EXTRACTION OF JAVA PROGRAM FINGERPRINTS FOR
SOFTWARE AUTHORSHIP IDENTIFICATION

By

HAIBIAO DING

Bachelor of Science
Central China Agricultural University
Wuhan, China
1989

Master of Science
Central China Agricultural University
Wuhan, China
1992

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
August 2002
EXTRACTION OF JAVA PROGRAM FINGERPRINTS FOR
SOFTWARE AUTHORSHIP IDENTIFICATION

Thesis Approved:

M. H. Samadzadeh
Thesis Adviser

J. Chandler

D. E. Heffern

Timothy J. Petri
Dean of the Graduate College
Computer programs belong to the authors who design, write, and test them. Authorship identification is concerned with determining the likelihood of a particular author having written some piece(s) of code, usually based on other code samples from the same programmer. Java is a popular object-oriented computer programming language. Programming fingerprints attempt to characterize the features that are unique to each programmer. This thesis was an investigation to identify a set of software metrics that could be used for authorship identification of Java programs. A program written in visual C++ was utilized to extract the metrics. Multivariate discriminant analyses with the statistical package SAS were used to evaluate the metrics for authorship identification.

The main objective of this study was to extract a set of software metrics of Java source code that could be used as fingerprints to identify the author of the Java code. For this purpose, a program was designed and implemented to extract metrics from the source code. The interface was developed using visual C++ with Microsoft Visual Studio 6.0. The contributions of the selected metrics to authorship identification were measured by a statistical process, canonical discriminant analysis, with the statistical software package SAS. Among the 56 extracted metrics, 48 metrics were identified as being contributive to authorship identification. The authorship of 62.6-67.2% of the Java programs could be correctly identified with the extracted metrics. The identification rate could be as high as
85.8%, with derived canonical variates. Moreover, layout metrics played a more important role in the authorship identification than the other metrics.
ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my major advisor Dr. Mansur H. Samadzadeh for his continuous guidance, motivation, and valuable instruction throughout the study.

My appreciation extends to Dr. George E. Hedrick and Dr. John P. Chandler for their advice and suggestions while serving on my committee. My appreciation also extends to Dr. William D. Warde for his suggestions on the statistics.

Moreover, I would like to thank Ms. Xianzhen Wang for her help on statistics, and Ms. Lingyan Li for providing data.

Finally, I would like to give my special appreciation to my wife, Huining, and my son, David, for their love and understanding during the study.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Importance and Possibility of Authorship Identification</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Authorship Analysis Is Different From Plagiarism Detection</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Objectives</td>
<td>5</td>
</tr>
<tr>
<td>II. LITERATURE REVIEW</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Authorship Analysis in Literature</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Authorship Analysis of Software</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1 Executable Code as Source</td>
<td>9</td>
</tr>
<tr>
<td>2.2.2 Program Code as Source</td>
<td>10</td>
</tr>
<tr>
<td>III. FEATURE ANALYSIS AND METRICS EXTRACTION</td>
<td>19</td>
</tr>
<tr>
<td>3.1 Feature Analysis of Programs Written in Java</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Data Source and Metrics</td>
<td>20</td>
</tr>
<tr>
<td>IV. DATA ANALYSIS, RESULTS, AND DISCUSSION</td>
<td>25</td>
</tr>
<tr>
<td>4.1 Manual Variable Selection for the Classification Analysis</td>
<td>25</td>
</tr>
<tr>
<td>4.2 Automatic Variable Selection for the Classification Analysis</td>
<td>27</td>
</tr>
<tr>
<td>4.3 Classification Analysis</td>
<td>28</td>
</tr>
<tr>
<td>4.4 Data Sets</td>
<td>29</td>
</tr>
<tr>
<td>4.5 Results and Discussion</td>
<td>30</td>
</tr>
<tr>
<td>V SUMMARY AND FUTURE WORK</td>
<td>43</td>
</tr>
<tr>
<td>5.1 Summary</td>
<td>43</td>
</tr>
<tr>
<td>5.2 Future Work</td>
<td>44</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>46</td>
</tr>
<tr>
<td>APPENDICES</td>
<td>50</td>
</tr>
<tr>
<td>APPENDIX A – GLOSSARY</td>
<td>51</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2.1</td>
<td>Programming layout metrics for C programs</td>
</tr>
<tr>
<td>2.2</td>
<td>Programming style metrics for C programs</td>
</tr>
<tr>
<td>2.3</td>
<td>Programming structure metrics for C programs</td>
</tr>
<tr>
<td>2.4</td>
<td>Metrics proposed by Gray and his colleagues for possible relationship to authorship</td>
</tr>
<tr>
<td>2.5</td>
<td>Measurements extracted by the IDENTIFIED system for authorship analysis of C++ source code</td>
</tr>
<tr>
<td>3.1</td>
<td>Programming layout metrics extracted from the source code of Java programs</td>
</tr>
<tr>
<td>3.2</td>
<td>Programming style metrics extracted from the source code of Java programs</td>
</tr>
<tr>
<td>3.3</td>
<td>Programming structure metrics extracted from the source code of Java programs</td>
</tr>
<tr>
<td>4.1</td>
<td>Derivation metrics sets</td>
</tr>
<tr>
<td>4.2</td>
<td>Four data sets of different data treatments</td>
</tr>
<tr>
<td>4.3</td>
<td>Selected metric sets for the four data treatments (for explanation/detail on the first column metrics, refer to Tables 3.1, 3.2, and 3.3)</td>
</tr>
<tr>
<td>4.4</td>
<td>Classification accuracies (%) with selected metrics and derived canonical variates (CV) (as in Table 4.3, also for details on the data sets, see Table 4.2)</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>41</td>
</tr>
</tbody>
</table>

4.1 Contributions of individual canonical variates to the classification. A, B, C, and D refer to the data sets as described in Table 4.2. CVi represents the i\textsuperscript{th} canonical variate. The numbers on y axis stand for the percentage changes of classification accuracy after adding a canonical variate into the discriminant analysis.

4.2 Canonical structures of the first five canonical variates (CV1, CV2, CV3, CV4, and CV5) for data set A. The canonical structure coefficients are correlation coefficients between the metrics and the canonical variates. The coefficients range from 1.0 to -1.0 and reveal how important each metric is. The higher the absolute value of the coefficient, the more important the metric is for the canonical variate [SAS 1990].

4.3 Canonical structures of the first five canonical variates (CV1, CV2, CV3, CV4, and CV5) for data set C. The canonical structure coefficients are correlation coefficients between the metrics and the canonical variates. The coefficients range from 1.0 to -1.0 and reveal how important each metric is. The higher the absolute value of the coefficient, the more important the metric is for the canonical variate [SAS 1990].
CHAPTER I

INTRODUCTION

1.1 Importance and Possibility of Authorship Identification

With the widespread use of computers, software authorship identification has become an issue of concern. Computer software is not only a kind of intellectual property whose copyright should be protected, but also a functional text that may bring about unexpected consequences on computer systems. In many situations, it may be necessary to identify the source of a piece of software. When a system is attacked and pieces of code as viruses or logic bombs are available, tracing the source of such code is of high interest. Other situations include resolution of authorship disputes, proof of authorship in court, and proof of code re-engineering [Krsul and Spafford 1996].

At least four areas benefit directly from the findings of research in authorship analysis. They are the legal community that can count on the evidence provided to support authorship claims, the academic sector that may use the evidence provided to support authorship claims of students, the industry that can identify the author of a previously unidentified piece of code, and the real-time intrusion detection systems that may be enhanced by including authorship information [Krsul and Spafford 1996] [Kilgour et al. 1997].
It is popular these days to identify a criminal, who has been charged with a crime, by DNA fingerprinting from blood, hair, etc. This is based on the belief and evidence that human DNA sequences are individual. Software is a piece of written text that can be compiled or interpreted to run on a computer. The question is: Is it possible to extract something like human DNA fingerprints from software to verify authorship of this kind of written text?

Authorship identification in literature is an ancient topic, but research in the area is still continuing. A typical example is to identify the author(s) of Shakespeare’s works [Elliot and Valenza 1991]. Why can the authors of the works of literature be identified based on the written texts? This is because of the belief that an author’s expressions, dependence on certain words or phases, the frequency of individual words, preference to use short or long sentences, and so on might be individually associated with the author’s education and personality [Mosteller and Wallace 1964] [Gray et al. 1997] [Oman and Cook 1991] [Spafford and Weeber 1993]. All these contribute to a profile of individual authorship characteristics [Oman and Cook 1991] which can be used to identify the author of written text. A similar hypothesis is also held for handwriting identification that people’s handwritings are as distinctly different from one another as their individual natures and as their own fingerprints [Cha and Srihari 2000].

Although computer program code is less flexible in format than literature works due to the requirements of compilers or interpreters, there is still ample room for programmers to develop their own programming styles [Gray et al. 1997]. The relationship between programs and programmers psychologically exists in the manner in which they approach the problem-solving process and the manner of programming to
which they are accustomed [Kilgour et al. 1997]. In this way, the task of software code authorship analysis is parallel to written text authorship analysis [Sallis 1994]. Thus, the ideas and methods used for traditional textual analysis and forensics can be transferred to software analysis [Kilgour et al. 1997].

A number of research efforts have been undertaken to examine the origins of computer code. However, most of them have been on plagiarism detection [Donaldson et al. 1981] [Whale 1990] [Prechelt et al. 2000].

1.2 Authorship Analysis Is Different from Plagiarism Detection

Plagiarism detection is a concept that can be easily mixed with authorship identification. Although they both examine the code text of computer programs and authorship verification can be used as proof for plagiarism detection of software in the academic community [Kilgour et al. 1997], authorship verification is markedly different from plagiarism detection. Plagiarism is a situation that may occur among software companies as well as in academic settings.

Software plagiarism has been defined as a general form of software theft: complete, partial, or modified replication of software without the permission of the original author [Moreaux 1991]. For example, to detect plagiarism in a computer science course, the students' assignments are compared to see if some are suspiciously similar. The extremely similar programs strongly suggest that one student's code may have been derived from another's [MacDonell et al. 1999]. In such cases, there is no need to refer back to the collected works of a programmer because the programs being compared are functionally equivalent [Sallis et al. 1996]. However, authorship identification is to assign
a piece of code to a programmer, according to how well the code matches the programming style profile established previously for the programmer.

The purposes of plagiarism detection and authorship identification are different. While plagiarism detection measures similarity of two pieces of code, authorship identification does not. Plagiarism detection cannot tell if different pieces of code were written by the same person, and authorship identification cannot determine how similar two programs are. Thus they may lead to conclusions that may seem controversial.

As Moreaux indicated [Moreaux 1991], plagiarized replications do not need to keep the programming style of the original code. For example, let us assume code PJ is a plagiarized version by programmer Sam of code J by programmer Bob. After copying from Bob, Sam changed some programming style of PJ to his own. In detail, say old comments were replaced with new ones, variables were renamed to what he was more comfortable with, indentation and bracket placements were altered to his favorites, and “while” loops were changed to “for” loops. When code PJ and code J are subjected to plagiarism detection, they will be suspiciously similar. However, for the purpose of authorship identification, they are from distinct authors.

Nevertheless, plagiarism detection is closely related with authorship identification, because they both examine the text of computer programs and use analogous methods. Authorship analysis may use metrics and methods usually utilized for plagiarism detection.
1.3 Objectives

An early goal of identifying program authorship was to determine software theft and to prevent plagiarism of Pascal programs [Oman and Cook 1989]. It was argued that using only software complexity metrics only was not adequate to define a relationship between programs and programmers.

To improve the accuracy of C program authorship identification, Krsul and Spafford [Krsul and Spafford 1996] employed a comprehensive set of measurements to extract programming style. They divided over fifty metrics into three categories: programming layout metrics, programming style metrics, and programming structure metrics. Programming layout metrics include such fragile metrics as comment placement, indentation, bracket placement, and while lines. These metrics can be easily altered by a code formatter and pretty printer. Also, the text editor used to compose the program can modify these metrics by changing the format to its default or to a preferred layout.

Programming style metrics are related to the code layout metrics, but are more difficult to change. Such metrics include variable length, comment length, naming preference, and preference of loop statements.

Programming structure metrics are assumed to be dependent on programming experience and the ability of the programmer. Example metrics in the category of style metrics are mean number of lines of code per method/function, data structure usage and preference, and the cyclomatic complexity number [McCabe 1976]. Although so many measurements were collected, many were eliminated and a smaller set remained for the
final analysis [Krsul and Spafford 1996]. It can be argued that the information hidden in the unselected measurements was ignored.

Other research groups have examined the authorship of computer programs written in C++ [Sallis et al. 1996] [Gray et al. 1997] [MacDonell et al. 1999]. A dictionary-based system called IDENTIFIED was developed to extract source code metrics for authorship analysis [Gray et al. 1998]. Satisfactory results were obtained for C++ programs using case-based reasoning, feed-forward neural network, and multiple discriminant analysis [MacDonell et al. 1999]. The concept of software forensics has also been introduced into program source code authorship analysis. Software forensics is an area of software science aimed at authorship analysis of computer source code. As stated by Sallis and his colleagues [Sallis et al. 1996], software forensics is also a super set of all metrics that can be used for authorship analysis.

However, little information about authorship identification of Java source code can be found in the open literature. The objective of the proposed thesis is to extract software metrics from Java source code for authorship identification. Therefore, a set of metrics of Java source code is recommended in this thesis work for authorship identification.
Computer code is a special category of written text that can be executed on computers. We can explore the question of authorship in literature (i.e., literary works) in order to get a better perspective on the question of code authorship.

2.1 Authorship Analysis in Literature

Computers have frequently been used to analyze literary style [Holmes 1985] [Kenny 1986] [Kjell and Frieder 1992]. The objectives were to characterize authors by the values of parameters extracted automatically from the written text. The characterizations were then used to resolve authorship disputes, and to display changes in an author’s style with time or other factors such as mood [Kjell and Frieder 1992]. Holmes [Holmes 1985] divided the features of literary style into three groups: word and sentence features, vocabulary features, and syntactic features.

In as early as 1887, authorship of Shakespearean plays was explored using word length distribution statistics [Kjell and Frieder 1992]. It was assumed that works from different authors would exhibit different frequency distributions for word and sentence lengths. Distribution of functional words such as articles and connectives was employed
for a stylometric study of the New Testament [Kenny 1986]. In a similar way, sentence lengths were used to resolve authorship disputes in the Federalist Paper [Mosteller and Wallace 1964], and to examine the authorship of The Quiet Don [Kjetsaa 1979]. Very often, a combination of all these features was involved [Stratil and Oakley 1987].

Besides stylometric parameters mentioned above, letter-tuple frequency statistics was used to discriminate between two authors writing in a similar style [Kjell 1994] [Kjell and Frieder 1992]. An N-tuple is a sequence of n contiguous letters in the text. After non-alphabetic characters such as punctuation and spaces are discarded and uppercase letters are converted to lowercase letters, tuple extraction proceeds by a shifting n-character window through the text one letter at a time. Thus adjacent tuples overlap by n-1 letters and a tuple may contain characters from more than one word. The relative frequencies of the n-tuples are calculated based on the total occurrence of all n-tuples.

Advantages of using letter-tuples are: easy feature extraction from text, effectiveness, avoidance of lexical analysis as in word frequency statistics, and large amount of data resulting in less variation within a class [Kjell 1994]. However, it has not been clearly explained why it works. If tuple frequencies encode the favorite words of the respective authors, using those words directly would be more efficient. It is not clear why 2-tuples, i.e., ordered pairs with two letters, are the most effective in authorship analysis among n-tuples for n from 1 to 5 [Kjell 1994] [Kjell and Frieder 1992].
2.2 Authorship Analysis of Software

Authorship analysis for computer software is different from and more difficult than that in literature. First, the stylistic characteristics are not the same. Program compilers and interpreters require strict format of computer text code. Next, people may reuse code, and software may be developed by teams of programmers. Also, code formatters and pretty printers can alter a program's appearance [Krsul and Spafford 1996]. In brief, computer code text is less flexible than text in literature, suggesting more difficulty of authorship analysis in program code than in the literature. However, room still exits for personalizing computer text code at least in the following aspects [Gray et al. 1997].

- The manner in which the task is achieved, such as the algorithms used to solve problems.
- Source code layout such as indentation and spacing.
- Stylistic manners utilized to implement algorithms.
- Choice of the computer platform, programming language, compiler, and editor.

In other words, these features (algorithm, layout, style, and environment) are programmer-specific. Thus, it is essential to extract discriminant software metrics associated with these features for authorship analysis of software.

2.2.1 Executable Code as Source

Sometimes, only executable object code is presented for examination. The common examples of this kind are viruses, worms, logical bombs, and Trojan horses which may attack systems [Krsul and Spafford 1996]. After compilation, much evidence
disappears including layout, comments, and variable names. However, some features still remain as listed below [Spafford and Weeber 1993] [Gray et al. 1997].

- Data structures and algorithms that are indicators of the programmer’s background. Programmers typically prefer to use algorithms and data structures that they were taught in class and with which they feel comfortable.
- Compiler and system information.
- Level of programming skill and areas of knowledge such as the level of sophistication and optimization.
- Use of system and library calls.
- Present errors. Programmers tend to make similar errors.
- The symbol table provided in the object code that has been produced using a debug mode.

Although a lot of information about the hardware platform and compiler can be obtained from metrics of object code, and the executable code can even be decompiled, there is considerable information loss [Gray et al. 1997]. For instance, a number of programs may produce the same executable code. This may account for why so far most authorship analyses deal with computer source code.

2.2.2 Program Code as Source

More often than object code, source code is presented for examination. The text of source code contains at least the following set of characteristic information that may be used for authorship analysis [Spafford and Weeber 1993] [Gray et al. 1997].

- Programming language choice
Various software metrics associated with these features have been considered for authorship identification purpose.

An early work on identifying program authorship was reported by Oman and Cook [Oman and Cook 1989]. They built a Pascal source code analyzer and extracted a set of Boolean measurements based on whether or not the following items occurred in the source code.

- Inline comments on the same line as a source code statements
- Blocked comments (two or more comments occurring together)
- Keywords followed by comments
- Lower case characters only (all source code)
- Upper case characters only (all source code)
- Case used to distinguish between keywords and identifiers
- Underscore used in identifiers
- BEGIN followed by a statement on the same line
- THEN followed by a statement on the same line
They found that these Boolean measurements were not adequate to characterize programming style and to identify authorship. Many of the Boolean measurements were proven inappropriate in measuring program-specific features [Krsul and Spafford 1996].

Krsul and Spafford [Krsul and Spafford 1996] developed a comprehensive set of metrics for C program authorship analysis based on a large amount of rules and metrics introduced in previous research efforts (such as [Oman and Cook 1991] and [Conte et al. 1986]). These metrics were divided into three categories: programming layout metrics, programming style metrics, and programming structure metrics [Krsul and Spafford 1996].

Programming layout metrics, as shown in Table 2.1, are associated with layout of programs and thus are fragile and easily alterable, for example by a code formatter. Programming style metrics, as given in Table 2.2, are also related to the layout of code but are difficult to change. The program structure metrics referred to those that were supposed to be dependent on programming ability and the experience of the programmer, Table 2.3 lists some metrics in this category.
Table 2.1 Programming layout metrics for C programs [Krsul and Spafford 1996]

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STY1</td>
<td>A list of metrics indicating indentation style inside functions</td>
</tr>
<tr>
<td>STY1a</td>
<td>Indentation of C statements within surrounding blocks</td>
</tr>
<tr>
<td>STY1b</td>
<td>Percentage of open braces ({) that are along a line</td>
</tr>
<tr>
<td>STY1c</td>
<td>Percentage of open braces ({) that are the first character in a line</td>
</tr>
<tr>
<td>STY1d</td>
<td>Percentage of open braces ({) that are the last character in a line</td>
</tr>
<tr>
<td>STY1e</td>
<td>Percentage of close braces (}) that are along a line</td>
</tr>
<tr>
<td>STY1f</td>
<td>Percentage of close braces (}) that are the first character in a line</td>
</tr>
<tr>
<td>STY1g</td>
<td>Percentage of close braces (}) that are the last character in a line</td>
</tr>
<tr>
<td>STY1h</td>
<td>Indentation of open braces (}</td>
</tr>
<tr>
<td>STY1i</td>
<td>Indentation of close braces (})</td>
</tr>
<tr>
<td>STY2</td>
<td>Indentation of statements starting with the “else” keyword</td>
</tr>
<tr>
<td>STY3</td>
<td>The use of a separator between function names and parameter lists</td>
</tr>
<tr>
<td>STY4</td>
<td>The use of a separator between function return type and function name</td>
</tr>
<tr>
<td>STY5</td>
<td>A vector of metrics specifying comment style</td>
</tr>
<tr>
<td>STY5a</td>
<td>Use of borders to highlight comments</td>
</tr>
<tr>
<td>STY5b</td>
<td>Percentages of code lines with inline comments</td>
</tr>
<tr>
<td>STY5c</td>
<td>Ratio of lines of block style comments to lines of code</td>
</tr>
<tr>
<td>STY6</td>
<td>Ratio of white lines to lines of code</td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>PRO1</td>
<td>Mean program line length in terms of characters</td>
</tr>
<tr>
<td>PRO2</td>
<td>A vector of metrics of name lengths</td>
</tr>
<tr>
<td>PRO2a</td>
<td>Mean local variable name length</td>
</tr>
<tr>
<td>PRO2b</td>
<td>Mean global variable name length</td>
</tr>
<tr>
<td>PRO2c</td>
<td>Mean function name length</td>
</tr>
<tr>
<td>PRO2d</td>
<td>Mean function parameter name length</td>
</tr>
<tr>
<td>PRO3</td>
<td>A list of metrics of name convention</td>
</tr>
<tr>
<td>PRO3a</td>
<td>Percentage of variable names that start with an uppercase letter</td>
</tr>
<tr>
<td>PRO3b</td>
<td>Percentage of function names that start with an uppercase letter</td>
</tr>
<tr>
<td>PRO3c</td>
<td>Is the underscore character used in names?</td>
</tr>
<tr>
<td>PRO4</td>
<td>Ratio of global variable count to local variable count</td>
</tr>
<tr>
<td>PRO5</td>
<td>Ratio of global variable count to lines of code</td>
</tr>
<tr>
<td>PRO6</td>
<td>Preference of either “while”, “for”, or “do” loops</td>
</tr>
<tr>
<td>PRO7</td>
<td>Are comments merely an echo of the code</td>
</tr>
</tbody>
</table>
Table 2.3 Programming structure metrics for C programs [Krsul and Spafford 1996]

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM1</td>
<td>Percentage of “int” function definitions</td>
</tr>
<tr>
<td>PSM2</td>
<td>Percentage of “void” function definitions</td>
</tr>
<tr>
<td>PSM3</td>
<td>Use of debugging symbols or keywords</td>
</tr>
<tr>
<td>PSM4</td>
<td>Use of the assert macro</td>
</tr>
<tr>
<td>PSM5</td>
<td>Average lines of code per function</td>
</tr>
<tr>
<td>PSM6</td>
<td>Ratio of variable count to lines of code</td>
</tr>
<tr>
<td>PSM7</td>
<td>Percentage of static global variables</td>
</tr>
<tr>
<td>PSM8</td>
<td>Ratio of decision count to lines of code</td>
</tr>
<tr>
<td>PSM9</td>
<td>Is the keyword “goto” used?</td>
</tr>
<tr>
<td>PSM10</td>
<td>A list of complexity metrics (such as the Cyclomatic complexity number [McCabe 1976])</td>
</tr>
<tr>
<td>PSM11</td>
<td>Error detection after system calls</td>
</tr>
<tr>
<td>PSM12</td>
<td>Does the programmer rely on internal representations of data objects?</td>
</tr>
<tr>
<td>PSM13</td>
<td>Do comments agree with code?</td>
</tr>
</tbody>
</table>

As stated by Spafford and Weeber [Spafford and Weeber 1993], a feature was said to be writer-specific if it showed small variations in the writings of one author and large variations over the writings of different authors. Unfortunately, the two criteria for metric selection were not necessarily correlated. Elimination of metrics that showed large variation among programs of one programmer, also surprisingly eliminated those metrics that showed large variation among different programmers as well, resulting in
unsatisfactory classifications. To solve the problem, a tool was built to visualize for each metric the variation of each programmer and the variation among programmers. A much smaller set of metrics, than those mentioned in Tables 2.1, 2.2, and 2.3, was chosen for final statistical analysis [Krsul and Spafford 1996].

Authorship analysis was also explored by another research group. Gray and his colleagues [Gray et al. 1998] proposed that the metrics listed in Table 2.4 might be related to authorship.

Table 2.4 Metrics proposed by Gray and his colleagues [Gray et al. 1998] for possible relationship to authorship

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric 1</td>
<td>Mean length of source code lines in terms of number of characters</td>
</tr>
<tr>
<td>Metric 2</td>
<td>Mean variable name length in terms of number of characters</td>
</tr>
<tr>
<td>Metric 3</td>
<td>Whether or not pointers are used</td>
</tr>
<tr>
<td>Metric 4</td>
<td>Mean length of a function in lines of code</td>
</tr>
<tr>
<td>Metric 5</td>
<td>Ratio of comment lines to non-comment lines of code</td>
</tr>
<tr>
<td>Metric 6</td>
<td>Ratio of blank lines to non-blank lines</td>
</tr>
<tr>
<td>Metric 7</td>
<td>Whether or not global variables are used</td>
</tr>
</tbody>
</table>

In addition, Gray and his colleagues [Gray et al. 1998] suggested that the following traditional software metrics, usually used for plagiarism detection, might also be utilized for authorship analysis.

- Volume measured by Halstead’s $V = n \log N$ [Halstead 1977]
• Control flow measured by McCabe’s \( V(G) = \) number of binary branches [McCabe 1976]

• Structure measured by Leach’s coupling assessment [Leach 1995]

• Data dependency measured by Bieman and Debnath’s GPG assessment [Bieman and Debnath 1985]

• Nesting depth measured by program nesting depth and average nesting depth [Dunsmore 1984]

• Control structure measured by Nejmeh’s NPATH [Nejmeh 1988]

To extract metrics for authorship analysis, a system called IDENTIFIED (Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination) was developed. It was claimed that the system could extract most of these measurements automatically [Gray et al. 1998].

Using the IDENTIFIED system, MacDonell and his colleagues [MacDonell et al. 1999] extracted 26 measurements from each standard C++ program for authorship analysis. Table 2.5 lists most of these measurements.
Table 2.5 Measurements extracted by the IDENTIFIED system [Gray et al. 1998] for authorship analysis of C++ source code [MacDonell et al. 1999]

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPS</td>
<td>Proportion of letters that are upper case</td>
</tr>
<tr>
<td>CCN</td>
<td>McCabe’s cyclomatic complexity number, V(G)</td>
</tr>
<tr>
<td>COM</td>
<td>Proportion of LOC that are purely comments</td>
</tr>
<tr>
<td>COND-1</td>
<td>Average number of #if per NCLOC</td>
</tr>
<tr>
<td>COND-2</td>
<td>Average number of #ifdef per NCLOC</td>
</tr>
<tr>
<td>COND-3</td>
<td>Average number of #ifndef per NCLOC</td>
</tr>
<tr>
<td>COND-4</td>
<td>Average number of #else per NCLOC</td>
</tr>
<tr>
<td>COND-5</td>
<td>Average number of #endif per NCLOC</td>
</tr>
<tr>
<td>DEC</td>
<td>Average number of decision statements per NCLOC</td>
</tr>
<tr>
<td>DEC-IF</td>
<td>Average number of if statements per NCLOC</td>
</tr>
<tr>
<td>DEC-SWITCH</td>
<td>Average number of switch statements per NCLOC</td>
</tr>
<tr>
<td>DEC-WHILE</td>
<td>Average number of while statements per NCLOC</td>
</tr>
<tr>
<td>ENDCOM</td>
<td>Proportion of end-of-block braces labeled with comment</td>
</tr>
<tr>
<td>GOTO</td>
<td>Average number of gotos per non-comment LOC (NCLOC)</td>
</tr>
<tr>
<td>INLCOM</td>
<td>Proportion of LOC that have inline comments</td>
</tr>
<tr>
<td>LOC</td>
<td>Non-white space lines of code</td>
</tr>
<tr>
<td>LOCCHARS</td>
<td>Mean number of characters per line</td>
</tr>
<tr>
<td>SPACE-1</td>
<td>Proportion of operators with white space on both sides</td>
</tr>
<tr>
<td>SPACE-2</td>
<td>Proportion of operators with white space on the left side</td>
</tr>
<tr>
<td>SPACE-3</td>
<td>Proportion of operators with white space on the right side</td>
</tr>
<tr>
<td>SPACE-4</td>
<td>Proportion of operators with white space on neither side</td>
</tr>
<tr>
<td>WHITE</td>
<td>Proportion of lines that are blank</td>
</tr>
</tbody>
</table>
3.1 Feature Analysis of Programs Written in Java

Java is one of the popular programming languages for application development [Lewis and Loftus 1998]. It is an object-oriented language similar to C++, but simplified to eliminate language features that cause common programming errors. Java source code files are compiled into bytecode, which can then be executed by a Java interpreter. Compiled Java code can run on most computers because Java interpreters and runtime environments, known as Java Virtual Machines (VMs), exist for most operating systems such as UNIX, the Macintosh OS, and Windows. Bytecode can also be converted directly into machine language instructions by a just-in-time compiler. Java is also a general purpose programming language with a number of features that make the language well suited for use on the World Wide Web. Java applets can be downloaded from a Web server and run on a computer by a Java-compatible Web browser such as Netscape Navigator or Microsoft Internet Explorer. Compared to C/C++, Java has no pointers, no global functions that are defined outside classes, automatic garbage collection, etc.
3.2 Data Source and Metrics

The data that was used for authorship analysis in this study was from three origins. The first part of the data was from computer science classes. To protect the students' privacy, all names were removed from the program texts. The programs were grouped per individual programmer. Forty groups of data were thus collected with each group containing four to six programs.

To be generic with respect to the data sources, a set of programs was collected from Internet shareware. Five groups of programs were collected from the following URL addresses:

- http://math.hws.edu/eck/

Seven to ten pieces of Java source code were collected for each group of programs.

The third source for data was from a fellow graduate student who voluntarily shared her programs with the author. This sample contains six programs.

The lengths of all of the sample programs in the collection ranged from several hundred to several thousand lines of Java source code.

To extract metrics for authorship identification, the Java programs were subjected to a Windows application that was developed by the author. The Windows application was developed with Visual C++ and MFC with the software package Microsoft Visual
Studio 6.0. When the application is run, a window opens and the input Java program is scanned. A set of metrics (see Section 3.2) are extracted from the Java program. The metrics values are added to a file of Microsoft Access 2000 that is connected with the Windows application by an ODBC Microsoft Access driver.

The database file of Microsoft Access 2000 contains one large table in which each metric is one column. Each row of the table contains metric values of one Java program. When the Windows application finishes the scanning of one program, the metric values are added to one row under the corresponding metric names.

The C++ program that was developed by the author for metrics extraction contains nine header files (.h), eight source files (.cpp), four resource files, and one readme text file. Total number of lines in the source files is approximately 5000. The header files and source files are listed in Appendix C.

The metrics that were extracted and collected in this study for Java program authorship identification were adapted from the metrics used by Krsul and Spafford for C programs [Krsul and Spafford 1996], the metrics proposed by Gray and his colleagues [Gray et al. 1998], and the metrics utilized for C++ source code [MacDonell et al. 1999]. Tables 3.1, 3.2 and 3.3 list the extracted metrics for programming layout, programming style, and programming structure, respectively.
Table 3.1 Programming layout metrics extracted from the source code of Java programs

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STY1</td>
<td>A list of metrics indicating indentation style</td>
</tr>
<tr>
<td>STY1a</td>
<td>Percentage of open braces ({) that are along a line</td>
</tr>
<tr>
<td>STY1b</td>
<td>Percentage of open braces ({) that are the first character in a line</td>
</tr>
<tr>
<td>STY1c</td>
<td>Percentage of open braces ({) that are the last character in a line</td>
</tr>
<tr>
<td>STY1d</td>
<td>Percentage of close braces (}) that are along a line</td>
</tr>
<tr>
<td>STY1e</td>
<td>Percentage of close braces (}) that are the first character in a line</td>
</tr>
<tr>
<td>STY1f</td>
<td>Percentage of close braces (}) that are the last character in a line</td>
</tr>
<tr>
<td>STY1g</td>
<td>Average indentation in white spaces after open braces ({)</td>
</tr>
<tr>
<td>STY1h</td>
<td>Average indentation in tabs after open braces ({)</td>
</tr>
<tr>
<td>STY2</td>
<td>A vector of metrics specifying comment style</td>
</tr>
<tr>
<td>STY2a</td>
<td>Percentages of pure comment lines among lines containing comments</td>
</tr>
<tr>
<td>STY2b</td>
<td>Percentages of &quot;//&quot; style comments among &quot;//&quot; and &quot;/*&quot; style comments</td>
</tr>
<tr>
<td>STY3</td>
<td>Percentages of condition lines where the statements are on the same line as the condition</td>
</tr>
<tr>
<td>STY4</td>
<td>Average white spaces to the left side of operators #</td>
</tr>
<tr>
<td>STY5</td>
<td>Average white spaces to the right side of operators #</td>
</tr>
<tr>
<td>STY6</td>
<td>Ratio of blank lines to code lines (including comment lines)</td>
</tr>
<tr>
<td>STY7</td>
<td>Ratio of comment lines to non-comment lines*</td>
</tr>
<tr>
<td>STY8</td>
<td>Ratio of code lines containing comment to code lines without any comments</td>
</tr>
</tbody>
</table>

#: Operators included "+", "-", "*", "/", "/\*", "/=", and "+=", "-=", "*=", "/=", "%=", and "===".  
*: Comment lines are pure comment lines. Non-comment lines include lines with inline comments.
Table 3.2 Programming style metrics extracted from the source code of Java programs

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO1</td>
<td>Mean program line length in terms of characters</td>
</tr>
<tr>
<td>PRO2</td>
<td>A vector of metrics of name lengths</td>
</tr>
<tr>
<td>PRO2a</td>
<td>Mean variable name length</td>
</tr>
<tr>
<td>PRO2b</td>
<td>Mean function name length</td>
</tr>
<tr>
<td>PRO3</td>
<td>Character preference of uppercase, lowercase, underscore, or dollar sign for name convention</td>
</tr>
<tr>
<td>PRO3a</td>
<td>Percentage of uppercase characters</td>
</tr>
<tr>
<td>PRO3b</td>
<td>Percentage of lowercase characters</td>
</tr>
<tr>
<td>PRO3c</td>
<td>Percentage of underscores</td>
</tr>
<tr>
<td>PRO3d</td>
<td>Percentage of dollar signs</td>
</tr>
<tr>
<td>PRO4</td>
<td>Preference of either “while”, “for”, or “do” loops</td>
</tr>
<tr>
<td>PRO4a</td>
<td>Percentage of “while” in total of “while”, “for”, and “do”</td>
</tr>
<tr>
<td>PRO4b</td>
<td>Percentage of “for” in total of “while”, “for”, and “do”</td>
</tr>
<tr>
<td>PRO4c</td>
<td>Percentage of “do” in total of “while”, “for”, and “do”</td>
</tr>
<tr>
<td>PRO5</td>
<td>Preference of either “if-else” or “switch-case” conditions</td>
</tr>
<tr>
<td>PRO5a</td>
<td>Percentage of “if” and “else” in total of “if”, “else”, “switch”, and “case”</td>
</tr>
<tr>
<td>PRO5b</td>
<td>Percentage of “switch” and “case” in total of “if”, “else”, “switch”, and “case”</td>
</tr>
<tr>
<td>PRO5c</td>
<td>Percentage of “if” in total of “if” and “else”</td>
</tr>
<tr>
<td>PRO5d</td>
<td>Percentage of “switch” in total of “switch” and “case”</td>
</tr>
</tbody>
</table>

*: Variables included only nine common data types: “short”, “int”, “long”, “float”, “double”, “byte”, “char”, “Boolean”, and “string".
Table 3.3 Programming structure metrics extracted from the source code of Java programs

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM1</td>
<td>Average non-comment lines per class/interface</td>
</tr>
<tr>
<td>PSM2</td>
<td>Average number of primitive variables per class/interface</td>
</tr>
<tr>
<td>PSM3</td>
<td>Average number of functions per class/interface</td>
</tr>
<tr>
<td>PSM4</td>
<td>Ratio of interfaces to classes</td>
</tr>
<tr>
<td>PSM5</td>
<td>Ratio of primitive variable count to lines of non-comment code</td>
</tr>
<tr>
<td>PSM6</td>
<td>Ratio of function count to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7</td>
<td>A list of ratios of keywords to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7a</td>
<td>Ratio of keyword “static” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7b</td>
<td>Ratio of keyword “extends” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7c</td>
<td>Ratio of keyword “class” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7d</td>
<td>Ratio of keyword “abstract” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7e</td>
<td>Ratio of keyword “implements” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7f</td>
<td>Ratio of keyword “import” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7g</td>
<td>Ratio of keyword “instanceof” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7h</td>
<td>Ratio of keyword “interface” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7i</td>
<td>Ratio of keyword “native” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7j</td>
<td>Ratio of keyword “new” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7k</td>
<td>Ratio of keyword “package” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7l</td>
<td>Ratio of keyword “private” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7m</td>
<td>Ratio of keyword “public” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7n</td>
<td>Ratio of keyword “protected” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7o</td>
<td>Ratio of keyword “this” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7p</td>
<td>Ratio of keyword “super” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7q</td>
<td>Ratio of keyword “try” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7r</td>
<td>Ratio of keyword “throw” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7s</td>
<td>Ratio of keyword “catch” to lines of non-comment code</td>
</tr>
<tr>
<td>PSM7t</td>
<td>Ratio of keyword “final” to lines of non-comment code</td>
</tr>
</tbody>
</table>
Fifty six metrics were extracted from the Java programs. The values of the 56 metrics were calculated for each of the 46 groups of programs. Each group contained 4 to 10 programs (see Section 3.2 for details). With so much information (so many metric values), one question might come up first: “Do all metrics contribute significantly to the authorship prediction?” The answer is: some do and the others might not. Thus, the first step for data analysis might be to extract contributive information from the obtained data set and to reduce informative noise. One way of information extraction is to select a set of contributive variables. Those variables can be selected manually or automatically, as explained in the following two sections.

4.1 Manual Variable Selection for Classification Analysis

To select contributive variables to the classification from the metrics set, several factors need to be considered. The first consideration is the significance of each individual variable for the group classification.

The significance level was measured by a statistical procedure called one-way ANOVA (analysis of variance) [SAS 1990]. One-way ANOVA was performed for each
individual variable with the SPSS statistical package (Version 11.0.1, SPSS Inc., Chicago, IL). Sums and mean squares between groups and within groups were calculated. F values (statistical ratios of mean square (variation) between groups to mean square (variation) within groups) and significance levels for group classification were then computed for each metric. To be a little conservative, the significance level to select was set at 0.15. This would not lose important information carried by those intermediate significant metrics.

The second aspect considers the relationship among the variables. With the SPSS statistical package, Pearson’s correlations were performed among all of the metrics. A Pearson’s correlation coefficient between two variables signals the linearity of them. To measure the statistical performance, a two-tailed significance test was specified. If the correlation coefficient between two variables is 1.0 or −1.0, this indicates that the two metrics carry the same information, and thus keeping one of them would be enough. One example is the relationship between PRO5a (percentage of “if” and “else” in total of “if”, “else”, “switch”, and “case”) and PRO5b (percentage of “switch” and “case” in total of “if”, “else”, “switch”, and “case”). They carry the same information and thus keeping either one is enough. However, one metric may be derived from two or more other metrics, as listed in Table 4.1. This may introduce redundancy in the data sets. Thus, to reduce the redundancy, one metric in a derivation variable set should be discarded. As to which one should be excluded from the data set depends on their correlation with each other and their significance in the group prediction. Theoretically, a variable with low F value, low significance level in the classification, and high correlation with other variable(s) would be excluded.
<table>
<thead>
<tr>
<th>Metrics derivation set</th>
<th>Metrics and derivation formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \text{STY}1a + \text{STY}1b + \text{STY}1c = 1 )</td>
</tr>
<tr>
<td>2</td>
<td>( \text{STY}1d + \text{STY}1e + \text{STY}1f = 1 )</td>
</tr>
<tr>
<td>3</td>
<td>( \text{PR}03a + \text{PR}03b + \text{PR}03c + \text{PR}03d = 1 )</td>
</tr>
<tr>
<td>4</td>
<td>( \text{PR}04a + \text{PR}04b + \text{PR}04c = 1 )</td>
</tr>
<tr>
<td>5</td>
<td>( \text{PR}05a + \text{PR}05b = 1 )</td>
</tr>
</tbody>
</table>

**STY1a:** Percentage of open braces (\{\}) that are along a line  
**STY1b:** Percentage of open braces (\{\}) that are the first character in a line  
**STY1c:** Percentage of open braces (\{\}) that are the last character in a line  
**STY1d:** Percentage of close braces (\}) that are along a line  
**STY1e:** Percentage of close braces (\}) that are the first character in a line  
**STY1f:** Percentage of close braces (\}) that are the last character in a line  
**PR03a:** Percentage of uppercase characters  
**PR03b:** Percentage of lowercase characters  
**PR03c:** Percentage of underscores  
**PR03d:** Percentage of dollar signs  
**PR04a:** Percentage of “while” in total of “while”, “for”, and “do”  
**PR04b:** Percentage of “for” in total of “while”, “for”, and “do”  
**PR04c:** Percentage of “do” in total of “while”, “for”, and “do”  
**PR05a:** Percentage of “if” and “else” in total of “if”, “else”, “switch”, and “case”  
**PR05b:** Percentage of “switch” and “case” in total of “if”, “else”, “switch”, and “case”

### 4.2 Automatic Variable Selection for the Classification Analysis

Contributive variables were also selected automatically by a statistical procedure called stepwise discriminant analysis (SDA). Stepwise discriminant analysis was performed with the SAS statistical package (Version 8.0, SAS Institute Inc., Cary, NC). A model of forward stepwise analysis was specified for SDA. The discrimination was built step by step. At each step, all variables were reviewed and evaluated to determine which one would contribute most to the discrimination among groups. That variable would then be included in the model, and the process started again. To be a little conservative, the statistical significance level to enter was set at 0.15. If all significance
levels of variables in the discrimination between groups were more than 0.15, the process would stop.

In this model of variable selection, variable correlation and derivation were all taken care of by the statistical procedure. For example, if one of the two highly correlated variables was included in the model, the significance level of the other variable would drop dramatically. The reason was that the variation of the second variable was largely represented by the first variable and the variation had already been included in the calculation.

4.3 Classification Analyses

After a set of contributive variables was selected, a statistical procedure called canonical discriminant analysis (CDA) was used for the classification. CDA was performed with the statistical package SAS (Version 8.0, SAS Institute Inc., Cary, NC). For CDA, a parametric method based on a normal distribution within each class was used to derive linear discriminant function with pooled covariance matrices. Canonical variates, linear combinations of metrics, were also derived from the metrics variables. Canonical variates summarize between-class variations [SAS 1990]. Due to very limited number of samples for each group (i.e., 4 to 10), cross-validation was specified in CDA analysis. So each program was classified by the discriminatory rule that was computed after the sample was deleted from the data set. In this way, samples would not affect their own classification and hence a realistic estimation of performance would be obtained [SAS 1990]. Classification accuracy was calculated as the percentage of correctly allocated samples.
4.4 Data Sets

Since one Java program may involve more than two source files, each sample can be either one Java source file, or all the Java source files for each compilation in which the main function is compiled. So there are two approaches. Also, contributive variables can be extracted either manually or automatically. Thus, four data sets and four corresponding result sets were obtained. The data sets were labeled A, B, C, and D, as shown in Table 4.2.

Table 4.2 Four data sets of different data treatments

<table>
<thead>
<tr>
<th>Sample Source</th>
<th>Variable Selection Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each Java source file as a sample</td>
<td>Manually: A</td>
</tr>
<tr>
<td></td>
<td>Stepwise Discriminant Analysis: C</td>
</tr>
<tr>
<td>All Java source files for each compilation as a sample</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>

When each Java source file is treated as a sample, the total number of samples is 259. When a sample is associated with one compilation, the total sample number is 225. To capture and reflect the difference, another metric was added to the variable set. The metric was FPC (File Per Compilation) that represents the number of Java source files per compilation. For individual metrics, its values might miss in some cases. For instance, metric PRO5c is the percentage of “if” in total of “if” and “else”, as shown in Table 3.2. If a program did not use any “if” or “else”, the value of PRO5c would miss for the program since the denominator would be zero.
4.5 Results and Discussion

Table 4.3 shows the selected metric sets after the variable selection process. In general, similar metric sets were selected by both approaches (i.e., based on source files) of variable selection. Most variables chosen by one method (i.e., either manual or automatic) were also selected by the other method. Nevertheless, the small difference between the two variable sets reflects different philosophies behind the two variable selection approaches. In manual variable selection, the process was static and thus no statistical information would change after a metric was included. In contrast, selection by SDA was dynamic and thus the selection of a variable would alter the significance of the remaining variables, because the correlation and interaction among the variables were taken into account.

To show the difference between the two approaches (i.e., based on source files), one example might be the selection of metrics STY1a (percentage of open braces (}) that are along a line), STY1b (percentage of open braces (}) that are the first character in a line), and STY1c (percentage of open braces (}) that are the last character in a line) in data sets A and C. In both data sets A and C, each Java file was treated as a sample. The difference is that in data set A, variables were selected manually but in data set C, variables were selected automatically. For data set A, one-way ANOVA was performed and the results showed that the F values of STY1a, STY1b, and STY1c were 72.8, 13.7, and 80.3, respectively. Although all of them were below the entry level of significance, only two of them could be included since one can be derived from the other two, as indicated in Table 4.1. Pearson's correlation revealed that STY1a was highly correlated
with STY1c (R=0.932 and P<0.001). Thus STY1c and STY1b were manually selected for data set A.

During automatic selection by SDA for data set C, STY1c was first selected since its F value was the highest. The selection of STY1c decreased the F value of STY1a dramatically to 14.8 and increased the F value of STY1b a little bit to 14.8. Subsequent inclusion of other variables also altered the performance of STY1a and STY1b in the discrimination. The final result was that STY1a retained in data set C but STY1b was discarded. Thus STY1a and STY1c were automatically selected for data set C.

In Table 4.3, it was also observed that the total number of selected variables for data sets C and D (in which, variables were selected automatically) were higher than those for data sets A and B (in which, variables were selected manually). In other words, stepwise discriminant analysis retained more metrics than manual selection. This might be due to the different procedures of the two variable-selection methods.

For all data sets, metric PRO5d was excluded because of too many missing values. It indicates that statements with “switch” and “case” were seldom used in the data source. Another metric, PSM7i, was discarded due to the constancy of its values, i.e., the values were all zeros. It seems that the keywords “native” were not used in the data source.

For both data sets B and D, metric FPC, i.e., files per compilation, was selected. It implies that separating a program into a number of source files might be a good piece of a programmer’s fingerprint.
Table 4.3 Selected metric sets for the four data treatments (for explanation/detail on the first column metrics, refer to Tables 3.1, 3.2, and 3.3)

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Data Treatments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>FPC</td>
<td>+</td>
</tr>
<tr>
<td>STY1a</td>
<td></td>
</tr>
<tr>
<td>STY1b</td>
<td>+</td>
</tr>
<tr>
<td>STY1c</td>
<td>+</td>
</tr>
<tr>
<td>STY1d</td>
<td></td>
</tr>
<tr>
<td>STY1e</td>
<td>+</td>
</tr>
<tr>
<td>STY1f</td>
<td></td>
</tr>
<tr>
<td>STY1g</td>
<td>+</td>
</tr>
<tr>
<td>STY1h</td>
<td>+</td>
</tr>
<tr>
<td>STY1i</td>
<td></td>
</tr>
<tr>
<td>STY1j</td>
<td></td>
</tr>
<tr>
<td>STY1k</td>
<td></td>
</tr>
<tr>
<td>STY1l</td>
<td></td>
</tr>
<tr>
<td>STY1m</td>
<td></td>
</tr>
<tr>
<td>STY1n</td>
<td></td>
</tr>
<tr>
<td>STY1o</td>
<td></td>
</tr>
<tr>
<td>STY1p</td>
<td></td>
</tr>
<tr>
<td>STY1q</td>
<td></td>
</tr>
<tr>
<td>STY1r</td>
<td></td>
</tr>
<tr>
<td>STY1s</td>
<td></td>
</tr>
<tr>
<td>STY1t</td>
<td></td>
</tr>
<tr>
<td>STY1u</td>
<td></td>
</tr>
<tr>
<td>STY1v</td>
<td></td>
</tr>
<tr>
<td>STY1w</td>
<td></td>
</tr>
<tr>
<td>STY1x</td>
<td></td>
</tr>
<tr>
<td>STY1y</td>
<td></td>
</tr>
<tr>
<td>STY1z</td>
<td></td>
</tr>
<tr>
<td>PRO1</td>
<td></td>
</tr>
<tr>
<td>PRO2a</td>
<td></td>
</tr>
<tr>
<td>PRO2b</td>
<td></td>
</tr>
<tr>
<td>PRO2c</td>
<td></td>
</tr>
<tr>
<td>PRO2d</td>
<td></td>
</tr>
<tr>
<td>PRO2e</td>
<td></td>
</tr>
<tr>
<td>PRO2f</td>
<td></td>
</tr>
<tr>
<td>PRO2g</td>
<td></td>
</tr>
<tr>
<td>PRO2h</td>
<td></td>
</tr>
<tr>
<td>PRO2i</td>
<td></td>
</tr>
<tr>
<td>PRO2j</td>
<td></td>
</tr>
<tr>
<td>PRO2k</td>
<td></td>
</tr>
<tr>
<td>PRO2l</td>
<td></td>
</tr>
<tr>
<td>PRO2m</td>
<td></td>
</tr>
<tr>
<td>PRO2n</td>
<td></td>
</tr>
<tr>
<td>PRO2o</td>
<td></td>
</tr>
<tr>
<td>PRO2p</td>
<td></td>
</tr>
<tr>
<td>PRO2q</td>
<td></td>
</tr>
<tr>
<td>PRO2r</td>
<td></td>
</tr>
<tr>
<td>PRO2s</td>
<td></td>
</tr>
<tr>
<td>PRO2t</td>
<td></td>
</tr>
<tr>
<td>PRO2u</td>
<td></td>
</tr>
<tr>
<td>PRO2v</td>
<td></td>
</tr>
<tr>
<td>PRO2w</td>
<td></td>
</tr>
<tr>
<td>PRO2x</td>
<td></td>
</tr>
<tr>
<td>PRO2y</td>
<td></td>
</tr>
<tr>
<td>PRO2z</td>
<td></td>
</tr>
<tr>
<td>PRO3a</td>
<td></td>
</tr>
<tr>
<td>PRO3b</td>
<td></td>
</tr>
<tr>
<td>PRO3c</td>
<td></td>
</tr>
<tr>
<td>PRO3d</td>
<td></td>
</tr>
<tr>
<td>PRO3e</td>
<td></td>
</tr>
<tr>
<td>PRO3f</td>
<td></td>
</tr>
<tr>
<td>PRO3g</td>
<td></td>
</tr>
<tr>
<td>PRO3h</td>
<td></td>
</tr>
<tr>
<td>PRO3i</td>
<td></td>
</tr>
<tr>
<td>PRO3j</td>
<td></td>
</tr>
<tr>
<td>PRO3k</td>
<td></td>
</tr>
<tr>
<td>PRO3l</td>
<td></td>
</tr>
<tr>
<td>PRO3m</td>
<td></td>
</tr>
<tr>
<td>PRO3n</td>
<td></td>
</tr>
<tr>
<td>PRO3o</td>
<td></td>
</tr>
<tr>
<td>PRO3p</td>
<td></td>
</tr>
<tr>
<td>PRO3q</td>
<td></td>
</tr>
<tr>
<td>PRO3r</td>
<td></td>
</tr>
<tr>
<td>PRO3s</td>
<td></td>
</tr>
<tr>
<td>PRO3t</td>
<td></td>
</tr>
<tr>
<td>PRO3u</td>
<td></td>
</tr>
<tr>
<td>PRO3v</td>
<td></td>
</tr>
<tr>
<td>PRO3w</td>
<td></td>
</tr>
<tr>
<td>PRO3x</td>
<td></td>
</tr>
<tr>
<td>PRO3y</td>
<td></td>
</tr>
<tr>
<td>PRO3z</td>
<td></td>
</tr>
<tr>
<td>PRO4a</td>
<td></td>
</tr>
<tr>
<td>PRO4b</td>
<td></td>
</tr>
<tr>
<td>PRO4c</td>
<td></td>
</tr>
<tr>
<td>PRO4d</td>
<td></td>
</tr>
<tr>
<td>PRO4e</td>
<td></td>
</tr>
<tr>
<td>PRO4f</td>
<td></td>
</tr>
<tr>
<td>PRO4g</td>
<td></td>
</tr>
<tr>
<td>PRO4h</td>
<td></td>
</tr>
<tr>
<td>PRO4i</td>
<td></td>
</tr>
<tr>
<td>PRO4j</td>
<td></td>
</tr>
<tr>
<td>PRO4k</td>
<td></td>
</tr>
<tr>
<td>PRO4l</td>
<td></td>
</tr>
<tr>
<td>PRO4m</td>
<td></td>
</tr>
<tr>
<td>PRO4n</td>
<td></td>
</tr>
<tr>
<td>PRO4o</td>
<td></td>
</tr>
<tr>
<td>PRO4p</td>
<td></td>
</tr>
<tr>
<td>PRO4q</td>
<td></td>
</tr>
<tr>
<td>PRO4r</td>
<td></td>
</tr>
<tr>
<td>PRO4s</td>
<td></td>
</tr>
<tr>
<td>PRO4t</td>
<td></td>
</tr>
<tr>
<td>PRO4u</td>
<td></td>
</tr>
<tr>
<td>PRO4v</td>
<td></td>
</tr>
<tr>
<td>PRO4w</td>
<td></td>
</tr>
<tr>
<td>PRO4x</td>
<td></td>
</tr>
<tr>
<td>PRO4y</td>
<td></td>
</tr>
<tr>
<td>PRO4z</td>
<td></td>
</tr>
<tr>
<td>PRO5a</td>
<td></td>
</tr>
<tr>
<td>PRO5b</td>
<td></td>
</tr>
<tr>
<td>PRO5c</td>
<td></td>
</tr>
<tr>
<td>PRO5d</td>
<td></td>
</tr>
<tr>
<td>PRO5e</td>
<td></td>
</tr>
<tr>
<td>PRO5f</td>
<td></td>
</tr>
<tr>
<td>PRO5g</td>
<td></td>
</tr>
<tr>
<td>PRO5h</td>
<td></td>
</tr>
<tr>
<td>PRO5i</td>
<td></td>
</tr>
<tr>
<td>PRO5j</td>
<td></td>
</tr>
<tr>
<td>PRO5k</td>
<td></td>
</tr>
<tr>
<td>PRO5l</td>
<td></td>
</tr>
<tr>
<td>PRO5m</td>
<td></td>
</tr>
<tr>
<td>PRO5n</td>
<td></td>
</tr>
<tr>
<td>PRO5o</td>
<td></td>
</tr>
<tr>
<td>PRO5p</td>
<td></td>
</tr>
<tr>
<td>PRO5q</td>
<td></td>
</tr>
<tr>
<td>PRO5r</td>
<td></td>
</tr>
<tr>
<td>PRO5s</td>
<td></td>
</tr>
<tr>
<td>PRO5t</td>
<td></td>
</tr>
<tr>
<td>PRO5u</td>
<td></td>
</tr>
<tr>
<td>PRO5v</td>
<td></td>
</tr>
<tr>
<td>PRO5w</td>
<td></td>
</tr>
<tr>
<td>PRO5x</td>
<td></td>
</tr>
<tr>
<td>PRO5y</td>
<td></td>
</tr>
<tr>
<td>PRO5z</td>
<td></td>
</tr>
<tr>
<td>PSM1</td>
<td></td>
</tr>
<tr>
<td>PSM2</td>
<td></td>
</tr>
<tr>
<td>PSM3</td>
<td></td>
</tr>
<tr>
<td>PSM4</td>
<td></td>
</tr>
<tr>
<td>PSM5</td>
<td></td>
</tr>
<tr>
<td>PSM6</td>
<td></td>
</tr>
<tr>
<td>PSM7a</td>
<td></td>
</tr>
<tr>
<td>PSM7b</td>
<td></td>
</tr>
<tr>
<td>PSM7c</td>
<td></td>
</tr>
<tr>
<td>PSM7d</td>
<td></td>
</tr>
</tbody>
</table>

continued on next page
Table 4.3 Selected metric sets for the four data treatments (for explanation/detail on the first column metrics, refer to Tables 3.1, 3.2, and 3.3) (continued)

<table>
<thead>
<tr>
<th>Metrics</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM7e</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7f</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7g</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7h</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7i*</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7j</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7k</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7l</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7m</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7n</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7o</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7p</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7q</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7r</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7s</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PSM7t</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Total number of selected metrics: 45, 41, 46, 48

#: FPC: The number of Java source files per compilation when a sample was associated with one compilation.

*: PRO5d was excluded due to too many missing values, PSM7i was constant because the variations of its values were zeroes.

+: Checked variables were selected.
Table 4.4 lists the classification accuracies of discriminant analysis. The classification accuracies were calculated with the selected metrics, or the derived canonical variates. The total number of canonical variates which were derived was equal to the number of selected metrics for the discriminant analysis. However, it is often the case that initially several canonical variates summarized the majority of between-group variations. To be a little conservative, first 20 canonical variates were used in the discriminant analysis, and they summarized over 97% of the between-group variations. The canonical correlations of canonical variates after the 20th with groups were insignificant (P<0.15) for all data sets.

As shown in Table 4.4, based on the original metric values, the authorship of 62.6-67.2% of Java source files could be assigned correctly to their 46 original authors. With the derived canonical variates, the percentages could be higher, up to 85.8%. This indicates that the authorship identification was effective and those metrics were contributive.

When each compilation was treated as one sample, in data sets B and D, the classification accuracies were higher than those when each Java source file was used as a sample, in data sets A and C. For the case of each source file as a sample, the variation of the metric values tends to increase within groups. One example of the reason could be that some files were short while some others were relatively long. However, treating each source file as a sample did not enlarge the between-class variation since the data source in each group did not change. Consequently, increased within-class variation would surely reduce the discriminant ability between groups and result in less classification accuracies.
No difference was revealed between the classifications with the original metric sets selected by the two approaches (i.e., manual and automatic selection). The classification accuracies were 62.7% and 67.2% when the metrics were selected manually while those values were 62.6% and 66.6% when the metrics were chosen by SDA. However, the two approaches of variable selection (i.e., manual and automatic) affected the classification results when canonical variates were used. In Table 4.4, it was observed that the highest classification accuracy of data set C, 80.0%, was higher than that of data set A, 77.6%. So was the highest classification accuracy of data set D, 85.8%, compared to that of data set B, 82.5%. It implies that SDA retained more between-class variations than the manual selection of the metrics.

To measure how much the between-class variation was kept in the metrics selected by SDA, a discriminant analysis with all of the metrics was performed. The result showed that the classification accuracy was 64.3% with the original metrics, and the best classification was 82.6% with the canonical variates in the case of each Java source file considered as one sample. When each compilation was treated as one sample, the classification accuracies were 66.2% with the original metrics and up to 87.0% with the canonical variates, respectively. Comparing the classification accuracies with those obtained for data sets C and D, it is evident that SDA retained almost all of the between-class variations in the data source.

In classifications with high accuracies, Java programs from the Internet shareware sources and from a fellow graduate student were all correctly assigned to their own classes. However, misclassification occurred frequently among the Java programs from computer science classes. The levels of the programmers taking the same computer
science course might be very similar. Moreover, some example pseudo codes may have been given by the instructors or in the respective textbooks. These factors would no doubt reduce the between-class variations and increase the difficulty of authorship identification. The results imply that the classification would be more efficient if the diversity of data sources increases.

Table 4.4 Classification accuracies (%) with selected metrics and derived canonical variates (CV) (as in Table 4.3, also for details on the data sets, see Table 4.2)

<table>
<thead>
<tr>
<th>Variables</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected metrics</td>
<td>62.7</td>
<td>67.2</td>
<td>62.6</td>
<td>66.6</td>
</tr>
<tr>
<td>CV1</td>
<td>19.8</td>
<td>17.7</td>
<td>18.6</td>
<td>16.1</td>
</tr>
<tr>
<td>CV1-2</td>
<td>41.0</td>
<td>39.0</td>
<td>41.3</td>
<td>38.2</td>
</tr>
<tr>
<td>CV1-3</td>
<td>52.1</td>
<td>51.0</td>
<td>54.3</td>
<td>54.1</td>
</tr>
<tr>
<td>CV1-4</td>
<td>57.5</td>
<td>60.6</td>
<td>57.3</td>
<td>61.9</td>
</tr>
<tr>
<td>CV1-5</td>
<td>64.1</td>
<td>67.7</td>
<td>65.1</td>
<td>66.2</td>
</tr>
<tr>
<td>CV1-6</td>
<td>66.4</td>
<td>72.1</td>
<td>69.0</td>
<td>72.6</td>
</tr>
<tr>
<td>CV1-7</td>
<td>70.7</td>
<td>76.1</td>
<td>71.5</td>
<td>75.8</td>
</tr>
<tr>
<td>CV1-8</td>
<td>71.6</td>
<td>76.8</td>
<td>70.7</td>
<td>78.1</td>
</tr>
<tr>
<td>CV1-9</td>
<td>71.7</td>
<td>78.7</td>
<td>72.9</td>
<td>81.6</td>
</tr>
<tr>
<td>CV1-10</td>
<td>70.6</td>
<td>79.5</td>
<td>73.7</td>
<td>82.0</td>
</tr>
<tr>
<td>CV1-11</td>
<td>70.8</td>
<td>78.2</td>
<td>72.3</td>
<td>84.4</td>
</tr>
<tr>
<td>CV1-12</td>
<td>72.6</td>
<td>80.2</td>
<td>74.5</td>
<td>85.2</td>
</tr>
<tr>
<td>CV1-13</td>
<td>76.0</td>
<td>81.1</td>
<td>79.2</td>
<td>84.4</td>
</tr>
<tr>
<td>CV1-14</td>
<td>76.5</td>
<td>80.2</td>
<td>79.7</td>
<td>85.8</td>
</tr>
<tr>
<td>CV1-15</td>
<td>76.9</td>
<td>82.0</td>
<td>79.6</td>
<td>85.8</td>
</tr>
<tr>
<td>CV1-16</td>
<td>77.1</td>
<td>81.2</td>
<td>80.0</td>
<td>85.4</td>
</tr>
<tr>
<td>CV1-17</td>
<td>77.6</td>
<td>81.6</td>
<td>77.8</td>
<td>84.9</td>
</tr>
<tr>
<td>CV1-18</td>
<td>77.0</td>
<td>81.6</td>
<td>78.2</td>
<td>84.4</td>
</tr>
<tr>
<td>CV1-19</td>
<td>77.0</td>
<td>82.5</td>
<td>78.4</td>
<td>84.9</td>
</tr>
<tr>
<td>CV1-20</td>
<td>76.9</td>
<td>81.9</td>
<td>78.8</td>
<td>83.9</td>
</tr>
</tbody>
</table>

Note: Classification accuracy was calculated as the percentage of correctly allocated samples.

To determine which metrics contributed more to the classification, the compositions of canonical variates were examined. Figure 4.1 displays the changes of
classification accuracy after a canonical variate was added to the classification model. The change represents the contribution of the added canonical variate to the classification because all canonical variates are orthogonal to one another. It was shown in Figure 4.1 that most of the contribution came from the first several (especially the first 5) canonical variates. This agrees with the results shown in Table 4.4 that the classification accuracies with the first 5 canonical variates exceeded those obtained with the original metrics.

![Figure 4.1 Contributions of individual canonical variates to the classification. A, B, C, and D refer to the data sets as described in Table 4.2. CVi represents the ith canonical variate. The numbers on y axis stand for the percentage changes of classification accuracy after adding a canonical variate into the discriminant analysis.](image)

Figures 4.2 and 4.3 show the canonical structure of the first five canonical variates for data sets A and C, respectively. The canonical structures were not be affected by the order how the metrics are listed in the figures. The canonical structure coefficients reveal how important each metric is for the canonical variate [SAS 1990]. Generally, the layout metrics played a more important role in the classifications than the style and
structure metrics. In the first canonical variate (CV1), the metric STY1c was very important. More metrics including STY1c, STY1e, STY1f, STY1g, STY1h, STY2a, STY2b, STY3, STY5, PRO2b, PSM5, PSM7e, and PSM7f were important for CV2. CV3 could be interpreted roughly as STY4 + STY5 while CV5 could be expressed as STY1g – STY1h. STY3 was important in CV4. However, contributions from the other metrics could not be neglected.

Similar canonical structures were revealed for data set B, C, and D. From the canonical structures, the effectiveness of the metrics selection process might be revealed. As discussed before (see Section 4.5), manual selection chose STY1b and STY1c for data set A but SDA selected STY1a and STY1c for data set C, among STY1a, STY1b, and STY1c. In canonical structures, as shown in Figures 4.2, STY1b was not important for the first essential canonical variates. In contrast, STY1a had much higher coefficients for the first several canonical variates of data set C, as shown in Figures 4.3. As a consequence, it is hard to conclude that selection of STY1a resulted in a better classification for data set C.
Figure 4.2 Canonical structures of the first five canonical variates (CV1, CV2, CV3, CV4, and CV5) for data set A. The canonical structure coefficients are correlation coefficients between the metrics and the canonical variates. The coefficients range from 1.0 to -1.0 and reveal how important each metric is. The higher the absolute value of the coefficient, the more important the metric is for the canonical variate [SAS 1990] (continued on next page).
Figure 4.2 (continued) Canonical structures of the first five canonical variates (CV1, CV2, CV3, CV4, and CV5) for data set A. The canonical structure coefficients are correlation coefficients between the metrics and the canonical variates. The coefficients range from 1.0 to -1.0 and reveal how important each metric is. The higher the absolute value of the coefficient, the more important the metric is for the canonical variate [SAS 1990].
Figure 4.3 Canonical structures of the first five canonical variates (CV1, CV2, CV3, CV4, and CV5) for data set C. The canonical structure coefficients are correlation coefficients between the metrics and the canonical variates. The coefficients range from 1.0 to -1.0 and reveal how important each metric is. The higher the absolute value of the coefficient, the more important the metric is for the canonical variate [SAS 1990] (continued on next page).
Figure 4.3 (continued) Canonical structures of the first five canonical variates (CV1, CV2, CV3, CV4, and CV5) for data set C. The canonical structure coefficients are correlation coefficients between the metrics and the canonical variates. The coefficients range from 1.0 to -1.0 and reveal how important each metric is. The higher the absolute value of the coefficient, the more important the metric is for the canonical variate [SAS 1990].
5.1 Summary

The thrust of this thesis work was to extract Java program metrics for authorship identification. Chapter I introduced the importance and possibility of authorship identification.

Chapter II reviewed the literature of authorship analyses. The reviewed aspects included target computer languages and metrics which were extracted for that purpose.

In Chapter III, the features of computing language Java were analyzed. A set of 56 metrics of Java programs was proposed for authorship analysis. Forty six groups of programs were diversely collected. The metric values were calculated with a computer program designed and implemented by the author.

In Chapter IV, the metric values were analyzed and the results were discussed. Metrics were selected for the classification purposes according to their significance. Classifications were performed on the extracted metrics with the derived canonical variates using a statistical procedure called canonical discriminant analysis. Classification accuracies were compared in aspects of sample sources and metric selection methods. To
explain the contribution of each metric to the classification, canonical structure coefficients were explored.

In conclusion, the authorship identification was effective and the extracted metrics were contributive. A set of metrics was proven contributive to the authorship identification. Among these metrics, outstanding metrics were STY1c (percentage of open braces (\{) that are the last character in a line), STY1g (average indentation in white spaces after open braces (\{)), STY1h (average indentation in tabs after open braces (\{)), STY3 (percentages of condition lines where the statements are on the same line as the condition), STY4 (average white spaces to the left side of operators), and STY5 (average white spaces to the right side of operators), etc. Thus, this study not only provided a set of contributive metrics but also added another approach of problem solving for authorship identification.

5.2 Future Work

Authorship identification is a very broad project. The work that was done in this thesis is a small part of it. Suggestions for future work include conducting a controlled experiment to test the extracted metrics using more programmers at more diverse programming levels and with more diverse experiences. Also more metrics could be extracted and tested for the classification models. Such metrics include classic measures such as McCabe's cyclomatic complexity metric [McCabe 1976].

Finally, much more work is expected to establish a powerful real system for authorship identification with a large database. After searching the database with a set of
metric values that are obtained from a program, the result will list a list of programmers in an order of likelihood of having written the program.
REFERENCES


## APPENDIX A

### GLOSSARY

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANOVA</td>
<td>Analysis of variance, a statistical procedure where a response or dependent variable is measured under experimental conditions identified by classification or independent variables [SAS 1990].</td>
</tr>
<tr>
<td>Author</td>
<td>One who designs, writes, and tests pieces of computer code.</td>
</tr>
<tr>
<td>Authorship Identification</td>
<td>Determining the likelihood of a particular author having written some pieces of code, usually based on other code samples from the same programmer.</td>
</tr>
<tr>
<td>CDA</td>
<td>Canonical Discriminant Analysis, a statistical procedure to find the linear combinations of the quantitative variables that best summarize the differences among the classes [SAS 1990].</td>
</tr>
<tr>
<td>CV</td>
<td>Canonical Variate, linear combination of the quantitative variables that summarizes the differences among the classes.</td>
</tr>
<tr>
<td>F Value</td>
<td>A statistical ratio of mean squares (variations) between groups to mean squares (variations) within groups.</td>
</tr>
<tr>
<td>IDENTIFIED</td>
<td>Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination, a metric extraction system developed by Gray and his colleagues [Gray et al. 1998].</td>
</tr>
<tr>
<td>LOC</td>
<td>Number of lines of code, a software metric.</td>
</tr>
<tr>
<td>MFC</td>
<td>Microsoft Foundation Class Library, a collection of classes that are written in C++ and can be used in building application programs. The MFC Library saves a programmer time by providing code that has already been written. MFC Library included classes for graphical user interface elements such as windows, frames, menus, tool bars, and status bars.</td>
</tr>
<tr>
<td>NCLOC</td>
<td>Number of non-comment lines of code, a software metric.</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open DataBase Connectivity, a standard method of sharing data between databases and other programs.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Programming</td>
<td>A distinctive or characteristic manner present in computer programs written by a programmer.</td>
</tr>
<tr>
<td>Style</td>
<td>Stepwise Discriminant Analysis, a statistical procedure to find a subset of quantitative variables that best reveals differences among the classes.</td>
</tr>
<tr>
<td>Software Forensics</td>
<td>An area of software science that aims at authorship analysis of computer source code. It is the use of measurement extracted from software source code or object code for legal or official purposes.</td>
</tr>
<tr>
<td>V(G)</td>
<td>McCabe's Cyclomatic Complexity Number [McCabe 1976].</td>
</tr>
</tbody>
</table>
CDA Processes

/*
Before the procedure, an Excel file containing the metrics values was imported as dataset_file. The CDA processes first perform a discriminant analysis with the metrics, and then discriminant analyses with the first 20 canonical variates that were derived from the metrics.
*/

proc discrim can data=dataset_file crossvalidate pool=yes crosslisterr out=output ncan=20;
class pn;
var /*variables for the analysis such as*/ STYlb STYld STYle STYlg STYlh;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3;
run;
proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 ;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 ;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 ;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 ;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 can10 ;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 can10 can11 ;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 can10 can11 can12 ;
run;

54
proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 can10 can11 can12 can13;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 can10 can11 can12 can13 can14;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 can10 can11 can12 can13 can14 can15;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 can10 can11 can12 can13 can14 can15 can16;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 can10 can11 can12 can13 can14 can15 can16 can17;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 can10 can11 can12 can13 can14 can15 can16 can17 can18;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 can10 can11 can12 can13 can14 can15 can16 can17 can18 can19;
run;

proc discrim data=output list crosslist method=normal;
class pn;
var can1 can2 can3 can4 can5 can6 can7 can8 can9 can10 can11 can12 can13 can14 can15 can16 can17 can18 can19 can20;
run;
/**
   Stepdisc is a procedure of variable selection. Variables depending on a class variables are
   selected according to their contributions to the variation among the classes.
   Stepforward (method==FW) add variables one by one into the variable set until no
   significant variables are available.
   Stepbackward (method==BW) selects all variables and then remove variables one by one
   from the set until no variable is not significant.
*/
proc stepdisc data=metric_dataset method=FW;
   class pn;
   var /*variables for the analysis such as*/ STY1b STY1d STY1e STY1g STY1h ;
run;
APPENDIX C

SELECTED PROGRAM LISTINGS OF THE WINDOWS APPLICATION FOR METRICS EXTRACTION

Follows list all C++ head files and source files in the implementation of the Window Application for the metrics extraction with Microsoft Visual Studio 6.0.

AddMetrics.h  AddMetrics.cpp
Indent.h  Indent.cpp
MainFrm.h  MainFrm.cpp
MetricsSet.h  MetricsSet.cpp
Resource.h  ThesisProject.rc
StdAfx.h  StdAfx.cpp
ThesisProject.h  ThesisProject.cpp
ThesisProjectDoc.h  ThesisProjectDoc.cpp
ThesisProjectView.h  ThesisProjectView.cpp

The total number of Head files (.h) is 9 and total number of Source files (.cpp) is 8. There are also four resource files and one readme file. The Total number of lines in the source files is approximately 5,000. The source code of ThesisProjectDoc.cpp that extracts metrics is listed as follows.

```cpp
// ThesisProjectDoc.cpp: implementation of the CThesisProjectDoc class
//deal with the extraction of metrics from text code.

#include "stdafx.h"
#include "ThesisProject.h"
#include <string.h>
#include "ThesisProjectDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

IMPLEMENT_DYNCREATE(CThesisProjectDoc, CDocument)

BEGIN_MESSAGE_MAP(CThesisProjectDoc, CDocument)
    ON_COMMAND(ID_DATABASE_ADD, OnDatabaseAdd)
END_MESSAGE_MAP()

// CThesisProjectDoc construction/destruction

CThesisProjectDoc::CThesisProjectDoc()
{
    // TODO: add one-time construction code here
```
fileName = "";
fileOpened = false;
loginId = "";
program_number = 0;
codeLines = 0;
blankLines = 0;
m_pMetricsSet = NULL;

//comment variables
pureComment = 0; /*lines begin with // or /*, only spaces are
allowed before it*/
inlineComment = 0; //lines of comment after a code statement
blockEndComment = 0; //a kind of inline comment, } }... or } */ ...
slashComment = 0; //lines of comment with //. this is to detn
preference of // or /** */
slashStarComment = 0; //lines of comment with /*
//curly bracket style counts
pureOpenCurly = 0; //line of open curly brackets alone a line
beginWithOpenCurly = 0; //lines begin with open curly brackets,
endWithOpenCurly = 0; //lines ending with open curly brackets,
after spaces ok */
pureCloseCurly = 0; //line of close curly brackets alone a line
beginWithCloseCurly = 0; //lines begin with close curly brackets,
spaces are allowed before. not inlude alone a line*/
endWithCloseCurly = 0; //lines ending with close curly
brackets, spaces are allowed after. not inlude alone a line*/
//indentation of lines just after open curly bracket
//indentation of line after if/else/while/for/switch/case without {
decisionSameLine = 0; //condition and statement code are at same
lines
decisionSeparateLine = 0; //
//operator separation spaces;
operatorCount = 0; /*number of operators, operators: +-/*%= and +=
- = /= %= and ==**/
beforeSpace = 0; //spaces before operator
afterSpace = 0; //spaces after operator
//variable and subroutine counts
variableCount = 0; //total variables declared
subroutineCount = 0; //number of subroutine count
//identifier length
variableNameLength = 0; //total length in characters.
functionNameLength = 0; /*total length in char. */
//character count, this is to detn preference of nameAfter or name_after
upcaseCount = 0; //upcase characters
lowercaseCount = 0; //lowercase
underscoreCount = 0; //underscore
dollarSignCount = 0; //underscore
//keywords count, structure preference
forCount = 0;
whileCount = 0;
doCount = 0; //percentage of above three can detn preference of
for while or do*/
ifCount = 0;
elseCount = 0;
switchCount = 0;
caseCount=0; // percentage of above detn preference of if else or switch case
// following keywords are based on code line number
staticCount = 0;
extendsCount = 0;
classCount = 0;
abstractCount = 0;
finalCount = 0;
implementsCount = 0;
importCount = 0;
instanceofCount = 0;
interfaceCount = 0;
nativeCount = 0;
newCount = 0;
packageCount = 0;
privateCount = 0;
protectedCount = 0;
publicCount = 0;
superCount = 0;
tryCount = 0;
catchCount = 0;
thisCount = 0;
throwCount = 0; // include throw and throws
slashStarFlag = false;
}
// destructor
CThesisProjectDoc::CThesisProjectDoc()
{
    delete m_pMetricsSet;
}

// for database
MetricsSet& CThesisProjectDoc::GetMetricsSet(void)
{
    return *m_pMetricsSet;
}

// receive a line and find the last open curly bracket
bool CThesisProjectDoc::findLastOpenCurly(const CString& cstr)
{
    // use a DFA to solve
    // input: {, /, *, space, others
    // states: 0-5 in which 1 and 4 are accepted states
    /*
     0 --> 0: space, * and others
     0 --> 1: {
     0 --> 2: /
     1 --> 1: space and {
     1 --> 3: /
     1 --> 5: * and others
     2 --> 1: {
     2 --> 5: / or *
     2 --> 0: space and others
     3 --> 1: {
     3 --> 4: / or *
     3 --> 5: space and others
     */
int i = 0, length = cstr.GetLength();
int state = 0, input = 0;
int table[6][6] = {{1,2,0,0,0,0},
{1,3,5,1,5,1},
{1,5,5,0,0,2},
{1,4,4,5,5,3},
{4,4,4,4,4,4},
{5,5,5,5,5,5}};
while (i<length) {
    if (cstr[i] == '{') input = 0;
    else if (cstr[i] == '/') input = 1;
    else if (cstr[i] == '*') input = 2;
    else if (cstr[i] == '\n') input = 3;
    else if (cstr[i] == '*') input = 4;
    else input = 5;
    state = table[state][input];
    i++;
}
if (state == 1 || state == 4) return true;
return false;

bool CThesisProjectDoc::findLastCloseCurly(const CString& cstr) {
    // use an DFA to solve
    // input: {, /, *, space, others
    // states: 0-5 in which 1 and 4 are accepted states
    int i = 0, length = cstr.GetLength();
    int state = 0, input = 0;
    int table[6][6] = {{1,2,0,0,0,0},
    {1,3,5,1,5,1},
    {1,5,5,0,0,2},
    {1,4,4,5,5,3},
    {4,4,4,4,4,4},
    {5,5,5,5,5,5}};
    while (i<length) {
        if (cstr[i] == '{') input = 0;
        else if (cstr[i] == '/') input = 1;
        else if (cstr[i] == '*') input = 2;
        else if (cstr[i] == '\n') input = 3;
        else if (cstr[i] == '*') input = 4;
        else input = 5;
        state = table[state][input];
        i++;
    }
    if (state == 1 || state == 4) return true;
    return false;
}
else if (cstr[i] == '/') input = 1;
else if (cstr[i] == '*') input = 2;
else if (cstr[i] == ' ') input = 3;
else if (cstr[i] == '\n') input = 4;
else input = 5;
state = table[state][input];
    i++;
}

if (state == 1 || state == 4) return true;
return false;

//open database here
BOOL CThesisProjectDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)
    if (m_pMetricsSet== NULL) {
        // Create the odbc set.
        m_pMetricsSet = new MetricsSet(&ffi_DB);
        // Get the default connection string
        CString connectStr = m_pMetricsSet->GetDefaultConnect();
        // Open the database.
        m_DB.Open(NULL, FALSE, FALSE, connectStr, FALSE);
        // Open the ODBC set
        m_pMetricsSet->Open();
    }
    return TRUE;
}

// CThesisProjectDoc serialization
void CThesisProjectDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}

// CThesisProjectDoc diagnostics
#ifdef _DEBUG
void CThesisProjectDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CThesisProjectDoc::Dump(CDumpContext& dc) const
{
//open a source file and get metrics
BOOL CThesisProjectDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
    if (!CDocument::OnOpenDocument(lpszPathName))
        return FALSE;
    fileName = lpszPathName; // copy file name

    // reset metrics to default values
    fileText.clear();
    loginId = "";
    program_number = 0;
    codeLines = 0;
    blankLines = 0;
    pureComment = 0; /* lines begin with // or /*, only spaces are allowed before it */
    inlineComment = 0; /* lines of comment after a code statement */
    blockEndComment = 0; // a kind of inline comment, }/* ... or */ /*...
    slashComment = 0; /* lines of comment with //, this is to deter
    preference of // or /* */
    slashStarComment = 0; /* lines of comment with */

    // curly bracket style counts
    pureOpenCurly = 0; // line of open curly brackets alone a line
    beginWithOpenCurly = 0; // lines begin with open curly brackets,
    endWithOpenCurly = 0; /* lines ending with open curly
    brackets, spaces are allowed after. not include alone a line*/
    /* spaces are allowed before. not include alone a line, use trimming space*/
    pureCloseCurly = 0; // line of close curly brackets alone a line
    beginWithCloseCurly = 0; // lines begin with close curly brackets,
    endWithCloseCurly = 0; /* lines ending with close curly
    brackets, spaces are allowed after. not include alone a line*/
    /* indentation of lines just after open curly bracket */

    // indentation of line after if/else/while/for/switch/case without {
    decisionSameLine = 0; /* condition and statement code are at same lines*/
    decisionSeparateLine = 0; //

    // operator separation spaces;
    operatorCount = 0; /* number of operators, operators: +-*/ /= and +=
    -= /= %= and %= */
    beforeSpace = 0; // spaces before operator
    afterSpace = 0; // spaces after operator

    // variable and subroutine counts
    variableCount = 0; /* total variables declared */
    subroutineCount = 0; /* number of subroutine count */

    // identifier length
    variableNameLength = 0; /* total length in characters */
    functionNameLength = 0; /* total length in characters */

    // character count. this is to deter preference of nameAfter or name_after
    upcaseCount = 0; /* upcase characters */
    lowercaseCount = 0; /* lowercase */

}
underscoreCount = 0; // underscore
dollarSignCount = 0; // underscore
// keywords count, structure preference
forCount = 0;
whileCount = 0;
doCount = 0; /* percentage of above three can detn preference of
for while or do */
ifCount = 0;
elseCount = 0;
switchCount = 0;
caseCount = 0; /* percentage of above detn preference of if else or
switch case */
// following keywords are based on code line number
staticCount = 0;
extendsCount = 0;
classCount = 0;
abstractCount = 0;
finalCount = 0;
implementsCount = 0;
importCount = 0;
instanceofCount = 0;
interfaceCount = 0;
nativeCount = 0;
newCount = 0;
packageCount = 0;
privateCount = 0;
protectedCount = 0;
publicCount = 0;
superCount = 0;
tryCount = 0;
catchCount = 0;
thisCount = 0;
throwCount = 0; /* include throw and throws
slashStarFlag = false;
extractMetricsFlag = false;
return TRUE;
}
// extract metrics and called by openDocument
void CThesisProjectDoc::extractMetrics(CString& fileName)
{
    // open and read file
    ifstream in(fileName, ios::in|ios::nocreate);
    char str[512]; // name[256], current[256];
    CString cstr;
    if (in.bad()) /* give a message if file error */
        AfxGetMainWnd()->MessageBox("Input file error", "Error",
        MB_OK|MB_ICONEXCLAMATION);
    return;
}
fileOpened = true;
while (in) {
in.getline(str, 512);
cstr.Format("%s", str);
fileText.push_back(cstr);
CString cstr1(cstr);
cstr.TrimRight();
cstr.TrimLeft();
if (cstr.GetLength()==0) {
    blankLines++;
    continue;
}
else codeLines++;

// for indentation
int k = findOpenCurly(cstr);
if (k>0)
    addIndentMem(cstr1, findOpenCurly(cstr));
else if (k<0) // delete indent from memory
    for (int n=0; n>k; n--)
        indentMem.pop_back();

if (!indentMem.empty()) addIndent(cstr1);

// condition word count and whether same line with statement
conditionCount(cstr);

// operatorSpace: before and after operators
operatorSpace(cstr);

// variable counts and length
variableHandle(cstr);

// upper, lower, underscore case count
caseHandle(cstr);
keywordHandle(cstr);

// for curly styles
if (cstr.Compare("\n\}==0) pureOpenCurly++;
else if (cstr.Compare("\n\})==0) pureCloseCurly++;
else {
    if (cstr[0] == '{') beginWithOpenCurly++;
    else if (findLastOpenCurly(cstr)) endWithOpenCurly++;

    // such as . . . / /, but not include pure curly lines
    if (cstr[0] == '}') beginWithCloseCurly++;
    else if (findLastCloseCurly(cstr))
        endWithCloseCurly++;
}

if (cstr.GetLength() > 1) {
    if (slashStarFlag) pureComment++;
    else if (cstr[0] == '/' && cstr[1] == '/')
        pureComment++;
    else if (cstr[0] == '/' && cstr[1] == '*')
        pureComment++;
    else if (findSlashSlash(cstr))
        inlineComment++;
    else if (findSlashStar(cstr)) {
        inlineComment++;
    }
}

// find after */
else slashStarFlag = true;
else if (findSlashSlash(cstr))
    { inlineComment++; slashComment++; }// find first
else if (findSlashStar(cstr)) {
    inlineComment++; slashStarComment++;
    if (!findStarSlash(cstr)) slashStarFlag = false;
}

// find after */
else slashStarFlag = false;
} // find first
} // end of if
// take care of */ and */
flagSlashStar(cstr);
} // end of while
in.close();

} // find /* style
bool CThesisProjectDoc::findSlashStar(CString& cstr)
{
    // if // is return */ and */ then is false
    int i = cstr.GetLength();
    int j = 0;
    bool slashStar = false;
    while (j < i - 1) {
        if (cstr[j] == '/' && cstr[j + 1] == '*') slashStar = true;
        else if (slashStar && cstr[j] == '/' && cstr[j + 1] == '*')
            slashStar = false;
        else if (((!slashStar) && cstr[j] == '/' && cstr[j + 1] == '/'))
            return true;
        j++;
    }
    return false;
}

} // find /* style
bool CThesisProjectDoc::findStarSlash(CString& cstr)
{
    // if // is return */ and */ then is false
    int i = cstr.GetLength();
    int j = 0;
    while (j < i - 1) {
        if (cstr[j] == '/' && cstr[j + 1] == '*') return true;
        else if (cstr[j] == '/' && cstr[j + 1] == '/') return false;
        j++;
    }
    return false;
}

} // find */
bool CThesisProjectDoc::findStarSlash(CString& cstr)
{
    // if // is return */ and */ then is false
    int i = cstr.GetLength();
    int j = 0;
    while (j < i - 1) {
        if (cstr[j] == '*' && cstr[j + 1] == '/') return true;
        else if (cstr[j] == '/' && cstr[j + 1] == '/') return false;
        j++;
    }
    return false;
}
//find {
int CThesisProjectDoc::findOpenCurly(const CString& cstr)
{
    bool slashStar = slashStarFlag, slsl = false;
    int open = 0, close = 0;
    int i = cstr.GetLength();
    int j = 0;
    while (j < i - 1) {
        if (cstr[j] == '{' && (!slashStar)) open++;
        if (cstr[j] == '}', && (!slashStar)) close++;
        if (cstr[j] == '*' && cstr[j+1] == '/')
            if (slashStar) slashStar = false;
        else if (cstr[j] == '/' && cstr[j+1] == '*')
            if (!slashStarFlag) slashStarFlag = true;
        else if (cstr[j] == '{' && !slsl)
            slsl = true; break;
        j++;
    }
    if (cstr[i-1] == '{' && !slsl) open++; //last char is {
    else if (cstr[i-1] == '}' && !slsl) close++; //last char is }
    return (open - close);
}

//remember indentation
void CThesisProjectDoc::addIndentMem(const CString& cstr, int k)
{
    if (k<0) return;
    int space=0, tab=0;
    int i = cstr.GetLength();
    int j = 0;
    while (j < i) {
        if (cstr[j] == ' ') space++;
        else if (cstr[j] == '	') tab++;
        else break;
        j++;
    }
    indentStructure ind;
    ind.indentSpace = space;
    ind.indentTab = tab;
    for (j= 0; j<k; j++)
        indentMem.push_back(ind);
}

//add indentation
void CThesisProjectDoc::addIndent(const CString& cstr)
{
    int space=0, tab=0;
    int i = cstr.GetLength();
    int j = 0;
    while (j < i) {
        if (cstr[j] == ' ') space++;
        else if (cstr[j] == '	') tab++;
        else break;
        j++;
    }
    indents.indentSpace += space-indentMem.back().indentSpace;
    indents.indentTab += tab-indentMem.back().indentTab;
// set flag of /* status
void CThesisProjectDoc::flagSlashStar(const CString& cstr)
{
    int i = cstr.GetLength();
    int j = 0;
    while (j < i - 1) {
        if (cstr[j] == '*' && cstr[j + 1] == '/')
            if (slashStarFlag) slashStarFlag = false;
        else if (cstr[j] == '/' && cstr[j + 1] == '*')
            if (!slashStarFlag) slashStarFlag = true;
        else if (cstr[j] == '/' && cstr[j + 1] == '/')
            return;
        j++;
    }
}

// count condition statements
void CThesisProjectDoc::conditionCount(const CString& cstr)
{
    bool forFlag = false, otherFlag = false;
    int i = cstr.GetLength();
    int j = 0;
    if (i >= 2 && cstr[0] == 'f' && cstr[1] == 'f')
        {ifCount++; otherFlag=true;}
    else if (i >= 2 && cstr[0] == 'd' && cstr[1] == 'f')
        {docCount++; otherFlag=true;}
    else if (i >= 3 && cstr[0] == 'f' && cstr[1] == '0' && cstr[2] == 'r')
        {forCount++; otherFlag=true;}
        {elseCount++; otherFlag=true;}
        {caseCount++; otherFlag=true;}
        {whileCount++; otherFlag=true;}
        {switchCount++; otherFlag=true;}
    if (otherFlag && (findSemicolon(cstr) > 0))
        decisionSameLine++;
    else if (forFlag && (findSemicolon(cstr) > 2))
        decisionSameLine++;
    else if (forFlag || otherFlag)
        decisionSeparateLine++;
}

// find ; for variable count
int CThesisProjectDoc::findSemicolon(const CString& cstr)
{
    int semic = 0;
    bool slashStar = false;
    int i = cstr.GetLength();
}
int j = 0;
while (j < i-1) {
    if (cstr[j] == ';') semic++;
    else if (cstr[j]=='/' && cstr[j+1] == '*')
        break;
    else if (cstr[j]=='/' && cstr[j+1] == '/') break;
    j++;
}
if (cstr[i-1] == ';') semic++;
return semic;

//operators: +-*%= and += -= *= /= %= and ==
void CThesisProjectDoc::operatorSpace(const CString& cstr)
{
    if (slashStarFlag) return;
    bool after = false;
    int i = cstr.GetLength();
    int j = 0, k;
    while (j < i) {
        if (j<i-l && cstr[j] == '/' && (cstr[j+l] == '*' ||
            cstr[j+1] == 'I')}
            break;  //in case begin with
        if (cstr[j] == '+' || cstr[j] == '-' || cstr[j] == '*'
             || cstr[j] == '%'
             || cstr[j] == '=') {
            if (findOperants(cstr, j)) {
                operatorCount++;
                for (k = j-1; k>=O; k--) {
                    if (cstr[k]==' ') beforeSpace++;//before spaces
                    else break;
                }
                if (j<i-l && cstr[j+1]=='=') j++;//jump over =,
                    for (k = j+1; k<i; k++) {
                        if (cstr[k]==' ') afterSpace++;
                    else break;
                }
            }
        } //if
        j++;
    } //while

    //find operants
    bool CThesisProjectDoc::findOperants (const CString& cstr, int k)
    {
        int left = k-1;
        int right = k+1;
        int len = cstr.GetLength();
        bool bstring = false;
        if (len>k+1 && cstr[k+1] == 'I') right++;  //for +=, *=, etc
        while (left>=0 && cstr[left] == 'I')
            left--;
        if (left>=0 && isalpha(cstr[left]) && isdigit(cstr[left])
            && cstr[left] != 'I' && cstr[left] != '$' &&
            cstr[left] != '\' && cstr[left] != '\' )
            return false;
        if (left>=0 && cstr[left] == '\' ) //left is a string
            bstring = true;
        while (left>=0 && !bstring ) { //check first char
if (cstr[left] == ' ') break;
if (cstr[left] == '
' || cstr[left] == '/' || cstr[left] == '|')
    return false;
left--;
}
bstring = false; //reset
while (right<len && cstr[right] == ' ')
    right++;
//test first char
if (right<len && !isalpha(cstr[right]) && !isdigit(cstr[right])
    && cstr[right] != '=' && cstr[right] != '(' && cstr[right] != '
' && cstr[right] != '.')
    return false;
if (right<len && cstr[right] == '"
')
    bstring = true;
while (right<len && !bstring)
    { //check first char
        if (cstr[right] == ' ') break;
        if (cstr[right] == '\' || cstr[right] == '/')
            return false;
        right++;
    }

int quote = 0;
left = k-1;
while (left>=0)
    { if (cstr[left--]=='"') quote++;
    }
if (quote%2) return false;
return true;

//count case of letters
void CThesisProjectDoc::caseHandle(const CString& cstr)
{
    int i = cstr.GetLength();
    int j = 0;
    while (j < i) {
        if (cstr[j] == '_') underscoreCount++;
        else if (cstr[j] == '$') dollarSignCount++;
        else if (isupper(cstr[j])) upcaseCount++;
        else if (islower(cstr[j])) lowercaseCount++;
        j++;
    }
}

//count keywords
void CThesisProjectDoc::keywordHandle(const CString& cstr)
{
    char str[512];
    char *token;
    strcpy(str, cstr); //copy the cstring to a char array
    char seps[] = " ,{}());";
    token = strtok( str, seps ); //get token
    while( token != NULL )
    {

if (strcmp(token, "static") == 0) staticCount++;
else if (strcmp(token, "extends") == 0) extendsCount++;
else if (strcmp(token, "class") == 0) classCount++;
else if (strcmp(token, "abstract") == 0) abstractCount++;
else if (strcmp(token, "final") == 0) finalCount++;
else if (strcmp(token, "implements") == 0) implementsCount++;
else if (strcmp(token, "import") == 0) importCount++;
else if (strcmp(token, "instanceof") == 0) instanceofCount++;
else if (strcmp(token, "interface") == 0) interfaceCount++;
else if (strcmp(token, "native") == 0) nativeCount++;
else if (strcmp(token, "new") == 0) newCount++;
else if (strcmp(token, "package") == 0) packageCount++;
else if (strcmp(token, "private") == 0) privateCount++;
else if (strcmp(token, "protected") == 0) protectedCount++;
else if (strcmp(token, "public") == 0) publicCount++;
else if (strcmp(token, "super") == 0) superCount++;
else if (strcmp(token, "try") == 0) tryCount++;
else if (strcmp(token, "catch") == 0) catchCount++;
else if (strcmp(token, "this") == 0) thisCount++;
else if (strcmp(token, "throws") == 0) throwCount++;

// Get next token:
    token = strtok(NULL, seps);
}

void CThesisProjectDoc::variableHandle(char* str)
{
    bool type = false, openBracket = false, closeBracket = false,
        variable = false, equal = false;
    bool openCurly = false, closeCurly = false;
    int isIdentifier = 0;
    char var[128];
    string temp;
    temp = nextIdentifier(str, isIdentifier);
    if (!isIdentifier) return; // not variables
    else if (type & isIdentifier & (strcmp(temp.c_str(), "int") == 0
            || strcmp(temp.c_str(), "short") == 0
            || strcmp(temp.c_str(), "long") == 0
            || strcmp(temp.c_str(), "float") == 0
            || strcmp(temp.c_str(), "double") == 0
            || strcmp(temp.c_str(), "byte") == 0
            || strcmp(temp.c_str(), "char") == 0
            || strcmp(temp.c_str(), "String") == 0)
    {
        type = true;
        isIdentifier = 0;
    }
    else if (type & (strcmp(temp.c_str(), "static") == 0
            || strcmp(temp.c_str(), "final") == 0
            || strcmp(temp.c_str(), "private") == 0
            || strcmp(temp.c_str(), "public") == 0));
    else return;

    while (temp.size() != 0)
    {
        temp = nextIdentifier(str, isIdentifier);
        if (temp.size() == 0) return;
    }
if (!type && (strcmp(temp.c_str(), "static")==0 ||
strcmp(temp.c_str(), "final")==0 || strcmp(temp.c_str(), "private")==0 ||
strcmp(temp.c_str(), "protected")==0 || strcmp(temp.c_str(), "public")==0) )
    continue;

if (!type && isIdentifier && (strcmp(temp.c_str(), "int")==0 ||
   strcmp(temp.c_str(), "short")==0 ||
   strcmp(temp.c_str(), "long")==0 ||
   strcmp(temp.c_str(), "float")==0 ||
   strcmp(temp.c_str(), "double")==0 ||
   strcmp(temp.c_str(), "boolean")==0 ||
   strcmp(temp.c_str(), "byte")==0 ||
   strcmp(temp.c_str(), "char")==0 ||
   strcmp(temp.c_str(), "String")==0 ) )
    { type = true; isIdentifier = 0; }
else if (isIdentifier && type) {
    strcpy(var, temp.c_str()); // copy the last identifier
    variable = true;
    isIdentifier = 0;
}
else if (strcmp(temp.c_str(), "/="/==0 ) return;
else if (strcmp(temp.c_str(), "*/"==0 ) return;
else if (strcmp(temp.c_str(), ")="/==0 && variable &&
openBracket) = true;
else if ( (strcmp(temp.c_str(), ",="/)==0 && type &&
openBracket) = true;
else if ( (strcmp(temp.c_str(), ",="/)==0 && variable &&
!openBracket && equal && !openCurly)
    equal = false;
else if ( (strcmp(temp.c_str(), ",="/)==0 && variable &&
!openBracket && equal && !openCurly)
    addVariable(var);
else if ( (strcmp(temp.c_str(), ",="/)==0 && variable &&
!openBracket && equal && !openCurly) { 
    addVariable(var);
    type = false; variable = false; 
    equal = false;
}
else if ( (strcmp(temp.c_str(), ",="/)==0 && variable &&
!openBracket) {
    equal = true;
    addVariable(var);
}
else if ( (strcmp(temp.c_str(), ",="/)==0 && variable &&
!openBracket) openCurly = true;
else if ( (strcmp(temp.c_str(), ",="/)==0 && variable &&
openBracket) openCurly = false;
else if ( strcmp(temp.c_str(), ",="/)==0 && variable)
return;


```cpp
else {
   type = false; variable = false;
   equal = false;
}

}//end of while

//identify and add a variable to count

void CThesisProjectDoc::addVariable(char* str)
{
   CString cstr;
   cstr.Format("%s", str);
   cstr.TrimLeft();
   cstr.TrimRight();
   variableCount++;
   variableNameLength += cstr.GetLength();
}

// This function return the char * of next identifier

string CThesisProjectDoc::nextIdentifier(char* str, int& iSIdentifier)
{
   string strg(str);
   int length = 0, blank = 0;
   char *word;
   while (*str == ',' || *str == '\t' || *str == '\n')
   {
      str++; blank++;
   }
   word = str; //pointing to the beginning of char array
   if (isalpha(* (str+length)) || (* (str+length) == '$') ||
       (* (str+length) == '\') ||
       (* (str+length) == '/') || (isalpha(*(str+length)) ||
       (isdigit(*(str+length)) ||
       (*(str+length) == '\') || *(str+length) == '/') ||
       (*(str+length) == '\') || *(str+length) == '/') ||
       (*(str+length) == '/') || *(str+length) == '/') ||
       (*(str+length) == '/') ||
   }
   isIdentifier = 1;
   else if (*(str+length)=='/') {
      length++;
      if (*(str+length)=='/' || *(str+length)=='*')
         length++;
   }
   else {
      length++;
   }

   word += length-1;
   *word++ = '\0';
   strcpy(str, word); /*this is necessary since str is cut by just
   above statement*/
   if (strg.length()==blank)
      return strg.substr(0, 0);
   return strg.substr(blank, length);
}
```
// for menu of add database
void CThesisProjectDoc::OnDatabaseAdd()
{
    // TODO: Add your command handler code here
    CAddMetrics Dialog;
    if (Dialog.DoModal() == IDOK) {
        if (((loginId.GetLength() == 0) || (program_number == 0))
            AfxGetMainWnd() ->MessageBox( "Please Specify login ID and Program number first", "Alert!", MB_OK | MB_ICONEXCLAMATION);
        else AddMetrics();
    }
}

// add metrics to database
void CThesisProjectDoc::AddMetrics()
{
    try {
        m_pMetricsSet->AddNew();
        m_pMetricsSet->m_blankLines = blankLines;
        m_pMetricsSet->m_codeLines = codeLines;
        m_pMetricsSet->m_loginId = loginId;
        m_pMetricsSet->m_Program_Number = program_number;
        m_pMetricsSet->m_abstractCount = abstractCount;
        m_pMetricsSet->m_afterSpace = afterSpace;
        m_pMetricsSet->m_beforeSpace = beforeSpace;
        m_pMetricsSet->m_beginWithCloseCurly = beginWithCloseCurly;
        m_pMetricsSet->m_beginWithOpenCurly = beginWithOpenCurly;
        m_pMetricsSet->m_caseCount = caseCount;
        m_pMetricsSet->m_classCount = classCount;
        m_pMetricsSet->m_decisionSameLine = decisionSameLine;
        m_pMetricsSet->m_decisionSeparateLine = decisionSeparateLine;
        m_pMetricsSet->m_doCount = doCount;
        m_pMetricsSet->m_dollarSignCount = dollarSignCount;
        m_pMetricsSet->m_elseCount = elseCount;
        m_pMetricsSet->m_endWithCloseCurly = endWithCloseCurly;
        m_pMetricsSet->m_endWithOpenCurly = endWithOpenCurly;
        m_pMetricsSet->m_extendsCount = extendsCount;
        m_pMetricsSet->m_finalCount = finalCount;
        m_pMetricsSet->m_forCount = forCount;
        m_pMetricsSet->m_ifCount = ifCount;
        m_pMetricsSet->m_implementsCount = implementsCount;
        m_pMetricsSet->m_importCount = importCount;
        m_pMetricsSet->m_inlineComment = inlineComment;
        m_pMetricsSet->m_instanceofCount = instanceofCount;
        m_pMetricsSet->m_interfaceCount = interfaceCount;
        m_pMetricsSet->m_lowercaseCount = lowercaseCount;
        m_pMetricsSet->m_nativeCount = nativeCount;
        m_pMetricsSet->m_newCount = newCount;
        m_pMetricsSet->m_operatorCount = operatorCount;
        m_pMetricsSet->m_packageCount = packageCount;
        m_pMetricsSet->m_privateCount = privateCount;
        m_pMetricsSet->m_protectedCount = protectedCount;
        m_pMetricsSet->m_publicCount = publicCount;
        m_pMetricsSet->m_pureCloseCurly = pureCloseCurly;
        m_pMetricsSet->m_pureComment = pureComment;
    }
}
m_pMetricsSet->m_pureOpenCurly = pureOpenCurly;
m_pMetricsSet->m_slashComment = slashComment;
m_pMetricsSet->m_slashStarComment = slashStarComment;
m_pMetricsSet->m_staticCount = staticCount;
m_pMetricsSet->m_superCount = superCount;
m_pMetricsSet->m_switchCount = switchCount;
m_pMetricsSet->m_thisCount = thisCount;
m_pMetricsSet->m_throwCount = throwCount;
m_pMetricsSet->m_tryCount = tryCount;
m_pMetricsSet->m_underscoreCount = underscoreCount;
m_pMetricsSet->m_upcaseCount = upcaseCount;
m_pMetricsSet->m_variableCount = variableCount;
m_pMetricsSet->m_variableNameLength = variableNameLength;
m_pMetricsSet->m_whileCount = whileCount;
if (m_pMetricsSet->CanUpdate())
    m_pMetricsSet->Update();
m_pMetricsSet->Requery();
} // end of try
catch(CDBException *theException) {
    AfxMessageBox(theException->m_strError);
} // end of catch
} // end of method
VITA

Haibiao Ding

Candidate for the Degree of

Master of Science

Thesis: EXTRACTION OF JAVA PROGRAM FINGERPRINTS FOR SOFTWARE

AUTHORSHIP IDENTIFICATION

Major Field: Computer Science

Biographical:

Education: Graduated from Fengcheng High School, Jiangxi Province, P. R. China, in July 1985; received Bachelor of Science in Animal Science from Central China Agricultural University in June 1989; received Master of Science in Food Science from Central China Agricultural University in June 1992; Completed the requirements for the Master of Science degree in Computer Science at the Computer Science Department at Oklahoma State University in August 2002.

Experience: Employed as a Lecturer in Central China Agricultural University from July 1992 to July 1999; graduate research assistant and teaching assistant at Oklahoma State University from January 2001 to July 2002.