Recognizing Authors: An Examination of the Consistent Programmer Hypothesis

Jane Huffman Hayes
Computer Science Department,
Laboratory for Advanced Networking,
University of Kentucky
hayes@cs.uky.edu
http://www.cs.uky.edu/~hayes

Jeff Offutt
Software Engineering
George Mason University
Fairfax, Virginia 22030-4444
offutt@gmu.edu
http://www.cs.gmu.edu/~offutt/

Summary:
Software developers have individual styles of programming. This paper empirically examines the validity of the *consistent programmer hypothesis*: that a facet or set of facets exist that can be used to recognize the author of a given program based on programming style. The paper further postulates that the programming style means that different test strategies work better for some programmers (or programming styles) than for others. For example, all-edges adequate tests may detect faults for programs written by Programmer A better than for those written by Programmer B. This has several useful applications: to help detect plagiarism/copyright violation of source code, to help improve the practical application of software testing, and to help pursue specific rogue programmers of malicious code and source code viruses. This paper investigates this concept by experimentally examining whether particular facets of the program can be used to identify programmers and whether testing strategies can be reasonably associated with specific programmers.

Keywords:

author identification, source code metrics, plagiarism detection, software testing, static analysis, testability

1. Introduction

This paper experimentally examines the *consistent programmer hypothesis (ConPH):* that a facet or set of facets exist that can be used to recognize the author of a given program. If this hypothesis is valid, there are several useful applications. If some test strategies work better for some programmers or programming styles than for others, the practical application of software testing will be improved. Second, the ability to recognize authors will support the detection of plagiarism or copyright violation of source code. Third, being able to recognize authors may help pursue specific rogue programmers of malicious code and source code viruses and deter others considering such undertakings.

The dissemination of infectious computer viruses has become commonplace. Viruses and malicious code have cost companies, governments, and individuals billions of dollars in lost data and productivity. Apprehending and convicting perpetrators is inhibited by many factors, including inadequate technology law,

unclear international criminal jurisdiction, and the difficulty of positively identifying the code's author. When malicious code or viruses are in the form of source (such as macro languages used for word processing or spreadsheet packages), the proposed method can serve to help law enforcement authorities in the pursuit of a specific rogue programmer and serve as a deterrent to those considering such illegal activities.

Plagiarism detection is another application area for the consistent programmer hypothesis. It benefits the education sector by helping to detect student plagiarism. It can also benefit the legal sector in proof of ownership and software forensics. Plagiarism detection requires author identification (who really wrote the code), author discrimination (did the same person write both pieces of code), and similarity detection [1, 2]. This paper describes a method for recognizing the author of a given program, (or narrowing down to a set of potential authors) that can serve to help detect software plagiarism and presents results from an empirical evaluation.

1.1. *Competent Programmer Hypothesis*

The *competent programmer hypothesis* was identified by DeMillo et al., who made the observation that programmers have an advantage that is rarely exploited: "they create programs that are close to being correct!" [3]. The competent programmer hypothesis underlies several testing methods or techniques such as mutation testing [3].

A similar hypothesis is postulated in this paper, that programmers are **consistent**. This is referred to as the *consistent programmer hypothesis (ConPH).* The authors believe that this consistency means that certain test techniques (such as random testing and category-partition testing) will be better suited to some programmers than to others. Knowing that specific test techniques suit specific programmers could be very useful during practical unit (and possibly integration) testing of large multi-programmer applications, by allowing developers to apply only those test techniques best suited to individual programmers. The goal of this experiment is to evaluate the static and/or dynamic facets (if any) that can be used to recognize the author of a given program, as well as to see if certain testing techniques are better suited to some programmers than to others. A *facet* is defined as an aspect, dimension or characteristic of a program that can be quantified[1]. For example, program measures such as number of comments, total number of operands, and unique number of operators are all facets. These facets may belong to one or more domains (also called components) or software quality attributes such as understandability, maintainability, etc. The ConPH is discussed at length in the next section.

1.2. *Consistent Programmer Hypothesis*

This paper examines the conjecture that programmers are unique and that this uniqueness can be observed in the code they write. Programmers often have a penchant for certain constructs (perhaps preferring the **while** to the **for** loop) much as writers have preferences for certain words and for certain media (e.g., iambic pentameter [4]). This uniqueness gives each program a "signature" of the person who wrote it. Educators

[1] The term facet is also employed in Maturity Modeling research (such as the Software Engineering Institute's Capability Maturity Model Integration [48]) to refer to measured aspects of an organization or undertaking that is being analyzed.

teaching young children to write refer to this property as "voice." As will be mentioned in Section 2.1, authorship attribution researchers refer to this as "style." The goal of this experiment is to learn the specific characteristics of a program that amplify the programmer's "voice."

To examine the consistent programmer hypothesis further, the ability to identify the author of a piece of text is studied. The motivation for this research started with an informal survey of a dozen colleagues in industry and academia. The survey was conducted via email over the course of several weeks. Most of those surveyed claimed they could identify the author of a hand-written note if the colleague had worked with the author for a number of years. Further, many commented that they could identify the author of a piece of text even if it had been typed, if they had worked with the author for several years. This was true regardless of whether the text section appeared in an email, specification, research paper, or report. When questioned further, they stated that certain phrases, certain words, bolding or capitalizing of certain words, or use of certain non-textual symbols (such as dashes) tipped them off to the author. Similarly, some of those interviewed agreed that they could identify the author of a piece of code, if they had worked with the author for some time. This was due in part to syntactic formatting of the code (indentation, use of headers, capitalization of variables), but also due to the use of certain constructs and patterns of constructs. Identification of the author of source code is of interest here. This informal, preliminary survey led us to formulate the principal hypothesis of this research.

The consistent programmer hypothesis postulates that a facet, or a set of related facets, exists that can identify the author of a program. That is to say, there exists a set of facets *f* that are elements of the program *p* that can be used to map the program *p* to one or more potential authors *a*. Further, it is postulated that a test technique *t* exists that is better suited to programs written by author a_1 than to programs written by author a_2. Ramsey and Basili also felt that the structure of code impacted testing: "the data suggests that it may be possible to differentiate test sets using structural coverage [5]." In addition to the structure of code, the authors hypothesize that programmers tend to make the same types of mental errors and hence introduce the same types of faults into the code that they write. This hypothesis was generated based on observation, a literature survey, and brainstorming. There is evidence to support the idea that some quality assurance techniques are better at finding some types of faults than others (testing techniques included) [6]. The authors decided to examine widely used testing techniques (that also may be applied in an automated fashion by advanced testing organizations) to determine if testing techniques are better suited to certain programmers. This question was studied by undertaking a series of experiments, first gathering data from professional programmers, and then from students.

1.3. *Experiment*

A series of experiments were undertaken to study the consistent programmer hypothesis. In the first experiment, five professional programmers wrote the same three applications that were then evaluated statically and dynamically. The static evaluation showed that five aspects of programs, or *facets*, have potential for recognizing the author of a program or at least narrowing down to a set of potential authors: *(1) number of lint warnings, (2) number of unique constructs, (3) number of unique operands, (4) average*

occurrence of operators, and *(5) average occurrence of constructs*[2]. Also, three of these facets were shown to be correlated with the individual programmer and the application. Dynamic evaluation was used to obtain a testability measure for each program, but the measure was not found to be correlated with the individual programmer and application (Section 3 discusses this further). In a second study, the work was repeated for four programs written by 15 graduate students, and it was found that ratio of the *number of semi-colons per comment* also has potential for recognizing authors.

1.4. *Paper Organization*

Section 2 discusses related work and the consistent programmer hypothesis. Section 3 describes the research hypothesis. Section 4 defines the experiment design. It presents a description of the techniques used, the subject programs evaluated, and the measurements taken for these subject programs. Section 5 addresses the analysis and results of the experiment. Finally, Section 6 presents conclusions and future work.

2. Background

The concept investigated for this paper is related to literary analysis and software forensics, so a brief review is required. This review is followed by a statement of the main topic for this research, the consistent programmer hypothesis.

2.1. *Literary Analysis and Software Forensics for Authorship Attribution*

Literary analysis for authorship attribution (also called *stylometry*) refers to determining the author of a particular piece of text. *Style*, also called *voice*, concerns the way in which a document is written rather than its contents [7]. Stylometry has been a field of interest for many centuries, for those in the humanities as well as lawyers, politicians, and others [8]. Many techniques exist for analyzing literature/text in order to determine the author. A three-step process is often applied: canonicization, determination of event set, and statistical inference. *Canonicization* refers to restricting the event space by treating similar events as identical (in English, the lower and upper case E would be treated as identical, for example) [8]. The *event space* is the collection of all possible event sets. Event set determination involves partitioning the literature (the text) into non-overlapping events and removal of non-meaningful events. The analyst can specify what should be extracted as *event sets*. For example, characters can be extracted as event sets, words can be event sets, character N-grams can be event sets (a collection of N characters), and word N-grams can be event sets (a collection of N words) [43]. The events that have not been discarded are then subjected to statistical analysis such as Principal Components Analysis [46] or Linear Discriminant Analysis [47].

[2] A construct is a reserved word in the programming language such as if, else, for. Constructs are a subset of Halstead's operators.

Let us step through a typical three-step literary analysis process, using the steps described above. One can imagine many ways to proceed with such analysis when attempting to discover the author of a piece of text written in English (a user's manual, for example). One may decide that individual letters will be events (such as a, e, p, etc.) or that words will be events. If using words as events, a removal list may be used to winnow the list of events. Once the event list has been obtained, statistical inference might be performed by building a histogram of the events and comparing this to a standard such as the Brown histogram [9]. The Burrows method follows the same three steps outlined above, with specific activities applied in steps two (event set determination) and three (statistical inference) [10, 11]. In the event set determination step, most words are thrown out, except for a few select *function words* (a part of speech that marks grammatical structure, such as an article, preposition, conjunction). These few words are used to build a histogram that is then compared to similar histograms collected from known documents [10, 11]. Juola and Baayen follow the same process, use either letters or words as the event set, and then look at the *cross-entropy* between the histogram (or probability distribution) of the current text and of other historical or standard texts [12]. Cross-entropy is the application of the Kullback-Leibler divergence [44] to small samples. Specifically, "the Kullback–Leibler divergence is a non-commutative measure of the difference between two probability distributions P and Q. KL measures the expected difference in the number of bits required to code samples from P when using a code based on P, and when using a code based on Q. Typically P represents the "true" distribution of data, observations, or a precise calculated theoretical distribution. The measure Q typically represents a theory, model, description, or approximation of P [45]."

Software forensics refers to the use of measurements from software source code or object code for some legal or official purpose [13]. Plagiarism detection requires **author identification** (who really wrote the code), **author discrimination** (did the same person write both pieces of code), and **similarity detection** [13]. Some features and metrics suggested for malicious code analysis by Gray et al. [1] and Kilgour et al. [2] include: choice of programming language, formatting of code (e.g., most commonly used indentation style), commenting style (e.g., ratio of comment lines to non-comment lines of code), spelling and grammar, and data structure and algorithms (e.g., whether pointers are used or not). Some metrics suggested for plagiarism detection by Sallis et al. [14] are volume (e.g., Halstead's n, N, and V [15]), control flow (e.g., McCabe's V(G) [16]), nesting depth, and data dependency.

A web application called JPlag has been used successfully to detect plagiarism in student Java programs. It finds pairs of similar programs in a given set of programs by parsing the programs, converting the program to token strings, and then applying the Greedy String Tiling algorithm [17]. Schleimer et al. [18] use *winnowing*, a local fingerprinting algorithm, to select document k-grams as the document's representative fingerprints (the k-gram model assumes that the probability of a "word" occurring in a sentence depends only on the previous k-1 words). The Moss tool, which Schleimer et al. used in experiments, is based on this technology [18]. Oman and Cook examined authorship analysis by focusing on markers such as blocks of comments, character case (upper versus lower), etc. They used clustering analysis and found that authors who consistently used such markers could be identified. A problem with this study is that they examined textbook implementations of algorithms, and these could have been improved or modified by editors and might not illustrate the original

author's style [19]. Spafford and Weeber [20] define software forensics as examining code remnants[3] to gather evidence about the author. They compare this to handwriting analysis. They suggest a number of features that may provide evidence of the author such as data structures and algorithms, choice of system calls, errors, comment styles, etc. Krsul examined programming layout metrics, such as placement of comments, placement of brackets, indentation, etc., in order to identify program authors [21].

Researchers at the University of Otago have developed a system, Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination (IDENTIFIED), to extract counts of metrics and user defined meta-metrics to support authorship analysis [1]. In a later paper [22], they examined the usefulness of feed-forward neural networks (FFNN), multiple discriminant analysis (MDA), and case-based reasoning (CBR) for authorship identification. The dataset included C++ programs from seven authors (source code for three authors came from programming books, source code for another author was from examples provided with a C++ compiler, and three authors were experienced commercial programmers). Twenty-six measures were extracted (using IDENTIFIED), including proportion of blank lines, proportion of operators with whitespace on both sides, proportion of operators with whitespace on the left side, proportion of operators with whitespace on the right side, and the number of **while** statements per non-comment lines of code. All three techniques provided authorship identification accuracy between 81.1% and 88% on a holdout testing set[4], with CBR outperforming the other two techniques (FFNN and MDA) in all cases by 5 – 7% [22].

Software engineering researchers have indirectly examined authorship when studying other areas. Programmer characteristics have been indirectly addressed by research examining the maintainability of modules written by different programmers [23, 24] and by research on the fault proneness of modules written by different programmers [25].

Kilgour et al. [2] looked at the usefulness of fuzzy logic variables for authorship identification. The dataset was comprised of eight C++ programs written by two textbook authors. Two experienced software developers then subjectively analyzed the programs, examined measures such as spelling errors, whether the comments matched the code, and meaningful identifiers. They then assigned one of the fuzzy values **Never**, **Occasionally**, **Sometimes**, **Most of the Time**, and **Always** to each measure. The authors concluded that fuzzy-logic linguistic variables have promise for improving the accuracy and ease of authorship analysis models [2].

Collberg and Thomborson [26] examined methods for defending against various security attacks. They suggested using code obfuscation to transform a program into another that is more difficult to reverse engineer, while maintaining the semantics of the program. It appeared that control and data transformations might hold promise for erasing a programmer's "style," though not all factors being explored for the consistent programmer hypothesis (ConPH) would be ""erased." Also, the lexical transformation they presented [26] would not serve to remove the programmer's signature.

---

[3] A remnant of an attack may take many forms, including programming language source files, object files, executable code, shell scripts, changes made to existing programs, or even a text file written by the attacker [20]. This paper concentrates on source files or fragments of source files.

[4] In holdout validation, "observations are chosen randomly from the initial sample to form the validation data, and the remaining observations are retained as the training data. Normally, less than a third of the initial sample is used for validation data." From Wikipedia entry on Cross Validation, http://en.wikipedia.org/wiki/Cross_validation .

The work presented here places the emphasis on structural measures instead of stylistic measures. When attempting to mask identity, an author can easily modify stylistic items within a program. Stylistic measures such as blank lines, indentation, in-line comments, use of upper or lower case for variable names, etc. are then no longer reliable. This approach to masking is seen often when multiple students have copied from one student's program. The plagiaristic students re-name all the variables. They remove or modify the comments and blank space. They re-order the methods or procedures. They modify indentation. Also, stylistic features may be omitted from source code whereas programmers must use structural constructs to write functioning code. The approach proposed here does not require large quantities of data (such as is needed to train a FFNN). A unique aspect is that a number of testing-related measures are used to augment the proposed authorship recognition approach. In addition, measures derived from dynamic analysis of the programs as well as measures derived from static analysis are used in the proposed approach. An experiment was performed to validate the research. Professional programmers who developed the same programs according to written program specifications were used in the experiment. This helped control for confounding factors.

2.2. *A Potential Application of the Consistent Programmer Hypothesis*

The consistent programmer hypothesis was introduced in Section 1.2. Here, a high level illustration of how the ConPH might be used is presented. Specific details are deferred until later in the paper. If the facets discussed above can be used to map a program to a unique author or at least narrow down to a set of potential authors, they may also be used to generate an identifier or discriminator for a programmer, called the *target discriminator*. A discriminator could similarly be built for individual programs, called the *source discriminator*. The idea is to collect enough information (experimentally) to be able to build a look-up table of target discriminators (unique to a programmer) and suitable quality assurance methods. To illustrate, testing techniques are examined first. Suppose that by experimentation and actual execution of various test techniques, it is discovered that the Category-Partition test technique is best suited to programmer 4's style (where programmer 4 has a target discriminator value of 374), that programmer 5 has a target discriminator value of 436, and that it appears that both random testing and category-partition testing work equally well for programmer 5. The corresponding look-up table would be:

	Target Discriminator Value	Category-Partition (test technique 1)	Random (test technique 2	….	….	Another Test Technique (test technique N)
Programmer 4	374	X				
Programmer 5	436	X	X	….	….	
…….						
…….				….	….	
Programmer M						

Note that a large number of test techniques and programmers could be listed in the table (hence the Mth programmer and Nth test technique).

When a given program is being evaluated to decide the type of testing technique to use: the source discriminator (discriminator for a program) will be calculated, a simple table look-up procedure will be used

to find the target discriminator, and the row for the target discriminator will identify the applicable test techniques.

It is also desirable if the facet or set of facets for identifying an author can be derived from as small a sample of code as possible. In many cases, only one module or source code method will be available as the "fingerprint" for an author that one is trying to "tie" to another piece of code (such as a piece of malicious code, another student's program, etc.).

3. Research Hypothesis

The general hypothesis for this experiment is that one or more facets exist for a program that can identify the author of the program. Informal hypotheses are presented below followed by more formal null hypotheses.

1. The dynamic measure of *testability* (the likelihood that software faults in a program will be detected during testing [27]) is correlated with the input distribution used and the individual programmer such that one particular testing technique (category-partition, random, or all-edges) is best suited to a particular programmer's code. The null hypothesis is that the mean value of testability is not affected by the programmer.
2. The static measure of *number of semi-colons per comment* is correlated with the individual programmer. The null hypothesis is that the mean value of number of semi-colons per comment is not affected by the programmer.
3. The static measure of *average occurrence of operands* is correlated with the individual programmer. The null hypothesis is that the mean value of average occurrence of operands is not affected by the programmer.
4. The static measure of *average occurrence of operators* is correlated with the individual programmer. The null hypothesis is that the mean value of average occurrence of operators is not affected by the programmer.
5. The static measure of *average occurrence of constructs* is correlated with the individual programmer. The null hypothesis is that the mean value of average occurrence of constructs is not affected by the programmer.
6. The static measure of *average occurrence of constants* is correlated with the individual programmer. The null hypothesis is that the mean value of average occurrence of constants is not affected by the programmer.[5]
7. The static measure of *number of unique constructs* is correlated with the individual programmer. The null hypothesis is that the mean value of number of unique constructs is not affected by the programmer.
8. The static measure of *number of unique operands* [15] is correlated with the individual programmer. The null hypothesis is that the mean value of number of unique operands is not affected by the programmer.
9. The static measure of *number of lint warnings* is correlated with the individual programmer. The null hypothesis is that the mean value of number of lint warnings is not affected by the programmer.

The null hypothesis was rejected when the probability that the differences in the facets were attributable to chance was small (p-value was 0.05 or less). That is to say, if the null hypothesis was rejected for facet X, facet X helps to identify uniquely the author of a program. The alternative hypothesis is that the mean value of the facet was affected by the programmer. Note that this process was used as a starting point for finding potential facets to help build a discriminator.

[5] A constant will be defined in a program (such as static final int MAX_LENGTH = 2000;), each subsequent use of the constant (MAX_LENGTH) is counted as one (1) occurrence of a constant.

4. Experiment Design

Two studies were undertaken, one using professional programmers and the other using students. In the first study, five programmers wrote the same three C applications. This study consisted of six steps. First, all programs were compiled and tested in an ad hoc fashion to ensure that the competent programmer assumption could be made (that the programs are close to correct). Second, control flow graphs were manually generated for each program. Third, test case sets were generated for all 15 programs using three separate testing techniques. Fourth, measures such as number of unique operands, and number of constants per semi-colon were collected manually to perform static evaluation of the applications. Fifth, the programs were submitted to *lint* (a static analyzer that detects poor programming practices such as variables that are defined but not used). Sixth, the test cases were submitted to the Automatic Test Analysis for C (ATAC) tool [28] to ensure that minimal coverage criteria were met. The PISCES^{TM} test analysis tool [27] was used to perform dynamic evaluation, measuring testability of the programs (dynamic evaluation is described in Section 4.3.1).

The second study attempted to see if similar results would hold true for students who may not have had time to develop a distinct style. Fifteen graduate students wrote the same four C programs. This study consisted of the first four steps performed for the first study (see above).

The remainder of section 4 is organized as follows. Section 4.1 discusses the testing techniques used in both studies. The programmers and programs of the first study are presented in section 4.2. The measurements for both studies are presented in section 4.3. Unique aspects of the second study are discussed in Section 4.4.

4.1. *Testing Techniques*

Testing is used to experimentally evaluate the consistent programmer hypothesis because a major purpose of the ConPH is to improve testing in a practical way. It is postulated that some programmers will achieve higher dynamic evaluation scores (testability) when using certain test techniques. Three common test techniques were chosen: random testing [29], category-partition testing [30, 31], and all-edges testing [32, 33]. It is worth noting that because programmers were given a great degree of flexibility in implementing solutions, not all test cases could be used for all versions of the programs. Specifically, one programmer implemented Mid in such a way that all three inputs (integer values) had to be unique. That is, if a "1" was entered for the first value, a "1" could not be entered for the second or third values. This made it impossible to run a few of the random and category-partition test cases for that program, so slight modifications had to be made to the test case set. The next three subsections describe how these test techniques were used in the experiment.

4.1.1. *Random testing*

Beizer defines *random testing* as the use of a random data generator that will force arbitrary but achievable paths through a routine [29]. For this work, random testing refers to the use of a random number generator to build test cases that meet a specified coverage criterion. The coverage aspect was added to ensure

measurability and comparability of the results of all the testing techniques. Random test cases were built for all three programs using a three-step procedure. First, values were generated using a random number generator based on the C library utility **rand**. To run the random number generator, the number and range of values were specified. The range of values was from -9999 to +9999. The second step was to submit the subject programs to the ATAC tool [28] to determine block, decision, computation-use, and predicate-use coverage for the test cases. Block coverage refers to the percent of basic blocks that are executed.

Checking the coverage of the random test cases required three ATAC runs, one run for each of the three programs[6]. The third step required the generation of additional test cases and an ATAC re-run for programs that did not achieve coverage of at least 90% for one of the 4 measures. This third step was repeated until at least one of the coverage measures equaled or exceeded 90%. Additional random test cases were added for both Trityp and Mid to reach 90% coverage.

4.1.2. *Category-partition testing*

The *category-partition testing* technique is a specification-based technique that uses partitioning to generate functional tests for programs [30, 31]. The specifications shown in Figure 1 were used to generate category-partition test cases. The steps followed were: (1) analyze the specification, (2) partition the categories into choices, (3) determine constraints among the choices, and (4) write test cases [30].

Once test cases were generated, they were submitted to ATAC, just as described for random test cases. This required three ATAC runs, one for each of the three programs[6]. For one program (Trityp), additional test cases had to be added to reach 90% coverage. The coverage aspect was added to ensure measurability and comparability of the results of all the testing techniques.

Program 1 ---> Write a subroutine (or function) FIND that has three parameters: An array of elements (A); an index into the array (F); and the number of elements that should be considered (N). F is an index into array A. After FIND is executed, all elements to the left of A(F) are less than or equal to A(F) and all elements to the right of A(F) are greater than or equal to A(F). Only the first N elements of the array are considered.

Program 2 ---> Write a subroutine or function MID that examines three integers and returns the middle value (for example, if 6, 9, and 3 are entered, 6 is returned).

Program 3 ---> Write a subroutine or function TRITYP that examines 3 integer values and determines what type of triangle (if any) is represented. Output will be:
TRIANG=1 if triangle is scalene
TRIANG=2 if triangle is isosceles
TRIANG=3 if triangle is equilateral
TRIANG=4 if not a triangle

Figure 1. Specifications for the Find, Mid, and Trityp.

[6] Only one version of each of the three programs was needed in order to generate test cases (random and category-partition). The cases were then run on all five programmer's programs. It was randomly decided to use programmer 3's version of each program (mid, find, trityp) to build the first set of test cases (that were then improved until the ATAC tool showed that the coverage goal had been met).

4.1.3. *All-edges testing*

The *all-edges criterion* (also called branch testing) requires that every edge in the program's flow graph be executed by at least one test case [32, 33]. Since branch testing is a white box testing technique it depends on actual details of the program implementation. Therefore, a flow graph had to be drawn for all 15 programs in order to build test cases. For this experiment, control flow graphs were manually generated, and the all-edges test cases were built by hand. The test cases were then submitted to ATAC for each program. This required fifteen ATAC runs; five runs for each programmer's version of Find, five runs for each version of Mid, and five runs for each version of Trityp.

It should be noted that 100% coverage could not be achieved for all versions of all programs. For example, one programmer's implementation of Trityp included some dead code, which made it impossible to achieve 100% for blocks covered. Similarly, faults in several of the programs prevented certain branches from being taken. For practical purposes, a set of test cases was considered adequate if at least one coverage measure reached or exceeded 90%. In most cases, the coverage reached 97% or more, but in one or two cases the highest coverage value was 91% (see table below for block coverage for all-edges). For these lower coverage percentage programs, additional test cases were not added. This is because the same method of generating flow graphs and building test cases was carefully applied to each program. Arbitrary generation of additional test cases for an individual program would require that all program test cases be revised; otherwise a bias would be introduced.

Programmer/Program	*Trityp*	*Find*	*Mid*
Programmer 1	91%	100%	93%
Programmer 2	97%	100%	100%
Programmer 3	91%	100%	100%
Programmer 4	97%	95%	100%
Programmer 5	100%	97%	100%

4.2. *Programmers*

This research seeks to exploit differences among individuals. All programmers were given the same specifications for the three programs, and were asked to implement the programs in C by a specified date. The subject programs were intentionally short: to serve as a small sample of code; to increase turn-around time; and to minimize volunteer dropout. Programmers were not monitored. No time limit was set. No order of development of the three programs was mandated. No requirements for in-line documentation (comments) were identified. These types of requirements and constraints were not levied in order to encourage the programmers to use their own characteristic style.

4.2.1. *Profile of the programmers*

An attempt was made to use experienced programmers in the experiment. One possibility is that novice programmers or programmers new to a particular programming language might not exhibit a unique style or

voice, because they are still developing their style. Researchers have found this to be true in the field of literary authorship attribution [34]. That is, novice writers may not exhibit a unique style or "authorial fingerprint." The second experiment used inexperienced programmers to separate this variable.

Five programmers with varying backgrounds and levels of experience assisted with the experiment. A profile of the programmers is shown in Table 1.

Table 1. Programmer Profiles.

Subject	Employer	Highest Degree	Work Experience
1	BDM Int'l Inc.	M.S. SWE	Programmer, Analyst
2	Science Applications International Corp. (SAIC)	M.S. CS	Programmer, Analyst
3	SAIC	M.S. CS	Analyst, Programmer
4	E-Systems	M.S. CS	Programmer, Analyst
5	George Mason University	Ph.D. CS	Asst. Professor, Programmer

The disparity in the size of the solutions and the amount of time programmers spent on the solutions was striking. For example, one programmer implemented the program Find in 23 lines (counting semi-colons), while another programmer used 69 lines. Similarly, the amount of in-line documentation varied greatly. Several programmers supplied no comments at all, while another provided a comment for every 1.5 semi-colons.

4.2.2. *Programs*

The three programs were Find, Mid, and Trityp. The specifications for these programs were derived from the in-line comments in implementations by Offutt [35]. Figure 1 presents the specifications exactly as they were provided to the programmers. It should be noted that the restrictions in program Find relating to the index (F) being less than or equal to the number of relevant elements (N), were ignored during test case generation. That is, test cases were generated to violate these constraints (F > N) as well as to comply with the constraints (F <= N).

Questions were received from three of the programmers on the Find program, so a subsequent specification clarification was sent to and was used by all five programmers (they had not finished their solutions before they received the information). It is shown in Figure 2. It should be noted that the researcher's response to the questions might have biased the programmers in favor of a sorting solution to Find. All programs met the specifications.

It does not matter what type of array you use or how you get input (from a file, interactively, etc.). You don't have to sort the array, though that is a possible solution. If N=6, elements a[7] through a[10] are ignored. If you have the following values, a=2 5 1 1 3 4 4 4 4 4, N=6, F=3, one correct result would be: a=1 1 2 5 3 4 because a[F]=1 and all values .LE. 1 are now to the left of 1 and all values greater than or equal to 1 are now to the right of it (even though they are not sorted). a=1 1 2 3 4 5 is also a correct result.

Figure 2. Clarification of Program Find.

Table 2 shows the size of the programs used in the first experiment. Five programs were used for each of the three specifications. The size is given in lines of executable code (LOC).

Table 2. Size of Programs from First Experiment.

Program	LOC 1	LOC 2	LOC 3	LOC 4	LOC 5
Find	46	28	23	69	46
Mid	35	10	16	18	17
TriTyp	34	13	20	36	26

4.3. *Measurements*

The C programs were analyzed statically and dynamically, as discussed below.

4.3.1. *Experimental evaluation using PISCES*

PISCES performs dynamic failure analysis to predict the testability of programs. *Testability* [27] is defined as the likelihood that software faults that exist in the program will be revealed during testing. If testability is high (close to 1), then it is likely that any faults in the code will easily be found during testing. If testability is low (close to 0), it is not likely that faults will be found [27].

PISCES performs execution, infection, and propagation for each program it analyzes. *Execution* analysis "estimates the probability of executing a particular location when inputs are selected according to a particular input distribution" [27]. *Infection* analysis is based on mutation analysis. Mutation analysis is a fault-based testing technique that helps the tester create a set of test cases to detect specific, predetermined types of faults [3]. Mutation analysis systems introduce a large number of simple faults, called mutations, into a test program to create a set of mutant programs. These mutants are created from the original program by applying mutation operators, which describe syntactic changes (such as relational operator replacement or statement deletion). Test cases are then measured by determining how many of the mutant programs produce incorrect output when executed. Each live mutant is executed with the test cases and when a mutant produces incorrect output on a test case, that mutant is said to be "killed" by that test case and is not executed against subsequent test cases [35]. Infection analysis generates program mutants and then estimates the probability that a program mutant alters a data state (variables, input files, output files, and the program counter) of the program [27].

Propagation analysis alters (or infects) a data state of a program (termed perturbing) and then examines the probability that this perturbation will result in a change in the program output (i.e., the perturbation propagated) [27]. An input distribution (that is, a set of test cases) must be provided for each program run with this tool. The tests are used to estimate the probabilities for execution, infection, and propagation.

Three testing techniques (category-partition, random, and all-edges) described in Section 4.1 were applied to all 15 programs. This required a total of 45 PISCES runs: one run for each of the 15 programs for all-edges tests (15 runs); one run for each of the 15 programs for category-partition tests (15 runs); and one run for each of the 15 programs for random tests (15 runs).

4.3.2. *Experimental evaluation using static analysis*

All programs were manually analyzed so that static measures could be gathered. Fourteen direct measures were extracted from the source code and test case generation. Five indirect measures, such as average number of comments per semi-colon, were also obtained.

Counting was performed consistently for each program (using the CMT++ tool by Testwell[7]). Most static measures are self-explanatory (such as number of semi-colons). The counting rules that are not so obvious are:

Operator: One of the following: ! != % %= & && || &= ( ) * *= + ++ += , - -- -= -> . ... / /= : :: < << <<= <= = == > >= >> >>= ? [ ] ^ ^= { } | |= ~

Operands: Operator and operand counts are very intuitive. The statement f(x, y) is counted as follows: f, x and y are operands and the parentheses and comma are operators.

Number of comments: From /* to the ending symbol */ was counted as 1 comment, even if it ran over several lines.

Average occurrence of operands: Defined as the total number of operands in a program divided by the number of unique operands. Suppose a program had only 2 unique operands, A and B. A occurred three times in the program and B occurred four times. The average occurrence is found by adding the total number of occurrences of all operands (3 + 4 = 7 occurrences) and dividing by the number of unique operands (7/2 = 3.5). Thus 3.5 would be the value for this program.

Average occurrence of operators: Defined as the total number of operators in a program divided by the number of unique operators.

[7] http://www.testwell.fi/cmtdesc.html

Average occurrence of constants: Defined as the total number of constants in a program divided by the number of unique constants. This value is set to 0 if the number of unique constants is 0.

Average occurrence of constructs: Defined as the total number of constructs in a program divided by the number of unique constructs. It should be noted that an **if** construct was counted as 1 even if it contained multiple **else if** clauses or an **else** clause.

In addition to these static measures, the UNIX utility *lint* was run on each program to gather information on the number of warnings received. The manual generation of the program flow graph, the extraction of each of the static measures from the code, and the running of *lint* took approximately 1 hour and 35 minutes per program (15 programs total). Note that the most time consuming process, manual extraction of measures, could be performed quite quickly if a tool was available (no tool was available to the authors until after the work had been performed manually).

4.3.3. *Data Analysis Techniques*

The static and dynamic measures for each program were recorded as the rows in a spreadsheet as shown in Table 3. Each measure in the table is mapped to a research hypothesis (such as Research hypothesis 1 (R1) -- testability) or is used to calculate a measure for a hypothesis (Research hypothesis 2 (R2) requires number of comments in order to calculate number of semi-colons per comment), as indicated in the first column of the table. Next, analysis of variance (ANOVA) was applied. The assumption of normality of errors was met (as could be seen visually by the residuals scatterplot – the residuals were normally distributed). Homogeneity of variance was met for the majority of the measures (as shown by Bartlett's chi-square)[8]. The design of the ANOVA was one between subject factor, with that factor being programmer. The factor had five levels (one for each programmer) and there were three observations in a condition (one for each program). This required thirteen runs, one for each measure shown in Table 3, each run took approximately one minute.

[8] ANOVA is a robust procedure and works well unless both the assumption of normality and of homogeneity of variance are violated. That was not the case for the data presented here.

Table 3. Static and Dynamic Measures Collected for Consistent Programmer Hypothesis Experiment.

Research		Find	Mid	Trityp	Find	Mid	Trityp	Find	Mid	Trityp	Find	Mid	Trityp	Find	Mid	Trityp
Hypothesis		Pgmr 1	Pgmr 1	Pgmr 1	Pgmr 2	Pgmr 2	Pgmr 2	Pgmr 3	Pgmr 3	Pgmr 3	Pgmr 4	Pgmr 4	Pgmr 4	Pgmr 5	Pgmr 5	Pgmr 5
	Variables:															
For R2	# comments	16	1	0	0	0	0	3	2	1	21	4	8	22	10	17
For most	# semi-colons	46	35	34	28	10	13	23	16	20	69	18	36	46	17	26
R8	# unique operands	17	5	4	16	9	9	9	4	4	26	12	15	13	9	7
R7	# unique constructs	11	9	9	7	5	5	4	3	3	6	4	6	6	4	6
R3	avg. occur. of operands	6.05	11	9	4.5	4.55	5.88	7.55	7.25	13.75	4.92	3.92	4.66	7.23	4.66	6.29
R4	avg. occur. of operators	6.33	7.5	6.75	3.13	2.5	2.36	5.6	3.33	6.4	5.57	2.5	5.25	5.66	3.66	5.4
R5	avg. occur of constructs	3.81	4.33	4.88	3.42	1.8	2.8	4.25	4.66	5	5.5	3.75	3.66	3.5	2.75	3
R6	avg. occur of constants	2.66	4	4	0	0	0	0	0	0	8	0	3	3	0	3.25
R2	# semi-colons/comment	2.9	35	0	0	0	0	7.66	8	20	3.28	4.5	4.5	2.09	1.7	1.53
R1	Random Cases Testability		0.04	1				0.03	0.04	0.08	0.1	0.04	0.04	0.03	0.04	0.08
R1	All-Edges Testability		0.04	0.008				0.125	0.04	0.04	0.25	0.04	0.04	0.03	0.06	0.04
R1	Category-Partition Testability		0.33	0.125				0.027	0.04	0.04	0.2	0.04	0.04	0.25	0.04	0.04
	Lint Results															
R9	# warnings	12	4	2	46	42	42	3	3	6	8	5	5	5	3	6

4.4. *Networking Dataset*

In the second study, fifteen graduate students wrote four C programs for a graduate networking course (a protocol layer assignment by instructor Ken Calvert). The same specifications were implemented by each student [36]. The students had the same due date for the programs. The only specified order was that programs one and two were to be turned in prior to programs three and four. Static analysis of the sixty programs was performed using a commercially available tool (CMT++ tool by Testwell), with the same measures collected as for the five-programmer dataset[9]. Analysis of variance was applied. The assumption of normality of errors was met (as could be seen visually by the residuals plot), as was homogeneity of variance. The programs ranged in size from 53 to 468 lines of code [37].

5. Analysis and Results

The two experiments in Section 4 had some limitations and constraints that must be kept in mind when examining the results. Threats to validity are discussed in Section 5.1.

[9] Note that it could not be confirmed that the counting rules used by the commercial tool matched the manual counting rules used for the programs of the five professional programmers (the rules were not available for each measure provided by the tool). This constitutes an instrumentation threat to internal validity. This threat was minimized by verifying the rules that were available. Therefore, the results of the networking dataset should be viewed independently of the results for the professional programmer dataset.

The data were analyzed:

1. to look for an effect based on programmer (or author), and
2. to look for an effect based on the programs.

These areas are addressed in Sections 5.2 and 5.3. The overall hypothesis results are discussed in Section 5.4. Results for the networking dataset are presented in Section 5.5.

5.1. *Threats to validity*

The results from the experiments are subject to a number of limitations. Of particular concern are internal, external, and conclusion validity. An experiment can be said to possess **internal validity** if it shows a causal relationship between the independent and dependent variables. A number of influences can impact internal validity. For example, the programmers' solution to the Find program may have been influenced by the experimenter responses to questions on the specification. However, this could have only served to prompt the programmers to use a similar solution, resulting in similar programs. So any bias would have been in favor of the null hypothesis. History threats to internal validity were not applicable, as all programmers experienced the same current events [38].

To minimize selection threats to internal validity, each programmer was assigned the same three programs. Also, the order in which the programs were developed was not mandated. Another threat to internal validity is that of programmers biasing each other. Specifically, two of the programmers work in the same location and may have discussed the programs, causing their solutions to be similar (note that no correlation was found between the programs of these two authors). Each programmer was asked to work in isolation to minimize this threat.

Experimental mortality threats to internal validity were not applicable, as all programmers completed the assigned programs [38]. Design contamination threats to internal validity were not applicable, as there was not a separate control and experimental group [38]. There is a possibility of an instrumentation threat to internal validity (see section 4.4) due to switching to a commercial tool for the networking dataset.

External validity is possessed by an experiment if the causal relationship can be generalized to other groups, environments, etc. There were several threats to external validity in the experiments. First, a small number of programmers were used in the experiments. Each programmer wrote a small number of programs (three). Also, these programs were very small and were not very complicated to write. It is not certain that the results seen here would be observed for larger, more complex applications. Also, there is a potential bias since all the programmers are from the Washington metropolitan area and four of the five work for defense contractors.

Another limitation of note relates to the use of the PISCES tool. An incompatibility existed between four of the 15 programs and PISCES in that the tool was unable to compile the programs and therefore could not perform dynamic analysis. The incompatibility dealt with the use of certain programming constructs. One approach would have been to "forbid" the programmers from using these constructs. However, it was decided that such a prohibition could bias the solution to the specifications and may cause a programmer to stray away from their characteristic style and preference of constructs. The programs were compatible with the ATAC

compiler. Unfortunately, three of the four programs were written by the same programmer (Programmer 2). So, dynamic evaluation was performed for only four of the five programmers.

Construct validity is possessed by an experiment if the theoretical concepts being examined and the measuring procedure are in concurrence. A possible threat to construct validity deals with the testability measure provided by the PISCES tool. This measure might not discriminate well between test suites..

Conclusion validity is possessed by an experiment if there are no issues "affecting the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment [39]." Due to the small number of observations, the power of the statistical test may increase the risk that an erroneous conclusion is drawn. Also, if one views each dataset as a separate investigation, the significance level increases, meaning that significant results may be discovered due to chance.

5.2. *Analysis by programmer*

Facets (such as number of comments and average occurrence of operators) were deemed to be distinguishing if the ANOVA for that facet showed statistical significance for a programmer (that is, the p-value was 0.05 or less). The facets that were found to be distinguishing are listed in boldface in Table 4. The abbreviations used in the table are Sum of Squares (SS), degrees of freedom (DF), mean squares (MS), F-value (F), and p-value (p).

Table 4. Distinguishing Facets By Programmer.

Programmer Facets	Distinguishing?	SS	DF	MS	F	p
number of semi-colons per comment	No	408.4	4	102	1.1	0.41
average occurrence of operands	No	60.2	4	15	3.5	0.06
average occurrence of operators	**Yes**	**27.1**	**4**	**6.8**	**6.7**	**0.01**
average occurrence of constants	No	39.3	4	9.8	2.6	0.12
average occurrence of constructs	**Yes**	**9.0**	**4**	**2.3**	**5.0**	**0.02**
number of unique constructs	**Yes**	**64.4**	**4**	**16.1**	**40.2**	**2.4E-05**
number of unique operands	**Yes**	**238.2**	**4**	**59.6**	**10.3**	**0.003**
number of lint warnings	**Yes**	**3505**	**4**	**876.2**	**135.5**	**2.2E-07**
Testability						
Random cases	No	0.23	4	0.05	0.92	0.49
All-edge cases	No	0.02	4	0.005	1.70	0.24
Category-partition cases	No	0.04	4	0.01	0.90	0.50

Based on Table 4, it appears that five measures correlate with the author of a program, with a sixth measurement, average occurrence of operands, having borderline correlation with a p of 0.06 (recall that a p value of 0.05 or less indicates statistical significance):

1) average occurrence of operators
2) average occurrence of constructs
3) number of unique constructs
4) number of unique operands
5) number of lint warnings
6) average occurrence of operands (borderline)

Number of unique constructs is not a "normalized" measure because a very large, complex program will have more constructs than a very small, simple program. It should be dismissed in favor of the "normalized" facet *average occurrence of constructs*. Further, it is felt that using a single facet to determine the author of a program is less likely to have external validity. Therefore, a discriminator that combines several facets is preferred over a single facet.

The six potential discriminating facets listed above were examined further. Various combinations of the facets were built (e.g., product of the number of lint warnings and average occurrence of constructs). The combined facets were then compared to each of the values for a particular programmer and program. The combination was rejected if visual examination of the data did not indicate any correlation. If the combination appeared to have some merit (it seemed to correlate well with the data for at least two programmers), it was examined further.

5.2.1. *A Discriminator for Identifying Programmers*

By applying the process described above, it appears that three static measures have the potential for determining the author of a program: (1) average occurrence of operands, (2) average occurrence of operators, and (3) average occurrence of constructs. A number of methods for combining the factors were considered. One idea considered was to define and calculate the "average deviation," by computing the facet values for a programmer, then computing the value for each facet on the program, and averaging the ratio of the program's value over the programmer's value for all the facets. Although this measure was logically appealing, when applied to the data, the results were very unsatisfactory.

Due to these shortcomings, a single value discriminator was selected. Further, it was decided to use the product of the factors. Multiplying measures to build discriminators or "identifiers" is common. In information retrieval and requirements tracing, term frequency and inverse document frequency are multiplied to build a similarity measure used to see if a document is similar to a posed query. The Maintainability Index (MI) [40] multiplies a number of static measures such as average extended cyclomatic complexity per module [16], average count of lines of code (LOC) per module, and average percent of lines of comments per module to build a measure of the maintainability of a piece of code. The unit of this composite measure is not specified by Welker and Oman. However, prevalent use of MI by practitioners and researchers alike indicate that the lack of a specified unit has not deterred its use or importance. Component measures have also been used successfully as discriminators. For example, logistic regression models that multiply various variables and their coefficients are used to discriminate modules that are fault-prone [41, 25]. Multiple and simple regression models multiply measures (often of varying units). The effect is that the resulting discriminators will be "spread" further apart than if addition, subtraction, or some other operation was applied.

The result of this analysis is the following equation, called the Nicholson discriminator[10]. The Nicholson discriminator is the product of the aforementioned factors:

$$N_a = f1 * (f2^2) * f3$$

[10] The Nicholsons (brother and sister) worked for Newport News Shipbuilding in the '30s and '40s. Part of their job was to determine if ships were seaworthy, based on several pieces of data.

where N_a is the target Nicholson discriminator for programmer a, *f1* is the average occurrence of operands, *f2* is the average occurrence of operators, and *f3* is the average occurrence of constructs. The average occurrence of operators is squared because it had the lowest p-value of the three.

Using the mean values obtained for each programmer, the target N_a for each programmer is:

N_1= 1772.8
N_2 = 93.9
N_3 = 1149.8
N_4 = 374.6
N_5 = 436.5

Closer examination of the target N_a made it clear that a single value would not work well for the look-up table. Using an increment of + and – 5% of N_a, a range of N_a values were calculated for each programmer (subtract 5% from the value for the low end of the range, add 5% to the value for the high end). The increment can be expected to vary based on the number of programmers being distinguished. Additional evaluation is required to gain more general understanding about the increment. The ranges for target N_a values are shown below:

Programmer 1: Range of 1684.2 through 1861.4
Programmer 2: Range of 89.2 through 98.6
Programmer 3: Range of 1092.3 through 1207.3
Programmer 4: Range of 355.9 through 393.3
Programmer 5: Range of 414.7 through 458.3

To truly validate these ranges, additional programs written by the same programmers need to be evaluated. Thus, these data should be considered preliminary and inconclusive, but nevertheless quite interesting as a starting point.

Note that all the programmers have mutually exclusive ranges, although the range values for programmers 4 and 5 are close (programmer 4's high end of 393.3 and programmer 5's low end of 414.7). It could be possible, based on the limited data of these experiments, that facets can be used to pinpoint individual programmers. It is also possible that the close ranges of programmers 4 and 5 could cause problems for the method.

5.2.2. *A Second Dataset*

To further examine the validity of this work, programs from the networking dataset described in Section 4.4 were chosen. The ranges for target N_a were calculated and nine of the fifteen were unique ranges (they did not overlap with the five programmer ranges nor with each other). Of the six that overlapped, three were overlapping only slightly (for example, one student's N_a lower bound overlapped with programmer 3's upper bound, but by less than 5%). It could be possible that facets can be used to identify categories of programmers as opposed to pinpointing individual programmers. This is still a useful finding, if Quality Assurance methods can be shown to work better for certain types of programmers. That is, it might not be possible to take an anonymous program and report that "programmer X" was the author, but it might be possible to report that "a

category 1 programmer" was the author (where some relatively small number of categories has been defined). This would also be useful in narrowing down suspected authors of malicious or virus source code as well as in plagiarism detection.

5.3. *Analysis by Program*

The ANOVA showed no evidence of a difference among the measures for the programs Mid and Trityp. For example, the p-value for average occurrence of operands was 0.35. However, there was a statistically significant difference (p-value less than 0.05) for three measures for program Find: *average occurrence of operands; average occurrence of operators;* and *average occurrence of constructs.*

The testability facet was not found to be distinguishing; the testability results were very similar for all of the programs. Figures 3 through 5 present the testability results for the three programs (these can also be found in Table 3). The three test techniques listed left to right are random (shown as rand), all edge (shown as ae), and category partition (shown as catp). In Figure 3, for example, it can be seen that all five programmer versions of Mid, when executed using random test cases (mid/rand), had testability scores of 0.04. As can be seen, the programs written by programmer 1 often had a higher testability score than that of the programs of other programmers, regardless of the test technique. For example, Figure 3 shows that programmer 1's version of Mid tested with category-partition test cases had higher testability than the versions of Mid written by the other programmers. Figure 4 shows that programmer 1's version of Trityp also had a higher testability score than other versions of Trityp, for both random and category-partition testing. Figure 5 shows that programmer 4's version of Find had higher testability scores for random and all edge testing than the versions of Find written by other programmers.

It should be noted that many of the programs had 0.04 as their testability score, regardless of the programmer or the test technique. Although it is not clear why this value showed up so often, three possible explanations are: (1) test techniques do not work better for certain programming styles than for others, (2) testability does not capture the desired aspect, and (3) coincidence. In future studies, the defect detection effectiveness of various test techniques when applied to programs written by distinguishable authors will be examined.

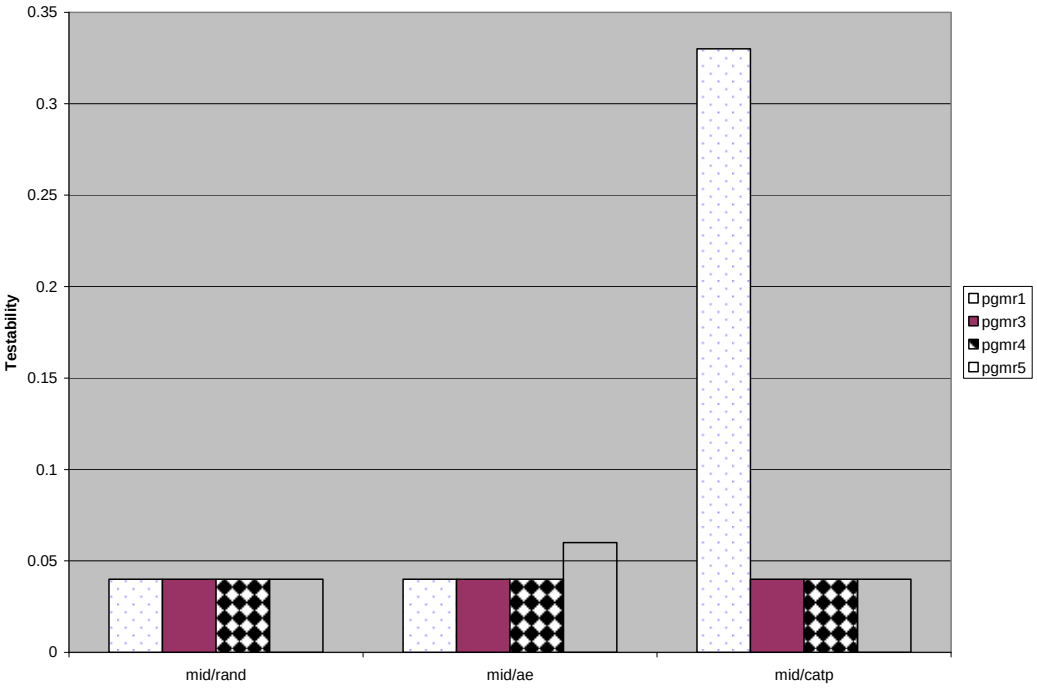

Figure 3. Testability Results for Program Mid.

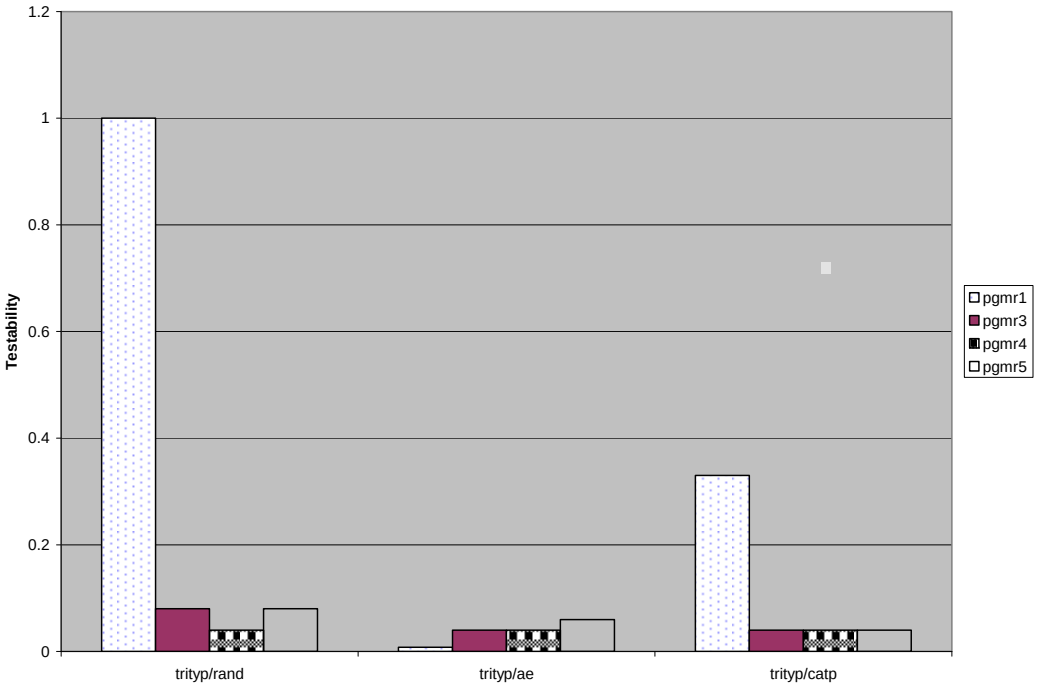

Figure 4. Testability Results for Program Trityp.

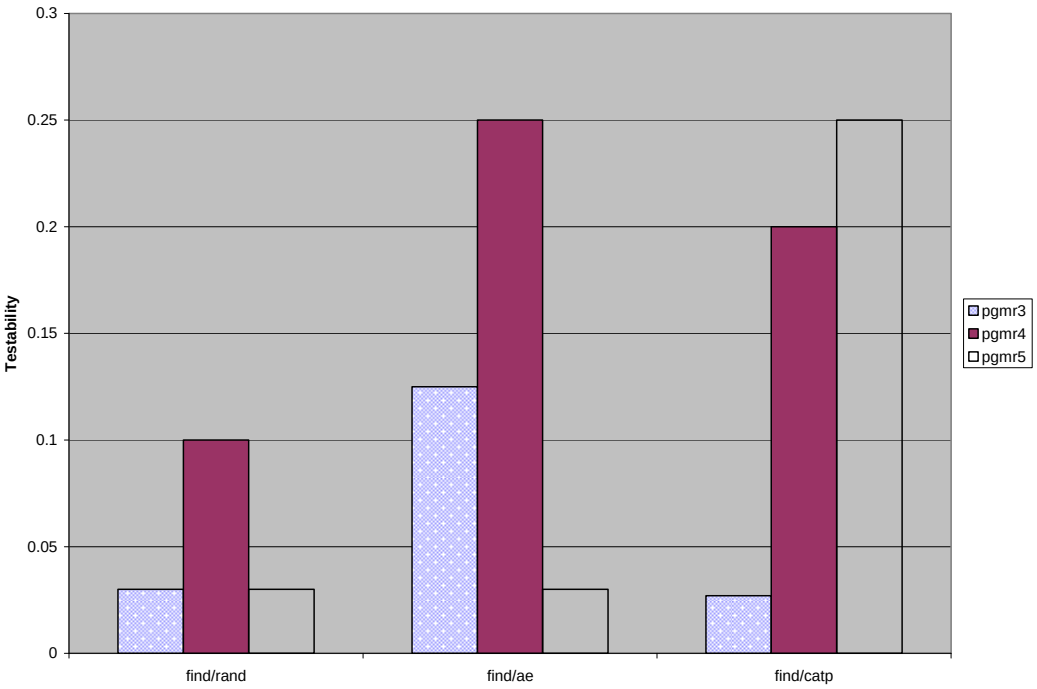

Figure 5. Testability Results for Program Find.

In examining order of development, one programmer did not report the order of program writing. One programmer wrote Mid last, one programmer wrote Find last, and one programmer wrote Trityp last. One programmer reported working on bits and pieces of Find while finishing the other two programs, claiming that the order was "difficult to say." Based on this varied order of development, it is believed that the order did not cause the differences between individual programmers.

5.4. *Hypothesis Results*

The general hypothesis for these experiments (the consistent programmer hypothesis) is that one or more characteristics exist for a program that can recognize the author. Evidence has been presented in Section 5.2 to support the notion that characteristics exist to identify the author of a program. However, the other aspect of the consistent programmer hypothesis, that some testing techniques are better suited to code developed by some programmers than others, was not supported experimentally. Therefore, the general hypothesis is not supported by the experiments.

Some of the specific hypotheses were supported and some were not. The results are listed below:

1. The dynamic measure of *testability* could not be correlated to a particular programmer.
2. The static measure of *number of semi-colons per comment* did vary for several individual programmers, but the hypothesis was not supported overall.
3. The static measure of *average occurrence of operands* is correlated with the individual programmer.

4. The static measure of *average occurrence of operators* is correlated with the individual programmer.
5. The static measure of *average occurrence of constructs* is correlated with the individual programmer.
6. The static measure of *average occurrence of constants* was not found to vary greatly based on programmer.
7. The static measure of *number of unique constructs* is correlated with the individual programmer.
8. The static measure of *number of unique operands* is correlated with the individual programmer.
9. The static measure of *number of lint warnings* is correlated with the individual programmer.

To examine the role of size within the measure set, a subsequent Principal Components Analysis (PCA) [46] of the original dataset was performed. This analysis identifies domains or components of interest by showing the sources of variance in a given dataset. The PCA showed that the measures do not all correlate with size. Table 5 shows the correlation between each identified principal component (called simply 'component') and the original measures (also called features in PCA) [37]. The largest values (when examining the absolute value) for a measure indicate the strongest relationship with a component (and are bolded in Table 5). Once the components have been identified, the relationships between measures and components must be examined. Based on this, the components are labeled or categorized.

Table 5. Rotated components of original dataset – five components selected.

	Component				
	1	2	3	4	5
V(G)	.318	.229	.179	.186	**.864**
Max v(G)	.096	-.309	.231	-.045	**.886**
LOCphy	.687	.682	.152	.058	-.031
LOCpro	**.806**	.488	.236	.021	.111
LOCbl	.488	.638	.043	.109	-.319
LOCcom	.462	**.856**	-.040	.074	-.093
N	**.966**	.168	.117	.122	.096
N1	**.947**	.180	.167	.114	.147
N2	**.973**	.154	.065	.128	.042
n	**.952**	.170	-.169	.161	.072
n1	.694	.010	-.163	.675	.075
n2	**.961**	.206	-.159	-.003	.066
V	**.974**	.166	.068	.106	.078
B	**.934**	.106	.120	.304	.060
D	.647	-.061	.195	**.719**	.053
E	**.950**	.111	.104	.224	.023
L	-.548	-.009	-.227	**-.757**	-.123
T	**.950**	.111	.104	.224	.023
Avg occ. opds	.318	-.138	.695	.577	.058
Avg occ. ops	**.822**	.231	.387	-.213	.220
H	**.959**	.168	-.167	.127	.063
H-N	.032	.000	**.957**	.039	.227
Comm/ locphy	.162	**.923**	-.143	-.034	-.024
Comm/ locpro	.101	**.954**	-.189	-.008	-.059
n2/locphy	.111	**-.847**	-.500	-.037	-.056
n2/locpro	.078	-.630	-.684	.011	-.227
n1/locphy	-.311	**-.762**	-.295	.431	-.107
n1/locpro	-.396	-.628	-.351	.491	-.219

The first component shown in Table 5 consists of lines of program code, Halstead's N (length), Halstead's N1, Halstead's N2 (N1 and N2 are summed to give N), Halstead's n (vocabulary), n2, V (Halstead's volume), B (Halstead's estimated number of bugs), E (Halstead's effort), T (Halstead's time), average occurrence of operators, and H (Halstead's predicted length) as shown by the relatively high values of these items in the table (in the [-0.7; +0.7] range). The longer the source code is, the higher the count of each of these measures. So, this can be seen as a component that represents the size or length of a program. The second component includes the number of comments, comments per physical lines of code, comments per program lines of code, number of unique operands per physical lines of code, and number of unique operators per physical lines of code. The first three measures, number of comments, comments per physical lines of code, and comments per program lines of code, are clearly related to comments. The latter two items, number of unique operands per physical lines of code and number of unique operators per physical lines of code, can be seen to tie to understandability also [37]. That is, the number of unique operators and operands drive the number of mental "lookups" a programmer must perform when maintaining code. Thus, it appears that this second component ties to understandability.

The third component includes only the difference between estimated and actual length (H – N). We label this component "error of predicted length." The fourth component contains D (Halstead's difficulty) and L (Halstead's program level). L is the relationship between Program Volume and Potential Volume. Only the most clear algorithm can have a level of unity. D is 1/L. A program grows more difficult as its level decreases. These both deal with difficulty. The final component includes McCabe's cyclomatic complexity and maximum cyclomatic complexity. These both deal with control flow [37]. The components are hence identified as follows:

- Component 1 – size/length
- Component 2 – understandability
- Component 3 – error of predicted length
- Component 4 – difficulty
- Component 5 – control flow

The eigenvalues of the correlation matrix from Table 5 are shown in Table 6. Only three components are needed to explain at least 86% of the overall variance. The principal factor method (with prior communality estimates of one followed by a varimax-rotation) is selected to establish the factor model for interpretation [42]. This adjustment gives weight to the highly correlated variables within each factor and lesser weight to any remaining variables [42]. That is, each variable will be highly correlated to exactly one factor and not to others.

Table 6. Principal components analysis – eigenvalues of the correlation matrix – Original Dataset.

	Eigenvalue	% of Variance	Cumulative %
Component 1	15.615	55.768	55.768
Component 2	5.246	18.734	74.502
Component 3	3.240	11.570	86.072
Component 4	1.789	6.388	92.460
Component 5	1.040	3.714	96.174

5.5. *Results for networking dataset*

As shown in Table 7, the distinguishing feature identified in the networking dataset was number of comments per program line. The Sum of Squares was 0.75, the mean square was 0.05, the F-value was 2.63, and the p-value was 0.006. Further analysis of the programs written by each author and an attempt to identify the authors based on this feature failed to identify authors or small sets of potential authors. The general hypothesis (static or dynamic facets of a program can be identified and used to recognize a program author) was not supported. This study is presented in detail in [37].

Table 7. Distinguishing Facets By Programmer – Networking Dataset.

Programmer: Features	Distinguishing?	SS	DF	MS	F	p
number of program lines	No	136106.40	14	9721.88	1.03	0.44
number of unique operators per program line	No	0.07	14	0.005	1.06	0.40
number of unique operands per program line	No	0.61	14	0.04	0.61	0.83
average occurrence of operators	No	1131.51	14	80.82	0.85	0.60
average occurrence of operands	No	278.85	14	19.91	0.67	0.78
number of comments per program line	**Yes**	**0.75**	**14**	**0.05**	**2.63**	**0.006**

6. Conclusions and Future Work

Experiments were undertaken to evaluate the general hypothesis that one or more facets (static or dynamic) exist for a program such that the author, or a small set of potential authors, of the program can be recognized, and that testing techniques correlate to programmers. A number of specific hypotheses were posited for individual characteristics such as "the static measure of number of unique operands is correlated with the individual programmer." The same three programs written by five different professional programmers as well as four C programs written by 15 different graduate students were examined. Support was not found for the general hypothesis that testing techniques correlate to programmer voice. This may have been due to the testability measure that was used to evaluate the techniques. As explained in Section 5.3, the testability values were often identical (0.04) regardless of the program, the programmer, or the testing technique applied. It is not clear that this measure supports the research. Therefore, for future work, the defect detection effectiveness of various test techniques will be examined rather than the testability. Evidence was found that the professional programmers exhibit voice in the professional programmer dataset, as shown by the strong correlation between five facets and programmer as well as weaker correlation with a sixth facet. In the networking dataset, strong correlation between one facet and programmer was found. Further work is required, particularly with a larger sample.

The results are encouraging. A number of new questions arose from this work. First, can an author be identified if he/she has modified someone else's code? There have been recent cases where virus writers copy an existing virus and make changes to it. There are certainly many cases where students copy wholesale sections of code from other students. How much code must be written by an individual for their "voice" to be evident? If an individual wrote 80% of the code, can that individual be identified? Also, it is not clear how this technique might be used to evaluate group software projects where several students have developed an application. It may be possible to approach this problem by focusing on modules as that is typically the level at which work is shared by programmers. The authors also plan to investigate how early a programmer develops voice. More detailed facets of testing techniques and inspection techniques are being examined to ensure that a lack of construct validity did not account for the lack of correlation. Section 4.3.2 used static analysis to count the number of operators, operands, and other program features used. It is possible that a finer

grained count, such as which specific operators were used, could lead to a more effective way to identify programmers. This would be an excellent topic for future work. An additional follow-up study could be to investigate whether factors such as experience, age, education, etc. affect these results.

Acknowledgements

This work was supported in part by the U.S. National Science Foundation under grant CCR-98-04111. We would like to thank Jeff Payne of Cigital for the use of PISCES as well as his technical assistance in unraveling some operator errors, and Bob Horgan at Telcordia Technologies for the use of ATAC. We would also like to thank the five programmers for volunteering their time to write three programs each. We thank Ken Calvert and his networking students. Thanks also to Kelly Noss Marcum of the Powell County School System for her insights on writing style and "voice."

References

1. A. R. Gray, P. J. Sallis, and S. G. 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, in *Proceedings of SE:E&P'98 (Software Engineering: Education & Practice Conference)*, (Dunedin, New Zealand, 1998), 252-259.
2. R. I. Kilgour, A. R. Gray, P. J. Sallis, and S. G. MacDonell, A fuzzy logic approach to computer software source code authorship analysis, in *Proceedings of ICONIP/ANZIIS/ANNES'97*, (Dunedin, New Zealand, 1997), 865-868.
3. Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward, Hints on test data selection: Help for the practicing programmer, *IEEE **Computer*** 11(4) (1978) 34-41.
4. Wikipedia, Iambic pentameter, http://en.wikipedia.org/wiki/Iambic_pentameter.
5. J. Ramsey and V.R. Basili, Analyzing the test process using structural coverage, in *Proceedings of the IEEE 8th International Conference on Software Engineering (ICSE 1985)*, (London, England, 1985) 306 – 312.
6. Basili, V. R. & R. W. Selby (1987) Comparing the Effectiveness of Software Testing Strategies. IEEE Trans. SE, 13. p. 1278-1296.
7. "Searching with style: Authorship attribution in classic literature", Y. Zhao and J. Zobel, Proceedings of the Australasian Computer Science Conference, G. Dobbie (ed), Ballarat, Australia, January 2007, pp. 59-68.
8. Juola, P. and Sofko, J. 'Proving and Improving Authorship Attribution Technologies,' Proceedings of Canadian Symposium for Text Analysis (CaSTA-04) "The Face of Text", Hamilton, ON, CA. November, 2004, pp. 1 – 5.
9. Kuˇcera, H. and Francis, W. N. (1967). Computational Analysis of Present-day American English. Brown University Press, Providence.
10. Burrows, J. F. (1989). 'an ocean where each kind. . . ': Statistical analysis and some major determinants of literary style. Computers and the Humanities, 23(4-5):309–21.
11. Burrows, J. (2003). Questions of authorships : Attribution and beyond. Computers and the Humanities, 37(1):5–32.
12. Juola, Patrick; Baayen, R. Harald, 'A Controlled-corpus Experiment in Authorship Identification by Cross-entropy,' Literary and Linguistic Computing, Volume 20, Supplement 1, 2005 , pp. 59-67(9).
13. A. R. Gray, P. J. Sallis, and S. G. MacDonell, Software Forensics: Extending authorship analysis techniques to computer programs, presented at the *Third Biannual Conference of the International Association of Forensic Linguists*, (at Duke University, Durham, North Carolina, 4 – 7 September 1997), p. 1 - 8.

14. P. Sallis, A. Aakjaer, and S. MacDonell, Software Forensics: Old methods for a new science, in *Proceedings of SE: E&P '96 (Software Engineering: Education and Practice Conference '96)*, (Dunedin, New Zealand, 1996) 367-371.
15. M. H. Halstead, *Elements of Software Science, (*Elsevier North-Holland, New York, 1977).
16. T. McCabe and C. Butler, Design complexity measurement and testing. *Communications of the ACM* 32(12) (1989) 1415–1425.
17. L. Prechelt, G. Malpohl, and M. Philippsen, Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science, vol. 8, no. 11 (2002), 1016-1038
18. S. Schleimer, D. Wilkerson, and A. Aiken, Winnowing: Local Algorithms for Document Fingerprinting, in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (San Diego, CA, 2003) 76-85.
19. P.W. Oman and C.R. Cook, Programming style authorship analysis, in *Proceedings of the Seventeenth Annual ACM Conference on Computer Science: Computing trends in the 1990's*, (Louisville, Kentucky, 1989) 320 – 326.
20. E.H. Spafford and S.A. Weeber, Software Forensics: Can we track code to its authors? Computers & Security 12 (1993) 585-595.
21. I. Krsul, Authorship Analysis: Identifying the Author of a Program, Purdue University, May, 1994, Technical Report CSD-TR-94-030.
22. S. G. MacDonell, A. R. Gray, G. MacLennon and P.J. Sallis, Software Forensics for discriminating between program authors using code-based reasoning, feed-forward neural networks, and multiple discriminant analysis, in *Proceedings of the 6th International Conference on Neural Information Processing ICONIP'99, ANZIIS'99, ANNES'99, and ACNN'99*, (Perth, Western Australia, 1999), 66-67.
23. J. Huffman Hayes, N. Mohamed, and T. Gao, The Observe-Mine-Adopt Model: An Agile Way to Enhance Software Maintainability, *Journal of Software Maintenance and Evolution: Research and Practice*, ***15***, *5* (October 2003), 297 – 323.
24. J. Huffman Hayes, S. Patel, and L. Zhao, A Metrics-Based Software Maintenance Effort Model, in *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, (Tampere, Finland, March 2004), pp. 254-258
25. A. Nikora and J. Munson, Developing Fault Predictors for Evolving Software Systems, in *Proceedings of the 9th International Software Metrics Symposium (Metrics2003)*, (Sydney, Australia, 2003), pp. 338 - 350.
26. Christian Collberg and Clark Thomborson, Watermarking, tamper-proofing, and obfuscation - Tools for software protection, University of Arizona Computer Science Technical Report number 2000-03, (February 10, 2000) 5-7.
27. J. M. Voas, PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering* 18(8), (1992) 717-727.
28. J. R. Horgan and S. London, 1992. A data flow coverage testing tool for C, in *Proceedings of the Symposium of Quality Software Development Tools*, (New Orleans, Louisiana, 1992) 2 - 10.
29. B. Beizer, *Software System Testing and Quality Assurance*, (Van Nostrand Reinhold, New York, NY, 1984) 45-51.
30. T. J. Ostrand and M. J. Balcer, The category-partition method for specifying and generating functional tests. *Communications of the ACM* 31(6), (1998) 676-686.
31. Mats Grindal, Jeff Offutt and Sten F. Andler. Combination Testing Strategies: A Survey. J*ournal of Software Testing, Verification and Reliability*, Wiley, 15(2):97-133, September 2005.
32. P. G. Frankl, and S. N. Weiss, An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria, in *Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification*, (1991) 154-164.
33. Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.
34. R. H. Baayen, A. Neijt, H. van Halteren, and F. Tweedie, An experiment in authorship attribution. In *Proceedings of the 6s Journies Internationales dAnalyse de Donnies Textuelles (JADT),* (Malo, France, 2002), pp. 29 - 37.
35. A. J. Offutt, Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodologies* 1(1), (1992) 5-20.
36. K. Calvert. "Programming Assignment 0: Protocol Layers, Version 1.2." CS 571 Computer Networks, University of Kentucky, Spring 2003a. Assignment available upon request.
37. Jane Huffman Hayes, Authorship Attribution: A Principal Component and Linear Discriminant

Analysis of teh Consistent Programmer Hypothesis, International Journal of Computers and Their Applications (IJCA), VOLUME 15, NO. 2, June, 2008, p. 79 – 99.
38. Experimental designs for research, http://www.csulb.edu/~msaintg/ppa696/696exper.htm.
39. C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen, Experimentation in Software Engineering, (Kluwer Academic Publishers, London, England, 2000).
40. K.D. Welker and P.W. Oman, Software Maintainability Metrics Models in Practice, *Journal of Defense Software Engineering*, 8, 11, (November/December 1995) 19-23.
41. G. Denaro, S. Morasca, and M. Pezze, Deriving Models of Software Fault-Proneness, in *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)*, (Ischia, Italy, July 2002), pp. 361 - 368.
42. N. Hanebutte, C. Taylor, and R. Dumke. "Techniques of Successful Application of Factor Analysis in Software Measurement." Empirical Software Engineering, Volume 8, p. 43-57, 2003.
43. Patrick Juola, John Noecker, Mike Ryan and Mengjia Zhao, Authorship Attribution for the Rest of Us, Proceedings of Digital Humanities 2008, http://www.ekl.oulu.fi/dh2008/.
44. S. Kullback, The Kullback-Leibler distance, The American Statistician 41:340-341, 1987.
45. Kullback–Leibler divergence, Wikipedia, http://en.wikipedia.org/wiki/Kullback-Leibler_divergence.
46. Pearson, K. (1901). "On Lines and Planes of Closest Fit to Systems of Points in Space". Philosophical Magazine 2 (6): 559–572, http://stat.smmu.edu.cn/history/pearson1901.pdf.
47. Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. Annals of Eugenics, 7, 179–188
48. Capability Maturity Model Integration (CMMI), Software Engineering Institute, http://www.sei.cmu.edu/cmmi/index.html.