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:

[illegible]

Thesis Adviser

J Chandler

[illegible]

[illegible]

Dean of the Graduate College

PREFACE

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

LIST OF TABLES

LIST OF FIGURES

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 includes 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.

CHAPTER II

LITERATURE REVIEW

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]. Homles [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 al 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

- Code formatting
- Commenting style
- Variable naming convention
- Spelling and grammar of comments
- Use of language features
- Sizes of routines
- Errors
- Code reuse

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

- Multiple statements per line
- Blank lines in program body

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]

Metric	Description
STY1	A list of metrics indicating indentation style inside functions
STY1a	Indentation of C statements within surrounding blocks
STY1b	Percentage of open braces ({) that are along a line
STY1c	Percentage of open braces ({) that are the first character in a line
STY1d	Percentage of open braces ({) that are the last character in a line
STY1e	Percentage of close braces (}) that are along a line
STY1f	Percentage of close braces (}) that are the first character in a line
STY1g	Percentage of close braces (}) that are the last character in a line
STY1h	Indentation of open braces ({)
STY1i	Indentation of close braces (})
STY2	Indentation of statements starting with the "else" keyword
STY3	The use of a separator between function names and parameter lists
STY4	The use of a separator between function return type and function name
STY5	A vector of metrics specifying comment style
STY5a	Use of borders to highlight comments
STY5b	Percentages of code lines with inline comments
STY5c	Ratio of lines of block style comments to lines of code
STY6	Ratio of white lines to lines of code

Table 2.2 Programming style metrics for C programs [Krsul and Spafford 1996]

Metric	Description
PRO1	Mean program line length in terms of characters
PRO2	A vector of metrics of name lengths
PRO2a	Mean local variable name length
PRO2b	Mean global variable name length
PRO2c	Mean function name length
PRO2d	Mean function parameter name length
PRO3	A list of metrics of name convention
PRO3a	Percentage of variable names that start with an uppercase letter
PRO3b	Percentage of function names that start with an uppercase letter
PRO3c	Is the underscore character used in names?
PRO4	Ratio of global variable count to local variable count
PRO5	Ratio of global variable count to lines of code
PRO6	Preference of either "while", "for", or "do" loops
PRO7	Are comments merely an echo of the code

Table 2.3 Programming structure metrics for C programs [Krsul and Spafford 1996]

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

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

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

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]

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

CHAPTER III

FEATURE ANALYSIS AND METRICS EXTRACTION

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:

- ftp://ftp.ai.mit.edu/pub/caroma/tools/1999-07-05/doc/edu/mit/ai/psg/traveler/examples/
- http://www.devx.com/sourcebank/search.asp
- http://math.hws.edu/eck/
- ftp://ftp.fct.unl.pt/.2/documents/published/oreilly/nutshell/java/examples
- ftp://ftp.ccl.net/pub/chemistry/software/SOURCES/JAVA

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 Appredix 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

Metric	Description
STY1	A list of metrics indicating indentation style
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
STY1g	Average indentation in white spaces after open braces ({)
STY1h	Average indentation in tabs after open braces ({)
STY2	A vector of metrics specifying comment style
STY2a	Percentages of pure comment lines among lines containing comments
STY2b	Percentages of "//" style comments among "//" and "/*" style comments
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 #
STY5	Average white spaces to the right side of operators #
STY6	Ratio of blank lines to code lines (including comment lines)
STY7	Ratio of comment lines to non-comment lines*
STY8	Ratio of code lines containing comment to code lines without any comments

#: 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

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

*: 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

Metric	Description
PSM1	Average non-comment lines per class/interface
PSM2	Average number of primitive variables per class/interface
PSM3	Average number of functions per class/interface
PSM4	Ratio of interfaces to classes
PSM5	Ratio of primitive variable count to lines of non-comment code
PSM6	Ratio of function count to lines of non-comment code
PSM7	A list of ratios of keywords to lines of non-comment code
PSM7a	Ratio of keyword "static" to lines of non-comment code
PSM7b	Ratio of keyword "extends" to lines of non-comment code
PSM7c	Ratio of keyword "class" to lines of non-comment code
PSM7d	Ratio of keyword "abstract" to lines of non-comment code
PSM7e	Ratio of keyword "implements" to lines of non-comment code
PSM7f	Ratio of keyword "import" to lines of non-comment code
PSM7g	Ratio of keyword "instanceof" to lines of non-comment code
PSM7h	Ratio of keyword "interface" to lines of non-comment code
PSM7I	Ratio of keyword "native" to lines of non-comment code
PSM7j	Ratio of keyword "new" to lines of non-comment code
PSM7k	Ratio of keyword "package" to lines of non-comment code
PSM7l	Ratio of keyword "private" to lines of non-comment code
PSM7m	Ratio of keyword "public" to lines of non-comment code
PSM7n	Ratio of keyword "protected" to lines of non-comment code
PSM7o	Ratio of keyword "this" to lines of non-comment code
PSM7p	Ratio of keyword "super" to lines of non-comment code
PSM7q	Ratio of keyword "try" to lines of non-comment code
PSM7r	Ratio of keyword "throw" to lines of non-comment code
PSM7s	Ratio of keyword "catch" to lines of non-comment code
PSM7t	Ratio of keyword "final" to lines of non-comment code

CHAPTER IV

DATA ANALYSIS, RESULTS, AND DISCUSSION

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 4.1 Derivation metrics sets

Metrics derivation set	Metrics and derivation formula
1	STY1a + STY1b + STY1c = 1
2	STY1d + STY1e + STY1f = 1
3	PRO3a + PRO3b + PRO3c + PRO3d = 1
4	PRO4a + PRO4b + PRO4c = 1
5	PRO5a + PRO5b = 1

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
PRO3a: Percentage of uppercase characters
PRO3b: Percentage of lowercase characters
PRO3c: Percentage of underscores
PRO3d: Percentage of dollar signs
PRO4a: Percentage of "while" in total of "while", "for", and "do"
PRO4b: Percentage of "for" in total of "while", "for", and "do"
PRO4c: Percentage of "do" in total of "while", "for", and "do"
PRO5a: Percentage of "if" and "else" in total of "if", "else", "switch", and "case"
PRO5b: 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

Sample Source	Variable Selection Method	
	Manually	Stepwise Discriminant Analysis
Each Java source file as a sample	A	C
All Java source files for each compilation as a sample	B	D

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)

	Data Treatments			
Metrics	A	B	C	D
FPC [#]		+		+
STY1a			+	+
STY1b	+	+		
STY1c	+	+	+	+
STY1d				+
STY1e	+	+	+	+
STY1f	+	+	+	
STY1g	+	+	+	+
STY1h	+	+	+	+
STY2a	+	+	+	+
STY2b	+	+	+	+
STY3	+	+	+	+
STY4	+	+	+	+
STY5	+	+	+	+
STY6	+	+	+	+
STY7	+	+	+	+
STY8	+	+	+	+
PRO1	+	+	+	+
PRO2a	+	+		+
PRO2b	+	+	+	+
PRO3a	+	+		+
PRO3b	+	+	+	+
PRO3c				
PRO3d	+	+	+	+
PRO4a			+	
PRO4b				+
PRO4c	+	+	+	+
PRO5a			+	
PRO5b				+
PRO5c	+	+	+	+
PRO5d*				
PSM1	+	+	+	+
PSM2				+
PSM3	+	+	+	+
PSM4	+		+	
PSM5	+		+	
PSM6	+	+	+	+
PSM7a			+	+
PSM7b	+	+	+	+
PSM7c	+	+	+	+
PSM7d	+	+	+	+

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)

	Data Treatments			
Metrics	A	B	C	D
PSM7e	+	+	+	+
PSM7f	+	+	+	+
PSM7g	+	+	+	+
PSM7h	+	+	+	+
PSM7i*				
PSM7j	+	+	+	+
PSM7k	+	+	+	+
PSM7l	+	+	+	+
PSM7m	+	+	+	+
PSM7n	+		+	+
PSM7o	+	+	+	+
PSM7p	+	+	+	+
PSM7q	+	+	+	+
PSM7r	+	+	+	+
PSM7s	+	+	+	+
PSM7t	+		+	+
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 discrimiant 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 20^{th} 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)

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

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.

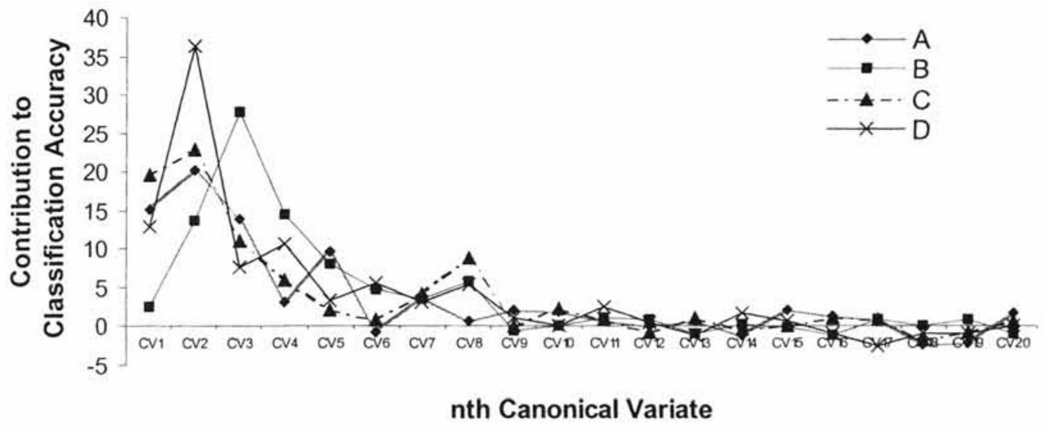
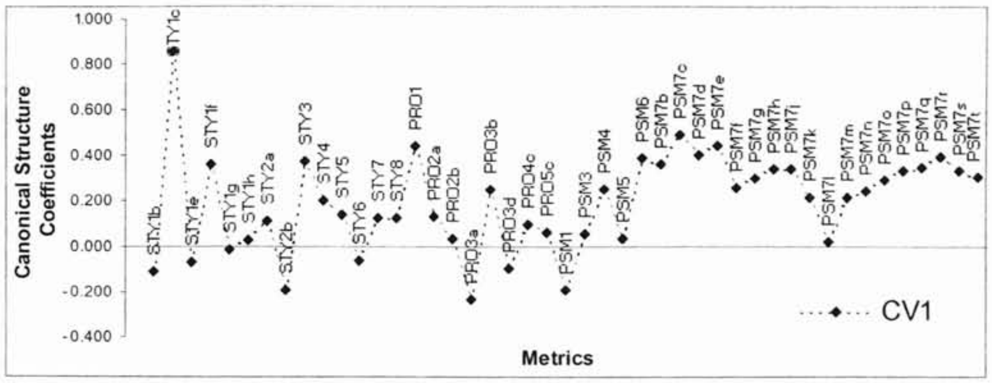
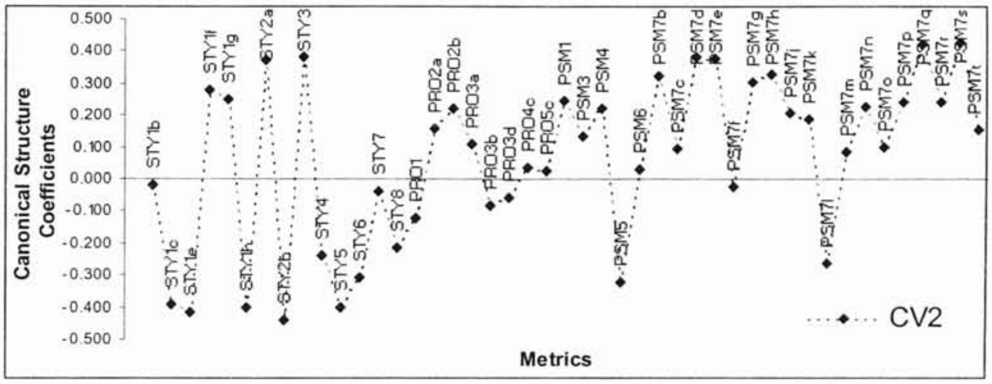


Figure 4.1 Contributions of individual canonical variates to the classification. A, B, C, and D refer to the data sets as described in Table 4.2. CVi represents the ith canonical variate. The numbers on y axis stand for the percentage changes of classification accuracy after adding a canonical variate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

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 PSM7l 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.





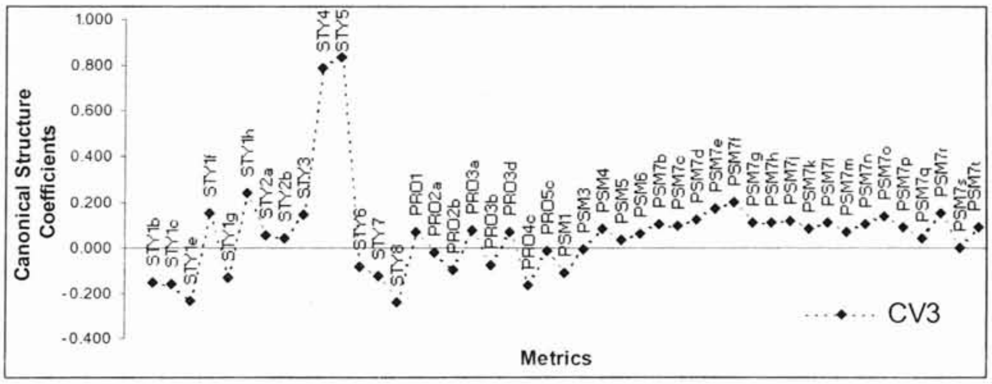
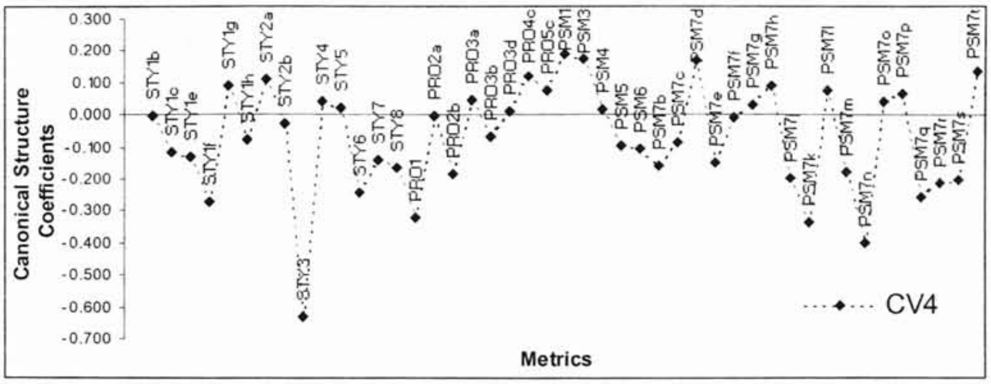


Figure 4.2 Canonical structures of the first five canonical variates (CV1, CV2, CV3, CV4, and CV5) for data set A. The canonical structure coefficients are correlation coefficients between the metrics and the canonical variates. The coefficients range from 1.0 to -1.0 and reveal how important each metric 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).



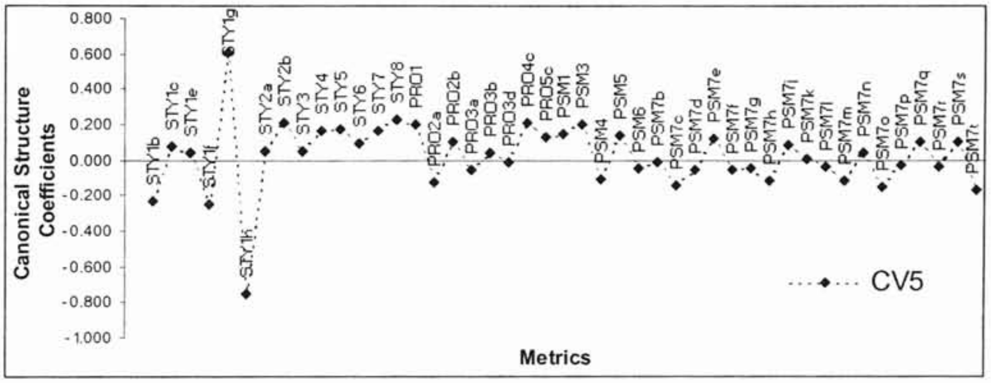
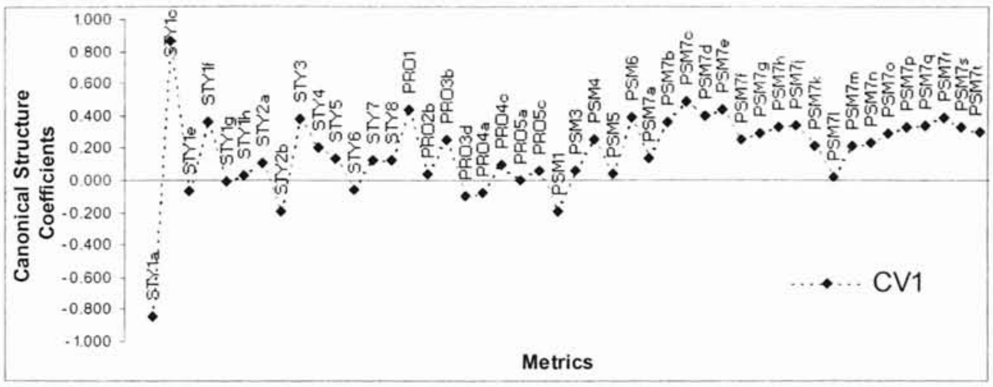
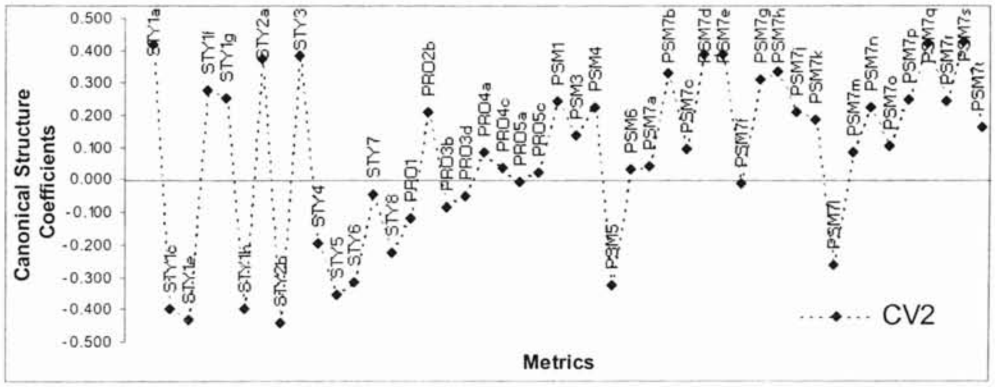


Figure 4.2 (continued) Canonical structures of the first five canonical variates (CV1, CV2, CV3, CV4, and CV5) for data set A. The canonical structure coefficients are correlation coefficients between the metrics and the canonical variates. The coefficients range from 1.0 to -1.0 and reveal how important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





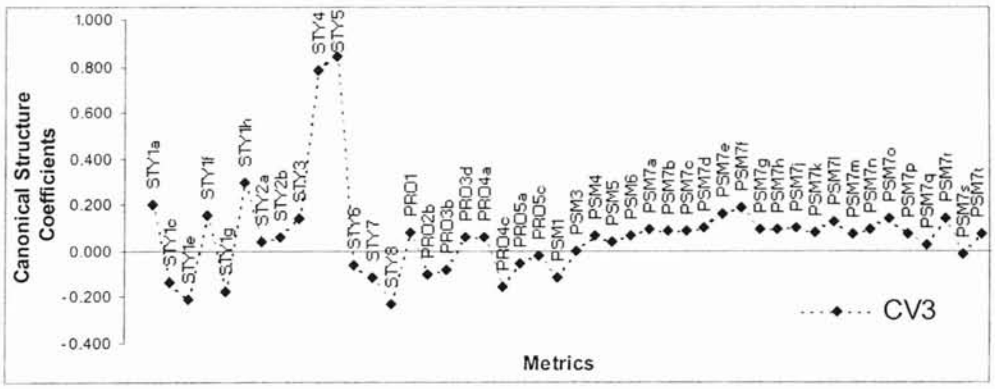
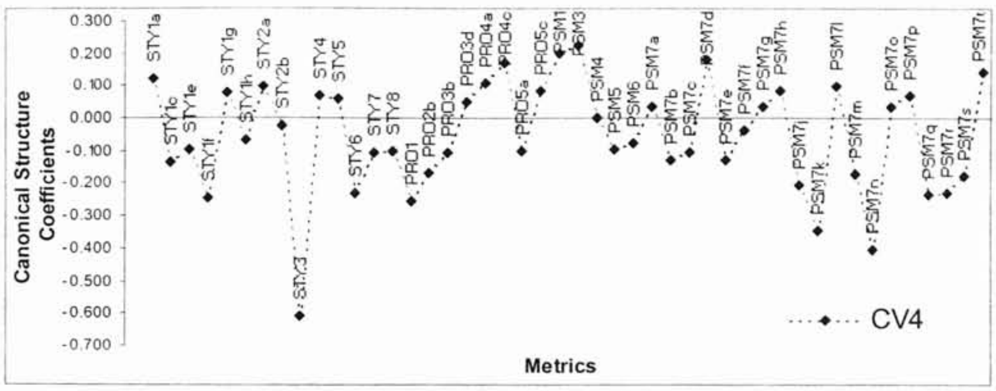


Figure 4.3 Canonical structures of the first five canonical variates (CV1, CV2, CV3, CV4, and CV5) for data set C. The canonical structure coefficients are correlation coefficients between the metrics and the canonical variates. The coefficients range from 1.0 to -1.0 and reveal how important each metric 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).



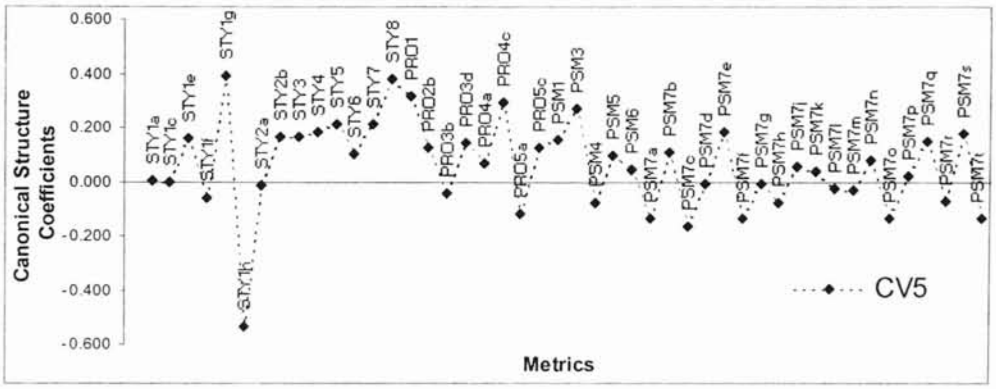


Figure 4.3 (continued) Canonical structures of the first five canonical variates (CV1, CV2, CV3, CV4, and CV5) for data set C. The canonical structure coefficients are correlation coefficients between the metrics and the canonical variates. The coefficients range from 1.0 to -1.0 and reveal how important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

CHAPTER V

SUMMARY AND FUTURE WORK

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

[Berghel and Sallach 1984] H. Berghel and D. Sallach, "Measurements of Program Similarity in Identical Task Environments", *ACM SIGPLAN Notices*, Vol. 19, No. 8, pp. 65-76, August 1984.

[Bieman and Debnath 1985] J. M. Bieman and N. C. Debnath, "An Analysis of Software Structure Using a Generalized Program Graph", *Proceedings of the 9th Annual IEEE International Computer Software and Applications Conference (COMPSAC'85)*, pp. 254-259, Chicago, IL, October 1985.

[Cha and Srihari 2000] S. H. Cha and S. N. Srihari, "Assessing the Authorship Confidence of Handwritten Items", *Proceedings of the Fifth IEEE Workshop on Applications of Computer Vision*, pp. 42-47, Buffalo, NY, December 2000.

[Conte et al. 1986] S. Conte, H. Dunsmore, and V. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings Publishing Company, San Francisco, CA, 1986.

[Donaldson et al. 1981] J. L. Donaldson, A. Lancaster, and P. H. Sposato, "A Plagiarism Detection System", *ACM SIGSCE Bulletin (Proceedings of the 12th SIGSCE Technical Symposium,* St. Louis, MO*)*, Vol. 13, No. 1, pp. 21-25, February 1981.

[Dunsmore 1984] H. E. Dunsmore, "Software Metrics: An Overview of an Evolving Methodology", *Information Processing & Management,* Vol. 20, No. 1, pp. 183-192, January 1984.

[Elliot and Valenza 1991] W. Elliot and R. Valenza, "Was the Earl of Oxford the True Shakespeare?", *Notes and Queries*, Vol. 38, No. 4, pp. 501-506, December 1991.

[Gray et al. 1997] A. Gray, P. Sallis, and S. MacDonell, "Software Forensics: Extending Authorship Analysis Techniques to Computer Programs", *Proceedings of the 3rd Biannual Conference of the International Association of Forensic Linguists (IAFL)*, pp. 1-8, Durham, NC, September 1997.

[Gray et al. 1998] A. Gray, P. Sallis, and S. MacDonell, "IDENTIFIED (Integrated Dictionary-Based Extraction of Non-Language-Dependent Token Information for Forensic Identification, Examination, and Discrimination): A Dictionary-Based System for Extracting Source Code Metrics for Software Forensics", *Proceedings of Software Engineering: Education & Practice Conference*, pp. 252-259, Dunedin, New Zealand, January 1998.

[Halstead 1977] M. H. Halstead, *Elements of Software Science,* Elsvier North-Holland, New York, NY, 1977.

[Holmes 1985] D. I. Holmes, "The Analysis of Literary Style - A Review", *Journal of the Royal Statistical Society: Series A*, Vol. 148, No. 4, pp. 328-341, April 1985.

[Jolliffe 1986] I. T. Jolliffe, *Principal Component Analysis*, Springer-Verlag Inc., New York, NY, 1986.

[Kenny 1986] A. Kenny, *A Stylometric Study of the New Testament*, Clarendon Press, Oxford, England, 1986.

[Kilgour et al. 1997] R. I. Kilgour, A. R. Gray, P. J. Sallis, and S. G. MacDonell, "A Fuzzy Logic Approach to Computer Software Source Code Authorship Analysis", *Proceedings of the 1997 International Conference on Neural Information and Intelligent Information Systems*, pp. 865-868, Dunedin, New Zealand, November 1997.

[Kjell 1994] B. Kjell, "Authorship Attribution of Text Samples Using Neural Network and Bayesian Classifiers", *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 2, pp. 1660-1664, San Antonio, TX, October 1994.

[Kjell and Frieder 1992] B. Kjell and O. Frieder, "Visualization of Literary Style", *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 1, pp. 656 – 661, Chicago, IL, October 1992.

[Kjetsaa 1979] G. Kjetsaa, "And Quiet Flows the Don Through the Computer", *Bulletin of the Association for Literary and Linguistic Computing*, Vol. 7, No. 3, pp. 248-256, September 1979.

[Krsul and Spafford 1996] I. Krsul and E. H. Spafford, "Authorship Analysis: Identifying the Author of a Program", *Technical Report*, CSD-TR-96-052, Department of Computer Science, Purdue University, West Lafayette, IN, September 1996.

[Leach 1995] R. J. Leach, "Using Metrics to Evaluate Student Programs", *ACM SIGCSE (The Association for Computing Machinery, Special Interest Group on Computer Science Education) Bulletin,* Vol. 27, No. 2, pp. 41-43 and 48, June 1995.

[Lewis and Loftus 1998] J. Lewis and W. Loftus, *Java Software Solutions: Foundation of Program Design,* Addison Wesley Longman, , Reading, MA, 1998.

[MacDonell et al. 1999] S. G. MacDonell, A. R. Gray, G. MacLennan, and P. J. Sallis, "Software Forensics for Discriminating between Program Authors Using Case-Based Reasoning, Feedforward Neural Networks, and Multiple Discriminant Analysis", *Proceedings of the 6th International Conference on Neural Information*, Vol. 1, pp. 66-71, Dunedin, New Zealand, November 1999.

[McCabe 1976] T. J. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 308-320, December 1976.

[Moreaux 1991] D. Moreaux, "A Formalism for the Detection and Prevention of Illicit Program Derivations", Master of Science Thesis, Department of Computer Science, University of Idaho, Moscow, ID, 1991.

[Mosteller and Wallace 1964] F. Mosteller and D. L. Wallace, *Inference and Disputed Authorship: The Federalist*, Addison Wesley, Reading, MA, 1964.

[Nejmeh 1988] B. A. Nejmeh, "NPATH: A Measure of Execution Path Complexity and Its Applications", *Communications of the ACM*, Vol. 31, No. 2, pp. 188-200, February 1988.

[Oman and Cook 1989] P. Oman and C. Cook, "Programming Style Authorship Analysis", *Proceedings of the Seventeenth Annual ACM Computer Science Conference*, pp. 244-247, Louisville, KY, February 1989.

[Oman and Cook 1991] P.W. Oman and C. R. Cook, "A Programming Style Taxonomy", *Journal of Systems and Software*, Vol. 15, No. 4, pp. 287-301, April 1991.

[Prechelt et al. 2000] L. Prechelt, G. Malpohl, and M. Philippsen, "Jplag: Finding Plagiarisms Among a Set of Programs", *Technical Report*, 2000-1, Fakultat fur Informatik, Universitat Karlsruhe, Karlsruhe, Germany, March 2000.

[Sallis 1994] P. Sallis, "Contemporary Computing Methods for the Authorship Characterisation Problem in Computational Linguistics", *New Zealand Journal of Computing*, Vol. 5, No. 1, pp. 85-95, January 1994.

[Sallis et al. 1996] P. Sallis, A. Aakjaer, and S. MacDonell, "Software Forensics: Old Methods for a New Science", *Proceedings of Software Engineering: Education and Practice Conference*, pp. 481-485, Dunedin, New Zealand, January 1996.

[SAS 1990] SAS, *SAS/STAT User's Guide*, Version 6, 4th ed., SAS Institute, Cary, NC, 1990.

[Si et al. 1997] A. Si, H. V. Leong, and R. W. H. Lau, "CHECK: A Document Plagiarism Detection System", *Proceedings of the ACM Symposium on Applied Computing*, pp. 70-77, San Jose, CA, February 1997.

[Spafford and Weeber 1993] E. H. Spafford and S. A. Weeber, "Software Forensics: Can We Track Code to Its Authors?", *Computers & Security*, Vol. 12, No. 6, pp.585-595, December 1993.

[Stratil and Oakley 1987] M. Stratil and R. J. Oakley, "A Disputed Authorship Study of Two Plays Attributed to Tirso de Molina", *Literary and Linguistic Computing*, Vol. 2, No. 3, pp. 153-160, September 1987.

[Whale 1990] G. Whale, "Identification of Program Similarity in Large Populations", *The Computer Journal*, Vol. 33, No. 2, pp. 140-146, March 1990.

APPENDICES

APPENDIX A

GLOSSARY

ANOVA	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].
Author	One who designs, writes, and tests pieces of computer code.
Authorship Identification	Determining the likelihood of a particular author having written some pieces of code, usually based on other code samples from the same programmer.
CDA	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].
CV	Canonical Variate, linear combination of the quantitative variables that summarizes the differences among the classes.
F Value	A statistical ratio of mean squares (variations) between groups to mean squares (variations) within groups.
IDENTIFIED	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].
LOC	Number of lines of code, a software metric.
MFC	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.
NCLOC	Number of non-comment lines of code, a software metric.
ODBC	Open DataBase Connectivity, a standard method of sharing data between databases and other programs.

Programming Style	A distinctive or characteristic manner present in computer programs written by a programmer.
SDA	Stepwise Discriminant Analysis, a statistical procedure to find a subset of quantitative variables that best reveals differences among the classes.
Software Forensics	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.
V(G)	McCabe's Cyclomatic Complexity Number [McCabe 1976].

APPENDIX B

PROGRAMS FOR CANONICAL DISCRIMINANT ANALYSIS (CDA) AND STEPWISE DISCRIMINANT ANALYSIS (SDA) WITH SAS (VERSION 8)

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*/ STY1b STY1d STY1e STY1g STY1h ;
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;
```

```
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;
```

SDA Processes

```
/*
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.

```
// 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;//toatl variables declared
       subroutineCount= 0;//number of subroutine count
//identifier length
       variableNameLength= 0;//total length in characters.
functionNameLength= 0;/*/toatl length in char. */
//chracter 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;//percentageof 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
```

```
4-->4: all
5-->5: all
*/
        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] == ' ') 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;

}

//receive a line and find the last open curly bracket
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
/*
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
4-->4: all
5-->5: all
*/
        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] == ' ') 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(&m_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
```

```
{
      CDocument::Dump(dc);
}
#endif //_DEBUG


//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  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,spaces are allowed after. not inlude alone a line*/
      /*spaces  are  allowed  before.  not  inlude  alone  a  line,  use
trimming space*/
      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;//toatl variables declared
       subroutineCount= 0;//number of subroutine count
//identifier length
       variableNameLength= 0;//total length in characters.
functionNameLength= 0;/*toatl length in char. */
//chracter 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;/*percentageof 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;
        extractMetrics(fileName);
        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 memeory
                  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(str);
//upper, lower, underscore case count
            caseHandle(cstr);
            keywordHandle(cstr);

//for curly styles
            if (cstr.Compare("{")==0) pureOpenCurly++;
            else if (cstr.Compare("}")==0) pureCloseCurly++;
            else {
                  if (cstr[0]=='{') beginWithOpenCurly++; //not include
pure curly lines
                  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++; slashComment++;}
                  else if (cstr[0]=='/' && cstr[1] == '*')
                        {pureComment++; slashStarComment++;
                        if (findStarSlash(cstr)) slashStarFlag = false;
//find after /*
                        else slashStarFlag = true;
                  }
                  else if (findSlashSlash(cstr))
                  {inlineComment++;slashComment++;}//find first
                  else if (findSlashStar(cstr)) {
                                    inlineComment++;slashStarComment++;
                        if (!findStarSlash(cstr)) slashStarFlag = true;
//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::findSlashSlash(CString& cstr)
{
        //if // is beturn /* 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::findSlashStar(CString& cstr)
{
        //if // is beturn /* 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 beturn /* 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]=='/' && cstr[j+1] == '/')
			{ 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] == '\t') tab++;
		else break;
		j++;
	}
	indentStructure ind;
	ind.indentSpace = space;
	ind.indentTab = tab;
	for (j= 0; j<k; j++)
		indentMem.push_back(ind);
}

//add indentaiton
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] == '\t') tab++;
		else break;
		j++;
	}
	indents.indentSpace += space-indentMem.back().indentSpace;
	indents.indentTab += tab-indentMem.back().indentTab;
```

```
        indents.indentLine++;
}

//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]=='i' && cstr[1]== 'f')
                {ifCount++; otherFlag=true;}
        else if (i>=2 && cstr[0]=='d' && cstr[1]== 'o')
                {doCount++;otherFlag=true;}
        else if (i>=3 && cstr[0]=='f' && cstr[1]== 'o' && cstr[2]=='r')
                {forCount++;forFlag = true;}
        else if (i>=4 && cstr[0]=='e' && cstr[1]== 'l' && cstr[2]=='s' &&
cstr[3]=='e')
                {elseCount++;otherFlag=true;}
        else if (i>=4 && cstr[0]=='c' && cstr[1]== 'a' && cstr[2]=='s' &&
cstr[3]=='e')
                {caseCount++;otherFlag=true;}
        else if (i>=5 && cstr[0]=='w' && cstr[1]== 'h' && cstr[2]=='i' &&
cstr[3]=='l' && cstr[4] == 'e')
                {whileCount++;otherFlag=true;}
        else if (i>=5 && cstr[0]=='s' && cstr[1]== 'w' && cstr[2]=='i' &&
cstr[3]=='t' && cstr[4] == 'c' && cstr[5] == 'h')
                {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-1  &&  cstr[j]  ==  '/'  &&  (cstr[j+1]  ==  '*'  ||
cstr[j+1] == '/')) break; //in case begin with
                if (cstr[j] == '+'||cstr[j] == '-'||cstr[j] == '*'||cstr[j]
== '/'||cstr[j] == '%' ||cstr[j] == '=') {
                        if (findOperants(cstr, j)) {
                          operatorCount++;
                          for (k = j-1; k>=0; k--) {
                                if (cstr[k]==' ') beforeSpace++;//before spaces
                                else break;
                          }
                        if (j<i-1 && cstr[j+1]=='=') j++; //jump over =,
                        for (k = j+1; k<i; k++) {
                                if (cstr[k]==' ') afterSpace++;
                                else break;
                        }//for
                }//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]=='=') right++; //for +=, *=, etc
        while (left>0 && cstr[left] == ' ')
                left--;
        if  (left>=0  &&  !isalpha(cstr[left])  &&  !isdigit(cstr[left])  &&
cstr[left] != '_' &&  cstr[left]  !=  ')'  &&  cstr[left]  !=  '$'  &&
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] != '"'&& 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] == '|'
                  || 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,   "throw")==0   ||   strcmp(token,
"throws")==0) throwCount++;
      // Get next token:
      token = strtok( NULL, seps );
   }
}

/* count variables and name length, nine common data types: short,
int,long, float, double, byte, char boolean, and String*/
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(), "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 (!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) )    ;
      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 && type)
                  openBracket = true;
            else  if   (strcmp(temp.c_str(),   "]")==0   &&   type   &&
openBracket)
                  openBracket = false;
            else if (strcmp(temp.c_str(), "[")==0 && variable)
                  openBracket = true;
            else if (strcmp(temp.c_str(), "]")==0 && openBracket)
                  openBracket = false;
            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)
                  equal = false;
            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;
```

```
			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 resturn 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)    ==    '$')    ||
(*(str+length) == '_')) {
			while      (isalpha(*(str+length))||isdigit(*(str+length))||
(*(str+length) == '$')|| (*(str+length) == '_')) {
				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
  DAddMetrics 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_catchCount = catchCount;
      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 2

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.