain a string. Another category of methods invoke method `getPropertyAsString` from class `Task` which has the downcast inside the method. The similarity between the methods that invoke `getProperty` and those that invoke `getPropertyAsString` resulted in a number of refactoring opportunities which were detected by our approach. Between versions 2 and 3, some of these opportunities were applied and the methods that invoke `getProperty` were changed to invoke `getPropertyAsString`. These cases were captured by our approach by comparing

	Validated results	Accuracy
Refactoring Opportunities	276	87%
Refactoring Historical	191	92.6%

Table 5.7: Accuracy in Detecting Refactoring

the two versions, but were not detected by RefFinder.

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 5.8 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.

These 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 took 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 [98, 99]. Lastly, we note that *Consolidate Conditional* and *Replace nested conditionals* each took about 59 minutes to complete. A plausible reason is that the abstract 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.

Refactoring Type	Number 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 with Symbolic Constant	1577	24m44
Consolidate Conditional	52	59m3
Replace Nested Conditional with Guard Clauses	60	59m29
Introduce Parameter Object	325	18m57
Replace Parameter with Method	228	18m57
Hide Delegate	13	19m44
Remove Middleman	10	19m14
Replace Method with Method Object	2	18m8

Table 5.8: Eclipse Results Summary

5.4.3 Analysis

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 [25, 48, 70, 98] 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 vector abstraction and concretization, on the other hand, can accurately encode and detect this small change and can thus detect a large number of *Self Encapsulate Field* refactorings with 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 [98] – the state-of-the-art historical refactoring approach– we note that RefFinder has a different definition for *Self Encapsulate Field*. Its definition, which is shown below, is based on the changes occurring between two versions of a program. In particular it focuses on the creation of a getter method (e.g. $added\_getter$ and $added\_fieldmodifier$) and does not capture the manifestation of the field encapsulation on the methods that, before refactoring, accessed a field directly and, after refactoring, will access a field through a `getter` method. This makes comparison between RefFinder and our approach impractical.

RefFinder Definition of Self Encapsulate Field

$encapsulate\_field(fFullName) \wedge \nexists\, access\, to\, the\, field\, besides\, the\, new\, getter\, and\, setter$
$\rightarrow self\_encapsulate\_field$

$deleted\_fieldmodifier(fFullName, public) \wedge$
$added\_fieldmodifier(fFullName, private) \wedge$
$added\_getter(mGetFullName, fFullName) \wedge$
$added\_setter(mSetFullName, fFullName)$
$\rightarrow encapsulate\_field$

Moreover, we note that RefFinder cannot be applied within the same version, thus is unable to discover any refactoring opportunity that occurs 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. Example 21 illustrates such a refactoring. 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 between the two versions of vectors for the caller (`getPrice`) has the same absolute value and a different sign as compared to the difference in the numbers of method calls between the two versions of vectors for the callee (`discountedPrice`).

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 [98]. 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 is faster than [98] based on the results reported in [98] for projects of similar size to the ones in our case study. [98] 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 the time spent on Eclipse versus that spent on 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.

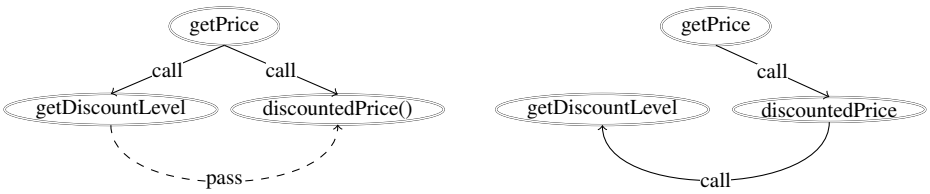
5.4.4 Discussion and Threats to Validity

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

Example 21. *(Replace Parameter with Method Refactoring)*

```
1 public double getPrice() {
2  int basePrice = _quantity * _itemPrice;
3  int discountLevel = getDiscountLevel();
4  double finalPrice = discountedPrice (basePrice, discountLevel);
5  return finalPrice;
6 }
7
8 private double discountedPrice (int basePrice, int discountLevel) {
9  if (discountLevel == 2)
10   return basePrice * 0.1;
11    else return basePrice * 0.05;
12 }
```

```
1 public double getPrice() {
2  int basePrice = _quantity * _itemPrice;
3  double finalPrice = discountedPrice (basePrice);
4  return finalPrice;
5 }
6
7 private double discountedPrice (int basePrice) {
8  if (getDiscountLevel() == 2)
9   return basePrice * 0.1;
10  else return basePrice * 0.05;
11 }
```



elements, the specific names of identifiers), they have been shown to be effective for code clone detection [41, 57], 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 [38, 56], 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 [18, 83, 84] and apply detected refactoring opportunities automatically.

5.5 Chapter Summary

This chapter 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 abstract vectors. Thus, our approach reduces the problem of detecting refactoring 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 (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 about 7 hours. We have also applied our prototype to 16 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 92% precision. Our prototype also detects more than 2.8K instances of refactoring opportunities within individual versions of the programs.

Chapter 6

Future Work

In the previous chapters we have shown that *efficient and effective detection of code fragments related by refactoring operations within large software can be achieved by performing abstraction and algebraic operations over high-dimensional vectors representing the code*. Specifically, in Chapter 4 we introduced *vector inlining* and showed that the detection of refactoring operations involving cross-function changes can be accurately achieved by *vector addition* and *vector query*. In Chapter 5 we further refined the approach and showed that the precise code change patterns induced by refactoring can be learned from examples, using a semi-automatic approach, and then detected by employing a flexible approach based on *abstractions* and *concretizations*. In the rest of this chapter we will outline several possible directions of future research.

6.1 Extensive Detection of Refactoring or Code Edits

One possible avenue of research involves extending the list of refactorings that we can detect and discovering the relations between the abstractions and concretizations of all refactoring operations.

Currently the detection of a refactoring operation is based on algebraic operations

over high-dimensional vectors representing the code. Classifying certain refactoring operations may however require us to incorporate more diverse metadata about the vectors, or information about other detected refactoring operations. For example, the new metadata criteria might include the positions in the class hierarchy of the classes enclosing the code fragments, the relation between such classes (extend relation), call graph information about the particular call relations between code fragments, or some other characteristics of code fragment composition. By introducing appropriate abstraction and concretization criteria we aim to detect a broad range of refactorings.

One such example of criteria that can be used to detect refactoring is illustrated by Example 22. Specifically, the example shows how clustering individual refactorings can provide definitions for complex refactorings such as *Parameterize method*.

Example 22. *(Parameterize Method)*

```
org.apache.jmeter.gui.action.Load$Test
public void testFile4() throws Exception {
 HashTree tree = getTree(testFile4);
 //loader.updateTree(tree);
 assertTrue(tree.getArray()[0] instanceof org.apache
      .jmeter.testelement.TestPlan);
 loader.convertTree(tree);
 assertTrue(tree.getArray()[0] instanceof org.apache
      .jmeter.control.gui.TestPlanGui);
}
```

```
org.apache.jmeter.gui.action.Load$Test
public void testFile5() throws Exception {
 HashTree tree = getTree(testFile5);
 //loader.updateTree(tree);
 assertTrue(tree.getArray()[0] instanceof org.apache
      .jmeter.testelement.TestPlan);
 loader.convertTree(tree);
 assertTrue(tree.getArray()[0] instanceof org.apache
      .jmeter.control.gui.TestPlanGui);
}
```

```
public void testFile12() throws Exception {
 ...
}
```

```
org.apache.jmeter.gui.action.Load$Test
public void testFile4() throws Exception {
 loadAndAssertTree(getTree(testFile4));
}

private void loadAndAssertTree(HashTree tree)
  throws Exception {
 assertTrue(tree.getArray()[0] instanceof org.apache
      .jmeter.testelement.TestPlan);
 loader.convertTree(tree);
 assertTrue(tree.getArray()[0] instanceof org.apache
      .jmeter.control.gui.TestPlanGui);
}
```

```
org.apache.jmeter.gui.action.Load$Test
public void testFileN(TestFile testFile)
  throws Exception {
 loadAndAssertTree(getTree(testFile));
}

private void loadAndAssertTree(HashTree tree)
  throws Exception {
 assertTrue(tree.getArray()[0] instanceof org.apache
      .jmeter.testelement.TestPlan);
 loader.convertTree(tree);
 assertTrue(tree.getArray()[0] instanceof org.apache
      .jmeter.control.gui.TestPlanGui);
}
```

In a *Parameterize Method* refactoring, *several* methods that do similar things but with different values, are transformed by creating *one* method that uses a parameter for

the different values. The code fragments in Example 22 are taken from Apache Ant. As can be seen from the example, the methods to the left, `testFile4`, `testFile5` up until `testFile12`, are very similar except for their use of different values: field `testFile4` for method `testFile4`, field `testFile5` for method `testFile5`, and so on. On the other hand, method `testFileN` takes a parameter `testFile` which holds the value to be used in the body of the method. When called with appropriate values for its parameter, `testFileN` will provide the exact same functionality as `testFile4`, `testFile5` up until `testFile12` thus asserting a *Parameterize Method* refactoring. However our current approach will only detect *individual Extract Method* refactorings.

As the *Parameterize Method* refactoring operation involves changes to *several* methods, its detection should thus be based on first detecting *several* individual refactorings (9 Extract Method in this case) and then comparing the detected results to check whether "*several* methods that do similar things but with different values, are refactored into *one* method". This can be achieved by first detecting *individual refactorings* and then clustering the detected refactorings based on similarity between the vectors used as query and by classifying clusters of results as *Parameterize Method*.

We also plan to research the detection of class level refactorings. An example of class level refactoring would be *Form Template Method* which is applied when there exist two methods in subclasses that perform similar steps in the same order. The application of *Form Template Method* will extract the steps into methods with the same signature, so that the original methods become the same, and will then pull the newly extracted methods into the superclass. Other examples would be *Replace Inheritance with Delegation, Replace Conditional with Polymorphism*, etc. Detecting this type of refactorings may require creation of vectors for the entire class by removing the boundaries of encapsulation. Comparing class level vectors will then result in detecting class level refactorings.

6.2 Composite Refactorings and Query Language

The second direction that we plan to investigate involves composite refactorings. Currently our approach can detect individual refactorings. However it might be the case that multiple refactoring operations are applied to a code fragment. Also, detecting historical refactoring between non-consecutive versions may benefit from the detection of composite refactoring.

A composite refactoring opportunity is shown in Example 23. Methods `getControllers` in (a) and `getVisualizer` in (c) have the same behavior yet different syntactic shapes. `getControllers` differs from `getVisualizer` by introducing a number of extra temporary variable declarations and by performing all functions in one method as opposed to `getVisualizer` that delegates some of its functionality in the extracted method `getVector`. As the *Extract Method* refactoring may often be performed together with *Inline Temporary* or *Introduce Variable* refactorings, our approach can detect this type of composite refactoring by applying the abstractions for both *Extract Method* and *Inline Temp/Introduce Variable* when querying for refactorings of type extract method.

The general idea to detect the composition of two refactoring operations γ_1 and γ_2 would be to either use the current approach and learn from examples depicting combinations of refactorings, or to "compose" the abstractions and "compose" the concretizations. Next we describe these two approaches.

The former approach would start with two vectors v_1 and v_2 corresponding to code fragments c_1 and c_2, where c_2 is obtained from c_1 after applying γ_1 and γ_2. Next it would construct the vector difference $\delta(v_1, v_2) = (v^\delta, m, \mathcal{D})$, which encodes both "common" parts (in v^δ) and differences (in $\mathcal{D}$) between the two vectors, and would generate the abstraction and the abstract vectors ${}^{\gamma_1 \circ \gamma_2}\mathcal{A}(v)$ or ${}^{\gamma_1 \circ \gamma_2}\psi$ and the concretization ${}^{\gamma_1 \circ \gamma_2}\mathcal{C}(v_1, v_2)$. Although this approach is in theory simple it does not scale as it does

Example 23. *(Composite Refactorings)*

```
1  public static Vector getControllers(Properties properties){
2   String name = "controller.";
3   Vector v = new Vector();
4   Enumeration names = properties.keys();
5   while (names.hasMoreElements()){
6    String prop = (String) names.nextElement();
7    if (prop.startsWith(name)){
8     Object o = instantiate(
9      properties.getProperty(prop),
10     "org.apache.jmeter.control.SamplerController");
11    v.addElement(o);
12   }
13  }
14  return v;
15 }
```

(a)

```
1  public static Vector getVector(Properties properties, String name) {
2   Vector v = new Vector();
3   Enumeration names = properties.keys();
4   while (names.hasMoreElements()) {
5    String prop = (String) names.nextElement();
6    if (prop.startsWith(name)) {
7     v.addElement(properties.getProperty(prop));
8    }
9   }
10  return v;
11 }
12 ...
13 public static Vector getVisualizer(Properties properties) {
14  return instantiate(
15    getVector(properties, "visualizer."),
16    "org.apache.jmeter.visualizers.Visualizer");
17 }
```

(b)

(c)

not reuse the definitions of abstractions and concretizations for individual refactorings and requires *new examples* to be constructed for all combinations of refactorings.

The latter approach on the other hand would reuse the models constructed for individual refactorings. It would start with the vector difference $\delta_{\gamma_1}(v_1, v_2) = (v^{\delta}_{\gamma_1}, m_{\gamma_1}, \mathcal{D}_{\gamma_1})$ learned from the example corresponding to γ_1 and the vector difference $\delta_{\gamma_2}(v_1, v_2) = (v^{\delta}_{\gamma_2}, m_{\gamma_2}, \mathcal{D}_{\gamma_2})$ learned from the example corresponding to γ_2 and the abstractions ${}^{\gamma_1}\mathcal{A}(v)$ and ${}^{\gamma_2}\mathcal{A}(v)$ and would generate the abstraction ${}^{\gamma_1}\mathcal{A}(v) \circ {}^{\gamma_2}\mathcal{A}(v)$ or ${}^{\gamma_1}\psi \circ {}^{\gamma_2}\psi$ and

the concretization ${}^{\gamma_1}\mathcal{C}(v_1, v_2) \circ^{\gamma_2} \mathcal{C}(v_1, v_2)$.

There are however a number of challenges with either approach. If the feature sets $\mathcal{D}_{\gamma_1}$ and $\mathcal{D}_{\gamma_2}$, which indicate the features that may be changed by γ_1 and γ_2, have no elements in common, the model for detecting $\gamma_1 \circ \gamma_2$ can be easily defined by ${}^{\gamma_1}\mathcal{A}(v) \circ^{\gamma_2} \mathcal{A}(v) = {}^{\gamma_1}\mathcal{A}(v) \wedge^{\gamma_2} \mathcal{A}(v)$ and ${}^{\gamma_1}\mathcal{C}(v_1, v_2) \circ^{\gamma_2} \mathcal{C}(v_1, v_2) = {}^{\gamma_1}\mathcal{C}(v_1, v_2) \wedge^{\gamma_2} \mathcal{C}(v_1, v_2)$. However if the refactorings γ_1 and γ_2 change the same elements, detecting the composition becomes more difficult. For example if γ_1 were *Introduce one Explaining Variable* and γ_2 were *Inline Temporary Variable* the composite refactoring will have no detectable effect on the features of the vectors as they are opposite and one invalidates the effect of the other. Detecting such composite refactorings is thus future work.

We also plan to develop a query language that allows us to specify the refactoring opportunities to search for. The query language will also allow us to specify the composition of multiple refactoring types based on vector compositions.

6.3 Inferring Vector Abstraction and Concretization

As another direction of future work, we plan to automate the definitions of abstraction and concretization operations with ideas and techniques from programming by examples [18, 83, 84] and apply detected refactoring opportunities automatically.

Our general idea to infer the definitions of abstraction and concretization operations is in spirit similar to that of Daikon [31]. In particular we aim to capture the code transformations induced by refactoring by a set of *logic templates*. The concretization and abstraction for a refactoring will be a set of *instances* of these templates. An example of such logic templates is given below:

Template for Inference of Refactoring: Direct Two Feature Correlation

> *Given two characteristic vectors, v and v', defined by a sequence of K unique features denoted by $[f_1, \ldots, f_K]$, and two vector features, $f_i, f_j, i < K, j < K$ a direct two feature correlation is defined as follows:*
>
> $v'[f_i] - v[f_i] == v'[f_j] - v[f_j]$

A *direct two feature correlation* template encodes those code transformation that involve changing two vector features in the same way. For instance the *Introduce Explaining Variable* refactoring, shown in Example 24, manifests itself by adding a new variable declaration and a new variable access. Thus $v'[var\_decl] - v[var\_decl] == v'[var\_access] - v[var\_access]$.

Example 24. *(Introduce Variable Structural Differences)*

```
1  void f(int a, int b)
2   int c = a + b;
3   ...
-4  if (... complex computation){
5    ...
6   }
7  }
```

```
1  void f(int a, int b)
2   int c = a + b;
3   ...
+4  int temp = ... complex computation;
+5  if (temp) {
6    ...
7   }
8  }
```

A variation of the *direct two feature correlation* template is the *indirect two feature correlation*. This logic template is meant to encode those code transformation that involve changing two vector features in opposite ways. For instance the *Reverse Conditional* refactoring, shown in Example 25, manifests itself by reversing the sense of a conditional and thus by removing one type conditional and adding another. Thus $v'[if\_less] - v[if\_less] == v'[if\_greater\_equal] - v[if\_greater\_equal]$.

Template for Inference of Refactoring: Indirect Two Feature Correlation

Given two characteristic vectors, v and v', defined by a sequence of K unique features denoted by $[f_1, \ldots, f_K]$, and two vector features, $f_i, f_j, i < K, j < K$ an indirect two feature correlation is defined as follows:

$v'[f_i] - v[f_i] == -(v'[f_j] - v[f_j])$

Example 25. *(Reverse Conditional Structural Differences)*

```
1  void f(int a, int b)
2   int c = a - b;
3   ...
4   if (c < 0){
5    ...
6   }
7  }
```

```
1  void f(int a, int b)
2   int c = a - b;
3   ...
4   if (c >= 0){
5    ...
6   }
7  }
```

There are however a number of challenges with this approach. The first challenge is related to generating a set of template instances that do not imply each other. For example if the concretization for a refactoring is defined by three template instances, out of which the first two are: 1) $v'[f_i]-v[f_i] == v'[f_j]-v[f_j]$ and 2) $v'[f_i]-v[f_i] == v'[f_k]-v[f_k]$ the third instance, $v'[f_j] - v[f_j] == v'[f_k] - v[f_k]$, is implied by the first two and not needed. Pruning the inference of such template instances, that are implied by the existing instances, is necessary.

The second challenge stems from the fact that one refactoring operation may change one feature out of a set of features. One idea to circumvent this challenge is to infer a general definition of refactoring by introducing *symbolic or high level features*. Each of these symbolic features essentially represent sets of vector features. An example of such a feature is var_decl which we briefly mentioned for the *Introduce Explaining Variable* refactoring. var_decl translates, in terms of bytecode vector features, to a set of vector

features: istore_1, istore_2 etc. Encoding such set of vector features and the changes in sets of feature is needed. One of abstraction rules, defined in Chapter 5 which we recall below, also makes use of sets of features.

Vector Abstraction Rule

Given the vector difference $\mathcal{D}$, if there is a subset of $\mathcal{D}$, denoted as $\mathcal{D}_s = \{f_{d_1} \mapsto n_{d_1}, f_{d_2} \mapsto n_{d_2}, \ldots, f_{d_s} \mapsto n_{d_s}\}$ where $2 \leq s \leq K$ and $1 \leq d_1 \leq d_2 \ldots \leq d_{s-1} \leq d_s \leq K$, such that $\Sigma_{i=1}^{s} n_{d_i} = 0$, then we consider the features in $\mathcal{D}_s$ as inter-exchangeable and we merge their counts in v all into a unique conceptual feature as follows:

- $^{\gamma}\psi[d_1] = \Sigma_{i=1}^{s} v[d_i]$;
- $\forall i \in 2..s, ^{\gamma}\psi[d_i] = 0$.

Other challenges are related to efficiently verifying the inferred template instances, to determining the number of examples to learn from for each refactoring operation, and to applying the inference to detect code transformations from a more broader scope such as bug fixes or other systematic code edits.

Chapter 7

Conclusion

In this thesis we have presented a novel approach to scalably detect refactoring. Our experiments validate our belief that efficient and effective detection of code fragments related by refactoring operations within large software can be achieved by performing abstraction and algebraic operations over high-dimensional vectors representing the code. The main technical contributions of our research thesis are as follows:

- We proposed a new technique called *vector inlining* to simulate the effect of method inlining, which enables scalable detection of cross-function refactoring;

- We designed a systematic way to represent essential code changes needed for various types of refactoring operations as *abstraction and concretization operations* on 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 abstract 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.

As future avenues of research we aim to investigate three interesting directions: extensive detection of refactoring, composite refactorings, and inference of abstractions and concretization. The former direction will involve new research in finding *novel abstractions and concretization criteria* that can detect a broad range of refactorings. The second direction will involve research into the features changed by *multiple refactorings* and their interference, and into techniques such as integer linear programming to determine whether the change pattern between two vectors matches the composite change pattern of each refactoring. Moreover, the latter direction aims to make our approach applicable to a class of code edits more broad than that of the class of edits induced by refactoring.

Bibliography

[1] Mohammad Alshayeb, Mohammad Alshayeb, and Sabri A. Mahmoud. Software refactoring at the function level using new adaptive k-nearest neighbor algorithm. In *Advances in Engineering Software, Volume 41 Issue 10*, pages 1160–1178, 2010. 26, 30

[2] Mohammad Alshayeb, Mohammad Alshayeb, and Sabri A. Mahmoud. Software refactoring at the class level using clustering techniques. In *Journal of Research and Practice in Information Technology, 43 (4)*, pages 285–306, 2011. 26

[3] Mohammad Alshayeb, Mohammad Alshayeb, and Sabri A. Mahmoud. Software refactoring at the package level using clustering techniques. In *Software IET, Volume 5 Issue 3*, pages 276–284, 2011. 26, 30

[4] Giuliano Antoniol, Gerardo Casazza, Massimiliano Di Penta, and Ettore Merlo. Modeling clones evolution through time series. In *Proceedings of the 2001 IEEE International Conference on Software Maintenance*, pages 273–280, Florence, Italy, November 6–10, 2001. IEEE Computer Society. 36

[5] Hamid Abdul Basit and Stan Jarzabek. Detecting higher-level similarity patterns in programs. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposyum on Foundations of Software Engineering (ESEC/FSE'05)*, pages 156–165, 2005. 36

[6] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea Lucia. Improving software modularization via automated analysis of latent topics and dependencies. In *Journal, ACM Transactions on Software Engineering and Methodologies, Volume 23 Issue 1, Article No. 4*, February 2014. 26

[7] Gabriele Bavota, Andrea Lucia, Andrian Marcus, and Rocco Oliveto. Automating extract class refactoring: an improved method and its evaluation. In *Journal of Empirical Software Engineering, Volume 19 Issue 6*, pages 1617–1664, December 2014. 26, 28, 29

[8] Gabriele Bavota, Andrea Lucia, Andrian Marcus, and Rocco Oliveto. Using structural and semantic measures to improve software modularization. In *Journal of Empirical Software Engineering, Volume 18 Issue 5*, pages 901–932, October 2013. 26

[9] Gabriele Bavota, Andrea Lucia, and Rocco Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. In *Journal of Systems and Software, Volume 84 Issue 3*, pages 397–414, 2011. 26

[10] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea Lucia. Methodbook: Recommending move method refactorings via relational topic models. In *Journal, IEEE Transactions on Software Engineering, Volume 40 Issue 7, Article No. 4*, pages 671–694, 2013. 26,

28, 29

[11] Gabriele Bavota, Rocco Oliveto, Andrea Lucia, Giuliano Antoniol, and Yann-Gael Gueheneuc. Playing with refactoring: Identifying extract class opportunities through game theory. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 1–5, 2010. 28, 29

[12] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS©: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, 2004. 6, 36

[13] K. Beck. *Extreme Programming Explained, Embrace Change*. Addison-Wesley Professional, 2000. iii

[14] L. A. Belady and M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976. iii

[15] Keith Cassell, Peter Andreae, and Lindsay Groves. A dual clustering approach to the extract class refactoring. In *Proceedings on the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE'11)*, 2011. 26, 30

[16] Richard Conway. *A Primer on Disciplined Programming using PL/I, PL/CS, and PL/CT*. Addison-Wesley Professional, 1978. 1

[17] James R. Cordy and Chanchal K. Roy. The NiCad clone detector. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220, 2011. 36

[18] Allen Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993. 109, 116

[19] Jehad Al Dallal. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. In *Journal Information and Software Technology archive, 54(10)*, pages 1125–1141, 2012. 26, 28, 29

[20] Jehad Al Dallal. Identifying refactoring opportunities in object-oriented code: A systematic literature review. In *Journal Information and Software Technology, Volume 58*, page 231249, 2015. 21

[21] Jehad Al Dallal and Lionel C. Briand. A precise method-method interaction-based cohesion metric for object-oriented classes. In *ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 21 Issue 2, Article No. 8*, March 2012. 26, 28, 29

[22] Yingnong Dang, Song Ge, Ray Huang, and Dongmei Zhang. Code clone detection experience at microsoft. In *IWSC*, pages 63–64, 2011. 36

[23] Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. XIAO: Tuning code clones at hands of engineers in practice. In *ACSAC*, pages 369–378, 2012. 8, 36, 82

[24] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *20th ACM Symposium on Computational Geometry (SoCG)*, pages 253–262, 2004. 45, 50, 93

[25] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 166–177, 2000. 8, 13, 33, 105

[26] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph E. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 404–428, 2006. 8, 33

[27] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In

Proceedings of the 29th International Conference on Software Engineering, 2007. 36

[28] Eclipse. Eclipse Foundation. Eclipse Java development tools (JDT). http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTNode.html. 68

[29] Eclipse Foundation. Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/. 66

[30] Armijn emel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 63–72, 2011. 36

[31] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming, Volume 69 Issue 1*, pages 35–45, 2007. 116

[32] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 467–468, 2007. 26, 28, 29

[33] Marios Fokaefs, Nikolaos Tsantalis, Alexander Chatzigeorgiou, and Jrg Sander. Decomposing object-oriented class modules using an agglomerative clustering technique. In *IEEE International Conference on Software Maintenance*, pages 93–101, 2009. 26, 30

[34] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: Identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1037–1039, 2011. 3, 26, 30, 43

[35] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. In *Journal of Systems and Software, Volume 85 Issue 10*, pages 2241–2260, 2012. 26, 30

[36] Francesca Arcelli Fontana, Marco Zanoni, Andrea Ranchetti, and Davide Ranchetti. Software clone detection and refactoring. *ISRN Software Engineering*, online open access, 2013. 5, 6, 38, 80

[37] Fowler. A refactoring catalog. 17

[38] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. 2, 7, 18, 31, 45, 46, 47, 73, 76, 98, 109

[39] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. LAMBDAFICATOR: From imperative to functional programming through automated refactoring. In *Proceedings of the 35th International Conference on Software Engineering*, pages 1287–1290, 2013. 36

[40] Robert M. Fuhrer, Markus Keller, and Adam Kiezun. Refactoring in the Eclipse JDT: Past, present, and future. In *Workshop on Refactoring Tools (WRT)*, pages 30–31, 2007. 2, 14

[41] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, pages 321–330, 2008. 8, 12, 37, 44, 49, 50, 51, 53, 63, 82, 83, 85, 93, 108

[42] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 175–190, 2010. 36

[43] Alejandra Garrido and Jose Meseguer. Formal specification and verification of java refactorings. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation(SCAM)*, pages 165–174, 2006. 35

[44] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999. 45, 50, 53, 63, 84, 93

[45] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005. 33

[46] William G. Griswold. Phd thesis: Program restructuring as an aid to software maintenance. *University of Washington*, 1991. 35

[47] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 317–328, 2011. 41

[48] Shinpei Hayashi, Yasuyuki Tsuda, and Motoshi Saeki. Detecting occurrences of refactoring with heuristic search. In *Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, pages 453–460, 2008. 8, 33, 105

[49] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Aries: Refactoring support environment based on code clone analysis. In *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications, Article No. 436-084*, pages 222–229, 2004. 3, 25, 26, 27

[50] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. ARIES: Refactoring support tool for code clone. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–4, 2005. 3, 6, 14, 25, 26, 27, 36, 38

[51] Yoshiki Higo, Shinji Kusumoto, and Katsuro 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. 3, 25, 26, 27, 43

[52] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, pages 53–62, 2012. 26, 31

[53] Benjamin Hummel, Elmar Jrgens, Lars Heinemann, and Michael Conradt. Index-based code clone detection: Incremental, distributed, scalable. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'10)*. IEEE, 2010. 39

[54] IBM T.J. Watson. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net. 66, 70

[55] Hiroshi Igaki, Shinji Kusumoto, Yoshiki Higo, and Shuhei Kimura. Move code refactoring with dynamic analysis. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 575–578, 2012. 26, 31

[56] JetBrains. Refactoring Source Code in IntelliJ IDEA 13. http://www.jetbrains.com/idea/webhelp/refactoring-source-code.html. 45, 74, 76, 98, 109

[57] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, 2007. 8, 36, 44, 49, 50, 51, 53, 63, 82, 83, 85, 93, 108

[58] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposyum on Foundations of Software Engineering (ESEC/FSE'07)*, 2007. 36

[59] Elmar Jürgens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones

matter? In *Proceedings of the 31st International Conference on Software Engineering*, pages 485–495, 2009. 5, 36, 80

[60] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions of Software Engineering, Volume 28 Issue 7*, pages 654–670, 2002. 8, 36, 82

[61] Tomoko Kanemitsu, Yoshiki Higo, and Shinji Kusumoto. A visualization method of program dependency graph for identifying extract method opportunity. In *Proceedings of the 4th Workshop on Refactoring Tools*, pages 8–14, 2011. 26, 30

[62] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. In *Journal Empirical Software Engineering, Volume 13 Issue 6*, pages 645–692, 2006. 5, 80

[63] Yu Kashima, Yasuhiro Hayase, Norihiro Yoshida, Yuki Manabe, and Katsuro Inoue. An investigation into the impact of software licenses on copy-and-paste reuse among oss projects. In *Proceedings of the 18th Working Conference on Reverse Engineering*, pages 28–32, 2011. 36

[64] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *Proceedings of the IEEE International Conference on Software Maintenance*, page 736, 2001. 26, 31

[65] H. Kegel and F. Steimann. Systematically refactoring inheritance to delegation in java. In *Proceedings of the 30th International Conference on Software Engineering*, page 431440, 2008. 3, 28

[66] Iman Keivanloo, Juergen Rilling, and Philippe Charland. Internet-scale real-time code clone search via multi-level indexing. In *Proceedings of 2011 18th Working Conference on Reverse Engineering (WCRE11)*, pages 23–27, 2011. xv, 39, 40

[67] Iman Keivanloo, Juergen Rilling, and Philippe Charland. Seclone - a hybrid approach to internet-scale real-time code clone search. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension (ICPC'11)*, pages 223–224, 2011. 39

[68] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. MeCC: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 301–310, 2011. 8, 82

[69] Miryung Kim. A field study of refactoring challenges and benefits. In *Proceedings of the 2012 International Symposyum on Foundatios of Software Engineering*, pages 1–5, 2012. 2, 3, 4

[70] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering*, pages 371–372, 2010. 8, 13, 33, 105

[71] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposyum on Foundations of Software Engineering (ESEC/FSE'05)*, 2005. 5, 36, 80

[72] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *The 27th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–169, 2000. 37

[73] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, 2001. 37

[74] Ralf Lämmel. Towards generic refactoring. In *ACM SIGPLAN workshop on Rule-based programming*, pages 15–28, 2002. 35

[75] Sukhee Lee, Gigon ae, Heung Seok Chae, Doo-Hwan Bae, and Yong Rae Kwon. Automated scheduling for clone-based refactoring using a competent ga. In *Software Practice and Experience, Volume 41 Issue 5*, pages 521–550, 2011. 26, 31

[76] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation (OSDI'04)*, pages 289–302, 2004. 36

[77] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881, 2006. 37

[78] Hui Liu, Zhiyi Ma, Weizhong Shao, and Zhendong Niu. Schedule of bad smell detection and resolution: A new way to save effort. *Journal IEEE Transactions on Software Engineering*, 38(1):220–235, 2012. 3, 31

[79] Hui Liu, Zhendong Niu, Zhiyi Ma, and Weizhong Shao. Identification of generalization refactoring opportunities. In *Automated Software Engineering, Volume 20 Issue 1*, pages 81–110, 2013. 3, 26, 27, 31

[80] Simone Livieri, Yoshiki Higo, Makoto Matsushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder. In *Proceedings of the 29th International Conference on Software Engineering*, pages 106–115, 2007. 36

[81] Rim Mahouachi, Marouane Kessentini, and Khaled Ghedira. A new design defects classification: Marrying detection and correction. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, pages 455–470, 2012. 26, 28, 29

[82] Hayden Melton and Ewan Tempero. The crss metric for package design quality. In *Proceedings of the 30th Australasian conference on Computer science, Volume 62*, pages 201–210, 2007. 26, 30

[83] Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: Locating and applying systematic edits by learning from examples. In *Proceedings of the 35th International Conference on Software Engineering*, pages 502–511, 2013. 14, 40, 41, 46, 109, 116

[84] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pages 187–195, 2013. 41, 109, 116

[85] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, pages 287–297, 2009. 3, 43

[86] Emerson R. Murphy-Hill. Scalable, expressive, and context-sensitive code smell display. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 771–772, 2008. 31

[87] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, pages 552–576, 2013. 3

[88] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, pages 69–79, 2012. 41

[89] Hoan A. Nguyen, Tung T. Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen.

Accurate and efficient structural characteristic feature extraction method for clone detection. In *International Conference on Fundamental Approaches to Software Engineering (FASE 2009)*, 2009. 36, 49, 51

[90] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*, pages 440–455, 2009. 44

[91] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, 2012. 8, 36, 82

[92] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposyum on Foundations of Software Engineering (ESEC/FSE'09)*, pages 383–392, 2009. 36, 41

[93] William F. Opdyke. Refactoring object-oriented frameworks, phd thesis. *University of Illinois at Urbana-Champaign*, 1992. 3, 18, 19, 35

[94] Wei-Feng Pan, Bo Jiang, and Bing Li. Refactoring software packages via community detection in complex software networks. In *International Journal of Automation and Computing, Volume 10 Issue 2*, pages 157–166, 2013. 26, 29

[95] Wei-Feng Pan, Bo Jiang, and Youyang Xu. Refactoring packages of object-oriented software using genetic algorithm based community detection technique. In *International Journal of Computer Applications in Technology, Volume 48 Issue 3*, pages 185–194, 2013. 26, 29

[96] Wei-Feng Pan, Jing Wang, and Mu-Chou Wang. Identifying the move method refactoring opportunities based on evolutionary algorithm. In *International Journal of Modelling, Identification and Control, Volume 18 Issue 2*, pages 182–189, 2013. 26, 29

[97] Weifeng Pan, Bing Li, Yutao Ma, Jing Liu, and Yeyi Qin. Class structure refactoring of object-oriented softwares using community detection in dependency networks. In *Journal, Frontiers of Computer Science in China, Volume 3 Issue 3*, pages 396–404, 2009. 26, 29

[98] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *Proceedings of the International Conference on Software Maintenance*, pages 1–10, 2010. 8, 33, 34, 104, 105, 106, 107

[99] Napol Rachatasumrit and Miryung Kim. An empirical investigation into the impact of refactoring on regression testing. In *Proceedings of the International Conference on Software Maintenance*, pages 357–366, 2012. 100, 104

[100] A. Ananda Rao and K. Narendar Reddy. Identifying clusters of concepts in a low cohesive class for extract class refactoring using metrics supplemented agglomerative clustering technique. In *International Journal of Computer Science Issues, Volume 8 Issue 5*, pages 185–194, 2011. 26, 30

[101] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. In *Theory and Practice of Object Systems, 3(4)*, 1997. 36

[102] Chanchal Kumar Roy and James R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC'08)*, pages 172–181, 2008. 8, 36, 82

[103] Filip Van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques from a refactor-

ing perspective. In *Proceedings of the 19th IEEE/ACM International Conference on Automated Software Engineering (ASE'04)*, pages 336–339, 2004. 5, 6, 38, 80

[104] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 117–128, 2009. 84

[105] Ripon K. Saha, Chanchal K. Roy, and Kevin A. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Proceedings of the International Conference on Software Maintenance*, pages 293–302, 2011. 36

[106] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente:. Recommending move method refactorings using dependency sets. In *Proceedings of IEEE 20th Working Conference on Reverse Engineering (WCRE)*, pages 232–241, 2013. 26, 28, 29

[107] Max Schäfer and Oege de Moor. Specifying and implementing refactorings. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 286–301, 2010. 13, 36

[108] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. Correct refactoring of concurrent java code. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, pages 225–249, 2010. 13

[109] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. Modernizing legacy systems: Software technologies, engineering process and business practices. In *Addison-Wesley Longman Publishing Co., Inc., Boston, MA,*, 2003. iii

[110] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916, 2006. 26, 31

[111] Gabriela Serban and István Gergely Czibula. Restructuring software systems using clustering. In *22nd International Symposium on Computer and Information Sciences*, pages 1–6, 2007. 26, 30

[112] Mati ShomraT and Yishai A. Feldman. Detecting refactored clones. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 502–526, 2013. 14, 37, 45

[113] Quinten David Soetens, Javier Perez, and Serge Demeyer. An initial investigation into change-based reconstruction of floss-refactorings. In *Proceedings of the International Conference on Software Maintenance*, pages 384–387, 2013. 8, 13, 33

[114] Chengnian Sun, Siau-Cheng Khoo, and Shao Jie Zhang. Graph-based detection of library api imitations. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pages 183–192. IEEE, 2011. 66

[115] Girish Suryanarayana. *Refactoring for Software Design Smells*. Morgan Kaufmann. 29

[116] Robert Tairas. Clone detection and refactoring. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 780–781, 2006. 6, 38

[117] Robert Tairas and Jeff Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. In *Information and Software Technology Volume 54 Issue 12*, pages 1297–1307, 2012. 26, 31

[118] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of API refactorings in libraries. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 377–380, 2007. 8, 13, 33

[119] Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *TOPLAS*, 33(3):9, 2011. 13

[120] Tom Tourwe and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *CSMR*, pages 91–100, March 2003. 3, 26, 31

[121] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pages 119–128, 2009. 3, 25, 26, 31, 43

[122] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. In *IEEE Transactions on Software Engineering Volume 35 Issue 3 May/June*, pages 347–367, 2009. 26, 31

[123] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of refactoring opportunities introducing polymorphism. In *Journal of Systems and Software, Volume 83 Issue 3*, pages 391–404, 2010. 26, 31

[124] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. In *Journal of Systems and Software, Volume 84 Issue 10*, pages 1757–1782, 2011. 25, 26, 31, 43

[125] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, pages 233–243, 2012. 3

[126] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: A scripting language for refactoring. In *Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, 2006. 35

[127] Peter Weisgerber and Stephan Diehl. Are refactorings less error-prone than other changes? In *Proceedings of the Working Conference on Mining Software Repositories*, pages 112–118, 2006. 4

[128] Peter Weissgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 231–240, 2006. 8, 13, 32

[129] Zhenchang Xing and Eleni Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, 2005. 2, 35

[130] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, 2006. 69

[131] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, 2006. 41

[132] Limei Yang, Hiu Liu, and Zhendong Niu. Identifying fragments to be extracted from long methods. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference*, pages 43–49, 2009. 26

[133] L. Zhao and J. Hayes. Predicting classes in need of refactoring: An application of static metrics. In *Proceedings of the 2nd International PROMISE Workshop*, 2006. 26, 28, 29

[134] Hao Zhong, Lu Zhang, and Hong Mei. Inferring specifications of object oriented apis from api source code. In *APSEC*, pages 221–228, 2008. 41

[135] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *Proceedings of the 24th IEEE/ACM International Conference on*

Automated Software Engineering (ASE'09), pages 307–318, 2009. 41

[136] Minhaz F. Zibran and Chanchal K. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *SCAM*, pages 105–114, 2011. 36

[137] Minhaz F. Zibran, Ripon K. Saha, Chanchal K. Roy, and Kevin A. Schneider. Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study. In *SAC*, pages 1123–1130, 2013. 36

Appendix A

Vector Features

Vector Index	Vector Feature
1	ANONYMOUS_CLASS_DECLARATION
2	ARRAY_ACCESS
3	ARRAY_CREATION
4	ARRAY_INITIALIZER
5	ARRAY_TYPE
6	ASSERT_STATEMENT
7	ASSIGNMENT
8	BLOCK
9	BOOLEAN_LITERAL
10	BREAK_STATEMENT
11	CAST_EXPRESSION
12	CATCH_CLAUSE
13	CHARACTER_LITERAL
14	CLASS_INSTANCE_CREATION
15	COMPILATION_UNIT
16	CONDITIONAL_EXPRESSION
17	CONSTRUCTOR_INVOCATION
18	CONTINUE_STATEMENT
19	DO_STATEMENT
20	EMPTY_STATEMENT
21	EXPRESSION_STATEMENT
22	FIELD_ACCES
23	FIELD_DECLARATION
24	FOR_STATEMENT
25	IF_STATEMENT
26	IMPORT_DECLARATION
27	INFIX_EXPRESSION
28	INITIALIZER
29	JAVADOC
30	LABELED_STATEMENT
31	METHOD_DECLARATION
32	METHOD_INVOCATION
33	NULL_LITERAL
34	NUMBER_LITERAL
35	PACKAGE_DECLARATION
36	PARENTHESIZED_EXPRESSION
37	POSTFIX_EXPRESSION
38	PREFIX_EXPRESSION
39	PRIMITIVE_TYPE
40	QUALIFIED_NAME
41	RETURN_STATEMENT
42	SIMPLE_NAME
43	SIMPLE_TYPE
44	SINGLE_VARIABLE_DECLARATION
45	STRING_LITERAL
46	SUPER_CONSTRUCTOR_INVOCATION
47	SUPER_FIELD_ACCESS
48	SUPER_METHOD_INVOCATION
49	SWITCH_CASE
50	SWITCH_STATEMENT
51	SYNCHRONIZED_STATEMENT
52	THIS_EXPRESSION

Figure A.1: Features of Java JDT AST Vectors – Part 1

Vector Index	Vector Feature
53	THROW_STATEMENT
54	TRY_STATEMENT
55	TYPE_DECLARATION
56	TYPE_DECLARATION_STATEMENT
57	TYPE_LITERAL
58	VARIABLE_DECLARATION_EXPRESSION
59	VARIABLE_DECLARATION_FRAGMENT
60	VARIABLE_DECLARATION_STATEMENT
61	WHILE_STATEMENT
62	INSTANCEOF_EXPRESSION
63	LINE_COMMENT
64	BLOCK_COMMENT
65	TAG_ELEMENT
66	TEXT_ELEMENT
67	MEMBER_REF
68	METHOD_REF
69	METHOD_REF_PARAMETER
70	ENHANCED_FOR_STATEMENT
71	ENUM_DECLARATION
72	ENUM_CONSTANT_DECLARATION
73	TYPE_PARAMETER
74	PARAMETERIZED_TYPE
75	QUALIFIED_TYPE
76	WILDCARD_TYPE
77	NORMAL_ANNOTATION
78	MARKER_ANNOTATION
79	SINGLE_MEMBER_ANNOTATION
80	MEMBER_VALUE_PAIR
81	ANNOTATION_TYPE_DECLARATION
82	ANNOTATION_TYPE_MEMBER_DECLARATION
83	MODIFIER
84	UNION_TYPE
85	METHOD_INVOCATION0
86	METHOD_INVOCATION1
87	METHOD_INVOCATION2
88	METHOD_INVOCATION3
89	METHOD_INVOCATION4
90	METHOD_INVOCATION5
91	METHOD_INVOCATION6
92	API_INVOCATION0
93	API_INVOCATION1
94	API_INVOCATION2
95	API_INVOCATION3
96	API_INVOCATION4
97	API_INVOCATION5
98	API_INVOCATION6

Figure A.2: Features of Java JDT AST Vectors – Part 2

Vector Index	Vector Feature	Feature Description
0	nop	perform no operation
1	aconst_null	push a *null* reference onto the stack
2	iconst_m1	load the int value -1 onto the stack
3 - 8	iconst_${n} $n in {0,1,2,3,4,5}	load the int value {0, 1,2,3,4,5} onto the stack
11 - 12	lconst_${n} $n in {0, 1}	load a constant on stack
13 - 15	fconst_${n} $n in {0,1,2}	load a constant on stack
14 - 15	dconst_${n} $n in {0,1}	load a constant on stack
16 - 17	bipush, sipush	push a byte/short onto the stack
18 - 20	ldc, ldc_w, ldc2_w	push a constant *#index* from a constant pool
21 - 25	${type}load $type in {‘i’,‘l’,‘f’,‘d’,‘a’}	load a *value* from a local variable *#index*
26-45	${type}load_${n} $n in {0,1,2,3} $type in {‘i’,‘l’,‘f’,‘d’,‘a’}	load a *value* from local variable {0,1,2,3}
46 - 53	${type}aload $type in {‘i’,‘l’,‘f’,‘d’,‘a’,‘b’,’c’,’s’}	load a value from an array
54 - 58	${type}store $type in {‘i’,‘l’,‘f’,‘d’,‘a’}	store a *value* from a local variable *#index*
59 - 78	${type}store_${n} $n in {0,1,2,3} $type in {‘i’,‘l’,‘f’,‘d’,‘a’}	store a *value* from local variable {0,1,2,3}
79 - 86	${type}astore $type in {‘i’,‘l’,‘f’,‘d’,‘a’, ‘b’,’c’,’s’}	store a value into an array
87 - 88	pop, pop2	discard the top value/two values from the stack
89 - 91	dup, dup_x1, dup_x2	duplicate the value on top of the stack
92 - 94	dup2, dup2_x1, dup2_x2 92	duplicate two values on top of the stack
95	swap	swaps two top words on the stack
96 - 119	${type}${op} ${type} in {i,l,f,d} op in {“add”,“sub”,“mul”,”div”,“rem”,“neg”}	Binary operations on integer, long, float, or double
120 - 123	ishl, lshl, ishr, lshr	shift left/right
124 - 125	iushr, lushr	logical shift right
126 - 131	iand, land, ior, lor, ixor, lxor	perform a bitwise operation on two integers
132	iinc	increment
133 – 147	i2l, i2f, i2d, l2i, l2f, l2d, f2i, f2l, f2d, d2i, d2l, d2f, i2b, i2c, i2s	type conversion
149 - 152	lcmp, fcmpl, fcmpg, dcmpl, dcmpg	compare two values
153 - 158	if${op} $op in {eq,ne,lt,ge,gt,le}	if *value* is eq/ne/lt/ge/gt/le than 0 than branch to instruction at *branchoffset*

Figure A.3: Features of Java Bytecode Vectors – Part 1

Vector Index	Vector Feature	Feature Description
159 - 164	if_icmp${op} $op in {eq,ne,lt,ge,gt,le}	if *condition* than branch to instruction at *branchoffset*
165 - 166	if_acmp${op} $op in {eq,ne}	if references are equal/neq, branch to instruction at *branchoffset*
167	goto	goes to another instruction
168	jsr	jump to subroutine at *branchoffset*
169	ret	continue execution from address taken from a local variable *#index*
170	tableswitch	continue execution from an address in the table at offset *index*
171	lookupswitch	a target address is looked up from a table using a key and execution continues from the instruction at that address
172 - 176	${type}return $type in {'i','l','f','d','a'}	return a value
177	return	return void
178 - 179	getstatic, putstatic	get/put a static field value
180 - 181	getfield, putfield	get/put a field value
182 - 185	invokevirtual, invokespecial, invokestatic, invokeinterface	invoke a method
186	xxxunusedxxx	
187	new	create new object
188	newarray	create new array
189	anewarray	create a new array of references
190	arraylength	get the length of an array
191	athrow	throws an error or exception
192	checkcast	checks whether an *objectref* is of a certain type
193	instanceof	determines if an object *objectref* is of a given type
194 - 195	monitorenter, monitorexit	enter/exit monitor for object
196	wide	execute *opcode*, where *opcode* is either iload, istore, ret, inc but assume the *index* is 16 bit
197	multianewarray	create a new array of *dimensions*
198 - 199	ifnull, ifnonnull	if *value* is null/nonnull, branch to instruction at *branchoffset*
200	goto_w	goes to another instruction
201	jsr_w	jump to subroutine at *branchoffset*
202	initinvoke	invoke constructor
203 - 209	clientinvoke${n} $n in {0,1,2,3,4,5}	invoke client defined method with n parameters
210 - 216	apiinvoke${n} $n in {0,1,2,3,4,5,6}	invoke api defined method with n parameters
217	methodinvoke_checkcast	checkcast applied to the return value from a client defined method
218	apiinvoke_checkcast	checkcast applied to the return value from a api defined method

Figure A.4: Features of Java Bytecode Vectors – Part 2

Appendix B

Examples

In this appendix we present examples of detected refactoring operations that were not covered by the previous chapters such as *Decompose Conditional, Self Encapsulate Field, Downcast Encapsulate, Reverse Conditional*.

Example 26. *(Decompose Conditional Refactoring)*

```
org.eclipse.team.internal.core.subscribers.
    SubscriberResourceCollector

private boolean isDescendantOfRoot(IResource resource,
    IResource[] roots)
{
 for (int i = 0; i < roots.length; i++) {
  IResource root = roots[i];
  if (root.getFullPath().
    isPrefixOf(resource.getFullPath())) {
   return true;
     }
 }
 return false;
}
```

```
org.eclipse.team.internal.core.subscribers.
    SyncInfoWorkingSetFilter

private boolean isIncluded(IResource resource) {
 // otherwise, if their is a parent of the resource
 // in the set, it is included
 for (int i = 0; i < resources.length; i++) {
  IResource setResource = resources[i];
  if (isParent(setResource, resource)) {
   return true;
  }
 }
 return false;
}


private boolean isParent(IResource parent, IResource
    child) {
 return (parent.getFullPath().
   isPrefixOf(child.getFullPath()));
}
```

Example 27. *(Self Encapsulate Field Refactoring)*

```
1 package org.eclipse.jdt.internal.compiler.codegen.FloatCache
2
3 public String toString() {
4     int max = this.elementSize;
5     StringBuffer buf = new StringBuffer();
6     buf.append("{"); //NON-NLS-1
7     for (int i = 0; i < max; ++i) {
8         if ((this.keyTable[i] != 0) || ((this.keyTable[i] == 0)
      && (this.valueTable[i] != 0))) {
9             buf.append(this.keyTable[i]).append("->").append(
     this.valueTable[i]); //NON-NLS-1
10        }
11        if (i < max) {
12            buf.append(", "); //NON-NLS-1
13        }
14    }
15    buf.append("}"); //NON-NLS-1
16    return buf.toString();
17 }
```

```
1 package org.eclipse.jdt.internal.compiler.codegen.FloatCache
2
3 public String toString() {
4     int max = size();
5     StringBuffer buf = new StringBuffer();
6     buf.append("{"); //NON-NLS-1
7     for (int i = 0; i < max; ++i) {
8         if ((this.keyTable[i] != 0) || ((this.keyTable[i] == 0)
      && (this.valueTable[i] != 0))) {
9             buf.append(this.keyTable[i]).append("->").append(
     this.valueTable[i]); //NON-NLS-1
10        }
11        if (i < max) {
12            buf.append(", "); //NON-NLS-1
13        }
14    }
15    buf.append("}"); //NON-NLS-1
16    return buf.toString();
17 }
18
19 public int size() {
20     return this.elementSize;
21 }
```

Example 28. *(Downcast Encapsulate Refactoring)*

```
AuthManager
public String getName()
{
 return (String)getProperty(
    TestElement.NAME);
}
```

```
CounterConfig
public String getVarName()
{
 return getPropertyAsString(
    VAR_NAME);
}
```

```
AuthManager
public String getName()
{
 return getPropertyAsString(TestElement.NAME);
}
```

Example 29. *(Reverse Conditional Refactoring)*

```
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;
}
```