ISSN: 2473-4901
v10 n6152

A Study of Software Metrics, Student Learning, and Systems Development Analytics

Seth J. Kinnett
seth.kinnett@colostate.edu

Jon D. Clark
jon.clark@colostate.edu

Computer Information Systems, College of Business
Colorado State University
Fort Collins, CO 80523, USA

Abstract

The purpose of this paper is to identify accepted systems development metrics and trace the ability of students learning relative to programming behavior and outcome. In particular, metrics identified by Maurice H. Halstead and Thomas J. McCabe were applied to a particular intermediate level Java programming exercise, part of a first course in Computer Information Systems. In addition, various approaches to solving a specific, timed exercise will be analyzed and subjected to a standard grading rubric used in the class. Counterintuitive results included: 1) all programs exceeded McCabe's cyclomatic complexity threshold and; 2) expected time to implement exceeded expectations.

Keywords: Software Metrics, Halstead, McCabe, Student Learning, System Development Analytics

1.INTRODUCTION

The purpose of this paper is to provide some insights into student learning in the computer programming domain. Having access to a large file of student assignment submissions, we hope to assess student learning and interrogate the implications of well-known metrics provided in the literature. In particular, we are fortunate to have at our disposal McCabe's suite of products (*BattleMap IQ*) which will be run against submitted programs in Java, generating both McCabe and Halstead metrics. The course in which these assignments are used is CIS240 named Application Design and Development and is the first programming course that undergraduate Computer Information Systems majors take. The 16-week course introduced students to object-oriented programming fundamentals using Java, spanning the concepts outlined in Table 1. We spent roughly two weeks of class per module.

Module Number	Topics
1	Course Overview & History of Programming Languages
2	Java Fundamentals (data typing, variables, constants)
3	Selection Statements (if/else/else if/switch)
4	Loops (while, do-while, for)
5	Methods & Method Overloading
6	Arrays (one and two-dimensional)
7	Classes & Objects (data encapsulation, constructors)
8	String Manipulation & File I/O

Table 1: Course Topic Summary

The course includes the use of programming using graphical user interfaces (GUIs) through

ISSN: 2473-4901
v10 n6152

the incorporation of selected methods from the JOptionPane class (javax.swing.JOptionPane) in addition to utilizing the console for inputs and outputs. Students write code using the Eclipse IDE. Prior to defining our specific research questions, a general discussion of metrics is in order.

There have been three distinctly different approaches to systems development metrics identified in the literature during the late 1970's and early 1980's (Navlakha, 1987). Arguably, Dr. Maurice H. Halstead, a professor of Computer Science at Purdue, developed a foundation for software science (Halstead, 1977) that can also be applied to domains other than computer code. Interestingly, with degrees in physics and physical science, he was committed to defining the science of Computer Science while at Purdue University. Thomas McCabe, a graduate student of mathematics at the University of Connecticut, developed metrics of control flow in programs (McCabe, 1976), including the well-known cyclomatic complexity, among others. In addition, McCabe's later tools for assessment included Halstead metrics as well. Finally, A.J. Albrecht's function points target project management but do not assess quality of implementation. There have been, however, detractors of software metrics (Jones, 2017) who have characterized the current state as a mess!

According to Jones, software applications are among the most expensive and error prone products of human effort. In order for this record to be improved, software needs to be accurate and reliable. An example of the current state of development and management practice is the CrowdStrike debacle (Fung, 2024) in which over 5 billion dollars of damage and 8.5 million devices were affected.

The relative positioning of these three approaches can be summarized as follows: Halstead metrics are based on tokens for operators and operands and some information theoretic functions that allow one to estimate errors, information content and mental processing time. Thus, Halstead's metrics can be applied to natural languages and programming. McCabe's work, on the other hand, focuses on control flow of a program to measure complexity, estimate errors and address coverage test strategies. Thus, this approach is more program domain specific. Albrecht's function points appear to have a loose theoretical basis, but due to empirical results from many case studies, offer a practical approach to determining the effort involved in latter stages of the systems development lifecycle.

Halstead

Halstead metrics are based on token counts of operators (verbs) and operands (nouns); information theory in terms of bits, the standard measure; and Stroud's (Stroud, 1956) information processing rate taken as a constant of 18 bps. The calculation of the various metrics depends on the counting strategy appropriate to a given language, and many programming languages have well defined strategies established including COBOL, Java, C and C++. Token counts are determined as follows:

n1 = number of distinct operators

n2 = number of distinct operands

n (vocabulary size) = total number of distinct tokens (operators + operands)

N1 = total number of occurrences of operators

N2 = total number of occurrences of operands

N (program length) = total number of tokens (operators and operands)

The derived metrics are as follows:

V (program volume) = $N*\log_2(n)$: program length times bits of information in vocabulary

D (program difficulty) = (n1/2)*(N2/n2) or 1/L: as volume increases, level decreases and difficulty increases

E (program effort) = D*V

T (time to implement) = E/18: effort in bits divided by human binary discriminations per second

I (intelligence content) = V/D

McCabe

McCabe's complexity measures were based on graphs of control flow, where nodes represent program statements and edges (arcs) represent the flow. Obviously, statements that determine decisions produce branches in the graphs and the count of various paths are an important determinant of complexity. These metrics are far more domain specific to procedural programming than Halstead's approach but are not predictive of effort across the stages of systems development.

The control graph produces the following metrics:

E = number edges of the graph

N = number of nodes of the graph

P = number of connected components (program exit points)

The derived metrics are as follows:

v(G) (Cyclomatic Complexity) = E-N+2: number of edges less number of nodes plus the number of connected components

ev(G) (Essential Complexity) = 1<=ev(G)<=v(G): based on reduced control flow graph

Interpretation of v(G) thresholds by McCabe:

- 1-10: simple procedure, little risk
- 11-20: more complex, moderate risk
- 21-50: complex, high risk
- >50: untestable code, very high risk

The Essential Complexity, ev(G) is produced by removing all of the structured programming primitives. These include 1) sequence; 2) selection statements, including *if* and *case* statements; 3) iteration constructs, including *while, do,* and *for*.

Others

There have been other attempts to define metrics that do not fit into these three categories. Several have involved using cognitive characteristics as a predictor of systems development performance at various stages including systems design (Clark, 1982 and 1983)), program maintenance (Clark & Khalil, 1989), and early-stage programming skill development in COBOL (Clark & Gibson, 1988).

2. RESEARCH DESIGN & QUESTION ARTICULATION

We chose, specifically, to focus on McCabe and Halstead metrics. We evaluate these families of metrics both individually and collectively seeking both to identify insights unique to each family of metrics while also seeking to understand where the metrics share communal results. Our data sample entails a collection of 30 student samples of a timed, in-class coding exercises. The metrics we generated for a subset of student submissions in this paper were taken from a particular selection of programming exercises, known as in-class exercises (ICEs). ICEs are combinations of short programming assignments, designed to be completed by the expert student in about 30 minutes and by all competent students within a 1.25-hour class period. The purpose of these exercises – in addition to allowing students opportunities for knowledge sharing and collaboration – is to provide *wake-up calls* to identify areas where students may need additional work. Since the exercises are designed to be solvable in 30 minutes or less, students who struggle to complete them in 1.25 hours should be alerted to the need for remedial work in a given area. Students are permitted to ask questions and at least one undergraduate, who previously completed the course, circulates to address student questions.

Our first research question recognizes that all of the metrics generation approaches we outlined rely on several assumptions or rules of thumb—McCabe and Halstead included. For example, regarding Halstead's metrics, one notable rule of thumb appears in the context of the *Stroud number*, which represents the number of binary discriminations per second (bps) assumed to be available to the human brain. Halstead's metrics — particularly the T (time) metric – utilizes a Stroud number of 18. Assessing the implementation time for student programming activities has important pedagogical implications for timed programming activities, such as the ones we examine. Namely, we ought not assign programs that are reasonably likely to take longer than the time allotted. Our first research question seeks to test the appropriateness of the default value for the Stroud number against actual timed student programming exercises.

RQ1: *What value of the Stroud number corresponds to the mean observed T (time) calculation from our student sample?*

Another related point and an implied reality in software development is that short, simple programs are more reliable and maintainable than longer, more complicated programs. In the context of an introductory Java programming course, we deliberately seek to present manageable assignments, which would be somewhat simple to implement. The logic underlying McCabe's metrics – particularly the measures of cyclomatic complexity (v(G)) and essential complexity (ev(G)) provides a more nuanced understanding of reliability and maintainability, respectively. While intuitively we might assume that our student-generated programs would be sufficiently bounded so as to have negligible scores in these areas, we propose our second research question to illuminate the

realities of student code reliability and maintainability.

RQ2: *How do student programs rank in McCabe's measures of cyclomatic complexity (v(G)) and essential complexity (ev(G)), and what – if any – characteristics of code are implicated in these findings?*

Although Halstead's and McCabe's metrics were developed independently and are based on drastically different analytical frameworks, we nevertheless suspect that certain metrics might have complementary utility with one or more McCabe metrics enriching Halstead metrics or vice versa. Accordingly, our final research question seeks to identify aspects of Halstead's and McCabe's metrics, which are complementary to one another.

RQ3: Which metrics from Halstead's analytical family are complementary to McCabe's or vice versa?

The remainder of our paper is structured as follows. First, we provide more detail surrounding the programming course, sample coding exercise, inclusion/exclusion criteria for our samples, and approach to addressing our research questions. Next, we present the key findings across the Halstead and McCabe metrics, along with a qualitative follow-up exercise. Finally, we discuss pedagogical implications, conclusions, and proposed future research.

Write a Java program that plays the game Rock, Paper, Scissors.
The rules are as follows:
•Rock (0) beats scissors (2)
•Scissors (2) beats paper (1)
•Paper (1) beats rock (0)

At the start of the program, the program must ask the user for their name. The program should then ask how many rounds the player wants to play. The program will then prompt the user to choose rock, paper, or scissors and randomly choose a value for the computer player. It will determine the winner of that hand, display the results, and keep track of the number of hands won by the player, the number won by the computer, and the number of tie games.

Use JOptionPane for all inputs and outputs.

Figure 1: ICE04 Assignment

We selected *ICE04: Loops* for our examination. It struck a balance of being reasonably complicated while also only requiring a single main method, which simplified analysis. Although the new topic being evaluated was loops, the program still required students – as is common in programming – to make use of all of the previous concepts they had learned. Specifically, the purpose of ICE04 was to assess student understanding of loops through the implementation of a rock-paper-scissors game, as shown in Figure 1.

The optimal solution used a *while* or *do-while* loop to elicit user input for number of rounds to play, while using a *for* loop to handle the gameplay for the user-specified number of rounds. For each round, students would need to implement if/else an/or if ladders and – optionally – *switch* statements to compare user guesses to random integers generated by the program to ascertain whether the player won, lost, or tied the computer for each round. In total, students can earn 10 points for successfully implementing the program. Table 2 contains the grading rubric used for the assignment, and the number of points possible for each item.

Proper coding habits including indentation & comments (1)
Proper compilation (no errors) (2)
Either *for* or *while* loop implemented properly to loop for the number of rounds specified by the user (2)
Correct use of if/else-if or switch to process user's choice each time (1)
Correct use of nested if/else-if to evaluate computer's choice (1)
Correctly implements counter variables to track computer wins, player wins, ties (1)
Correct computation of winner using if/else-if/else to evaluate the counters (1)
Correct generation of output using string concatenation (1)

Table 2: ICE04 Grading Rubric

We extracted a sample of student submissions across three semesters of our introductory programming course. We only retrieved the source code of students, who had achieved perfect scores (10/10) on the assignment. This yielded a sample of 30 submissions (N = 30). By selecting only perfect scores, we obtain a – on the surface – homogenous collection of coding samples. Any observed differences throughout the sample will be particularly notable given our decision to select only fully working programs without errors, which had yielded full credit across our grading rubric.

ISSN: 2473-4901
v10 n6152

We utilized the software analysis application *BattleMap IQ* for this project. After extracting the student submissions, we loaded each student project into *BattleMap IQ* and generated Halstead (Long) metrics, McCabe metrics, and a scatterplot of McCabe's v(G) and ev(G) metrics. We next exported these results to a text file and used the file to populate a spreadsheet. Alongside the system-generated Halstead metrics, we added in the elapsed time in minutes between when the exercise was assigned and when each student submitted it, thus providing an actual time metric, which allowed us to proceed with our comparisons against Halstead's T values. We loaded the entire data set into IBM's SPSS statistics software in order to perform comparisons of means across selected metrics as outlined below.

3. RESULTS

Analysis of the Stroud Number

As noted, Halstead's metrics span a variety of analytical points of interest. One area of focus surrounds our comparison of Halstead's measurement of development time (T) to the elapsed time it took students to complete our selected assignment. By comparing students' elapsed time to complete the program against Halstead's T (time) calculation, we observed that Halstead's T, converted to minutes, overestimates the time to complete the program in 83% of the samples. Indeed, the correlation between Halstead's T and the time we observed students took to complete programs approaches zero ($r = .03$). Only 17% of student submissions had a longer elapsed time than would have been predicted from the Halstead T metric.

By revisiting the formula for calculating T, we note that it represents Halstead's effort (E) divided by the *Stroud number*, representing an estimate of the number of bps the human brain can process. Halstead's T uses a default value of 18 for the *Stroud number*.

Student Elapsed Time		**Halstead's T**	
Mean (mins)	Standard Deviation	Mean	Standard Deviation
71.47	7.95	111.33	34.53

Table 3: Elapsed Time Compared to Halstead's T

The mean value for Halstead's T in our samples was 111.33, while the mean student time was 71.47. By calculating the mean value for effort (E), 120,235.3, we verified the T calculation, then began iteratively recreating Halstead's T calculations with progressively greater values for the Stroud number, noting that 18 was too small. We arrived at a near match when we reached a Stroud value of $k = 28$, 10 more bps than the default value and which yielded a modified Halstead's T of 71.57, a difference of only 0.1 from our observed mean value for time. Rather than discrediting Halstead's T, we believe these findings reinforce the validity of T when the proper value for the Stroud number is employed. Accordingly, we can answer our first research question, noting that a Stroud number value of 28 binary discriminations per second reflects the mean program development time. Incidentally, scientists have suggested that brain processing rates can vary depending on a number of factors including attention and task familiarity. Rates in excess of 100bps have been posited.

Evaluating McCabe's Metrics

Turning to our second research question, we analyzed McCabe's primary metrics of cyclomatic complexity (v(G)) and essential complexity (ev(G)). The former measurement assesses reliability while the latter measures maintainability. While McCabe noted four ranges of interest for v(G), the threshold of 10 separates low risk procedures from medium risk, with values above 20 and 50 separating the comparatively higher risk ranges. In simplest terms, code samples yielding v(G) values less than 10 are reliable and values greater than ten are unreliable. Similarly, a threshold of 4 was proposed for Essential Complexity (ev(G)). Values for ev(G) greater than four are considered unmaintainable, while those less than four are considered maintainable. A review of our student code sample reveals that while no code sample meets the standard for reliable code (v(G)), a subset meets the standard for maintainable ($N_{maintainable} = 13$), while the remainder were considered unmaintainable ($N_{unmaintainable} = 17$). Figure 2 depicts a scatterplot, generated from the *BattleMap IQ* program showing the four quadrants of reliable/maintainable, unreliable/maintainable, reliable/unmaintainable, and unreliable/unmaintainable, based on the prior thresholds.

ISSN: 2473-4901
v10 n6152

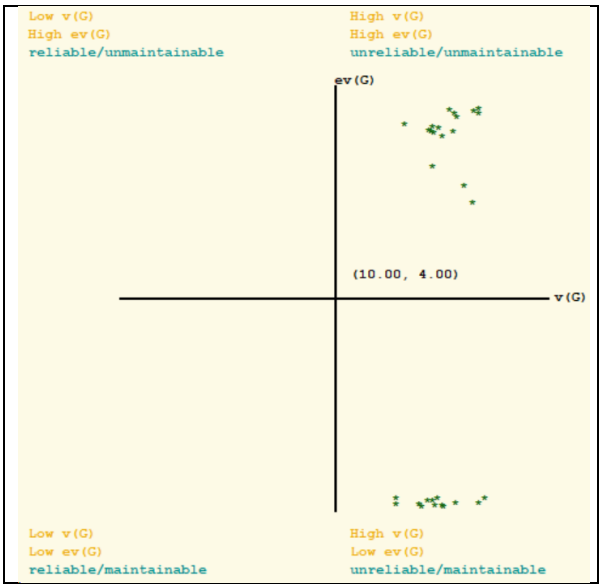

Figure 2: v(G) & ev(G) Scatterplot of Student Submissions

Table 4 outlines the percentage of samples in each category based on a view of their v(G) scores, confirming that the unreliable group had a higher proportion of high-risk code and lower proportion of moderate risk code than the reliable group.

	v(G)	
	Moderate Risk (11-20)	**High Risk (21-50)**
Reliable	77%	33%
Unreliable	59%	41%

Table 4: Assessment of v(G) Ranges

As the essential complexity metric (ev(G)) seeks to measure the degree of *structuredness* of code – a term both somewhat intuitive yet challenging to delineate with examples – we performed a qualitative analysis of a subset of five maintainable and five unmaintainable code samples. We selected the maintainable choices at random, then selected five unmaintainable samples generally at random, assuring only the same mixture of male and female students as we obtained through the random selection of the maintainable sample.

Qualitative Comparison of Working Student Programs

Each set of programs was then subjected to an exploratory qualitative analysis feature which included -- but was not limited to – code organization, structure/readability, and inventory of Java constructs used in the implementation. While the analysis was both tedious and unstructured, the exploratory nature of the topic necessitated such an approach, which ultimately revealed students' paths to understanding and development of skills.

At an abstract level, the assignment was one that depended on a set of 12 to 15 variables of several types, a loop to iterate through a number of game rounds, and a set of decisions to match outcomes of each round. In addition, output was required to display the results of each round, as well as a game-wide summary at the game's conclusion. The maintainable (M) samples differed from the unmaintainable (UM) samples across the dimensions of program length, types of constructs used, variable organization & scoping, and structure & readability. First, the M group's programs were more consistent in length, spanning 2.5 to 3.4 pages, while the UM group varied from 2.75 to 4.5 pages.

While both groups made use of looping constructs, only the M group incorporated switch statements. Of particular note, only the UM group employed compound *if* statements, often stringing together at least two conditionals separated by *&&* to evaluate the rock-paper-scissors choice combinations. No one in in the M group used such constructs and instead used single clauses and relied on combinations of *if* and *switch* or nested *if* statements. We believe the employment of compound *if* statements may have been one of the main reasons why higher ev(G) values were obtained for the UM group. One of the UM samples even tried to simulate a loop using a counter integer and an if statement instead of using a simple looping construct.

Regarding code organization and variable declarations, the M group was far more consistent with variable declarations. 4 out of 5 of the M samples defined variables primarily at the beginning of the main method. The UM group tended to define variables within loops or other code blocks whenever they realized they were needed, thereby limiting their scope and causing re-declaration for every pass of a loop, for example. Additionally, the M group adhered to better readability and indentation practices compared to the UM group. Collectively, our evaluation of the McCabe metrics and qualitative code sample assessment address our second research question.

Halstead and McCabe Complementary

Regarding RQ3 and whether Halstead and McCabe produce similar results based on information theory as well as control flow, the answer appears to be yes based on Table 5, but not at a statistically significant level.

ISSN: 2473-4901
v10 n6152

Group	ID	N	D	v(G)	ev(G)
M	1	585	43.32	18	1
	2	509	42.81	17	1
	3	475	31.89	16	1
	4	413	26.77	13	1
	5	466	38.0	13	1
		M=490, SD=63	M=37, SD=7	M=15, SD=2	M=1, SD=0
UM	14	503	40.00	21	18
	15	529	43.73	17	8
	16	462	39.15	17	14
	17	731	49.67	22	7
	18	434	37.29	18	14
		M=532 SD=117	M=42, SD=5	M=19, SD=2	M=12, SD=5

Table 5: Metrics for Qualitative Samples

Indeed, we performed both nonparametric tests and independent samples t-tests through the employment of bootstrapping, comparing the maintainable and unmaintainable group across the elapsed time, Halstead metrics, and McCabe metrics. Although the nonparametric test advised retention of null hypotheses (i.e., u_1-u_2=0) for all metrics, we persisted in employing independent samples t-tests with bootstrapping across 1000 simulated samples. Bootstrapping is a technique used to simulate a higher sample size based on the distribution of the existing sample. Our intention is to expand the "true" sample size by evaluating student programs in subsequent offerings of this programming course. Of note, the mean calculated elapsed time – not to be conflated with Halstead's T values – between the maintainable and unmaintainable groups had statistically significantly different means evaluated against 95% confidence intervals: CI_{95} = [-11.9,-1.06] , (*p = .03*).

Essentially, students who wrote unmaintainable code took longer to do it. Despite the lack of statistical significance in mean difference tests across other metrics, we nevertheless note that the samples we selected for our qualitative analysis do indicate patterns suggesting the two families of metrics are complementary. In particular, we observed higher mean values for Halstead's N and D in the McCabe unmaintainable sample than in the maintainable sample. Revisiting this question with larger sample sizes is a part of our future research agenda.

4. DISCUSSION/LIMITATIONS

One of the most surprising findings from this examination is the poor scoring we observed on McCabe's measure of cyclomatic complexity. In particular, the observation that every single student program obtained a v(G) value above 10 suggests none of the programs could be considered reliable by that standard. We found this surprising in that these programs are highly targeted to the implementation of a simple rock-paper-scissors game. If such simple exercises result in unduly cyclomatically complicated code, what hope do large-scale enterprise applications have of meeting this one of McCabe's standards?

One avenue for our future research encompasses the evaluation of even simpler programs to determine just what program characteristics could meet McCabe's v(G) standard. While we do not necessarily question the formulation of v(G), we believe our examples could lead to a more relaxed threshold for where code should be cleaved along the reliable vs. unreliable axis. One wonders if the threshold McCabe suggested for *reliable* levels of cyclomatic complexity may be too strict.

Next, the dispersion in results across the essential complexity metric (ev(G)) has pedagogical implications. Namely, since this metric measures the *structuredness* of code, with higher values indicating higher quantities of unstructured code constructs, it is clear that both highly unstructured and more structured code is suitable in achieving a perfect score via our rubric, yet some students are clearly – based on the implicit reliability of the metric – writing better code than others. On one hand, students have not – at this point in the course—yet been introduced to custom methods, parameter passing, and the general enablement of modular design. This makes it even more notable, however, that the sample cleaved along the boundary of what McCabe considered maintainable compared to unmaintainable code.

In an introductory programming course, a natural tendency exists to focus on ensuring students can form algorithms, understand syntax, and tie business requirements to appropriate solutions. One contribution of this paper is the illumination of a second layer of competency considerations: namely, the importance of *best approach solutioning*. In a ten-point assignment like the one we used here, point allocation options are limited. Longer assignments like the biweekly 20-point programming assignments in our course provide more opportunities to reward students for choosing solutions, which best minimize cyclomatic complexity and essential complexity, as two potential metrics.

The examples explored here confirm that these metrics are not simply theoretically interesting

ISSN: 2473-4901
v10 n6152

but correspond to important decisions made in the development of coding solutions. By identifying a collection of best practices in an introductory course – far from overwhelming students – we posit that it will actually be freeing and allow students to work more efficiently and effectively. One of the most concrete findings from this paper surrounds the inferiority of compound *if clauses* compared to using nested single-clause *if* statements or combining *switch* and single-clause *if* statements. Rewarding the choice of the latter solutions would be a good starting point for awarding bonus points in a revised assignment rubric. Advising students to choose better structured program statements is one curricular adjustment we plan to implement following this research. We plan to focus on these constructs to reaffirm our findings in future studies.

Another important contribution of this research lies in the revelation that not only were the programs with high ev(G) statistics unmaintainable, but also that this group had statistically significantly higher development times compared to the mean development times of the students, who achieved ev(G) values under the 4.0 threshold proposed by McCabe. While we are limited to only drawing inference from correlation, we found it notable that students writing comparatively sub-par code also took longer to write it, despite still achieving perfect scores per the grading rubric.

Finally, although we employed as rigorous a methodology as possible, we acknowledge limitations of this research, which are primarily centered in the sample data itself. Namely, students are prohibited from the use of unapproved resources (e.g., Chegg, CourseHero, ChatGPT) – and assignment graders are trained to spot techniques with deviation from those taught in the course – it remains possible a subset of these observations represent bad-faith attempts at completing the assignments, which could skew the results. The use of bootstrapping sought to mitigate the additional limitation of a relatively small sample size.

5. CONCLUSIONS

Software metrics play an important role in ascertaining a variety of dimensions of software quality, including complexity, reliability, and maintainability. Similarly, the assumptions underlying the computations of these metrics are important both to understand but also to challenge and verify. In this paper, we selected a sample of student programming assignments from an introductory Java programming course. We generated a bank of metrics from both the Halstead and McCabe categories of software metrics.

Our findings have implications for both software science in general and the teaching of computer programming in particular. First, we illuminated an important nuance pertaining to grading rubrics: not all perfect scores are created equal. We plan to modify our teaching of the course to give *preferred* vs. acceptable coding techniques derived from our findings of this paper. We recommend instructors gradually incorporate bonus points for certain stylistic practices that align with more reliable and maintainable code. Ultimately, this research affirms the use of Halstead and McCabe's approaches to software science as important, with notable pedagogical implications.

6. REFERENCES

Clark, Jon D. (1982, October), A Psychometric Evaluation of Yourdon's Design Methodology, *SIGCHI Bulletin*, 14(2), 9-12. https://doi.org/10.1145/1044759.1044761

Clark, Jon D. (1983, July), A Psychometric Evaluation of the Use of Data Flow Diagrams, *SIGCHI Bulletin*, 15(1), 3-6. https://doi.org/10.1145/1044779.1044780

Clark, Jon D. & Gibson, Michael Lucas (1988, November 21-23), Measuring Program Complexity Using Halstead's Metric, *Proceedings of the Annual Meeting of the Decision Sciences Institute*, Las Vegas, Nevada.

Clark, Jon D. & Khalil, Omar (1989, February), The Influence of Programmer's Cognitive Complexity on Program Comprehension and Modification, *International Journal of Man-Machine Studies*, 219-236. https://doi.org/10.1016/0020-7373(89)90028-X

Flatter, David (2018, May), 'Software Science' revisited: rationalizing Halstead's system using dimensionless units, *NIST Technical Note 1990,* 1-9. https://doi.org/10.6028/NIST.TN.1990

Halstead, Maurice H. (1977), Elements of Software Science, Elsevier.

ISSN: 2473-4901
v10 n6152

Fung, Brian (2024, July 24), We finally know what caused the global tech outage – and how much it cost, *CNN Business.*

Jones, Capers (2017, May 4), The Mess of Software Metrics, Version 10.0, Namcook.com.

Mccabe.com (2024, May 15). McCabe Software: The software Path Analysis Company. Retrieved from https://mccabe.com.

McCabe, Thomas J. (1976, December), A complexity measure, *IEEE Transactions on Software Engineering,* SE-2(4):308-320, https://doi.org/10.1109/TSE.1976.233837.

Navlakha, J.K. (1987), A Survey of System Complexity Metrics, *The Computer Journal*, 30(3),233-238. https://doi.org/10.1093/comjnl/30.3.233

Stroud, J.M. (1956), The fine structure of psychological time, in H. Quastler (Ed.), *Information theory in psychology: problems and methods (pp. 174-207),* Free Press.

ISSN: 2473-4901
v10 n6152

Appendix

	Halstead						McCabe	
ID	**Time (T)**	**Length (N)**	**Volume (V)**	**Difficulty (D)**	**Intelligent Content (I)**	**Effort (E)**	**v(G)**	**ev(G)**
1	8,600	585	3,573	43.32	82.49	154,803	18	1
2	7,586	509	3,190	42.81	74.52	136,541	17	1
3	5,242	475	2,959	31.89	92.77	94,361	16	1
4	3,814	413	2,565	26.77	95.79	68,659	13	1
5	5,946	466	2,817	38.00	74.12	107,034	13	1
6	7,064	518	3,256	39.05	83.37	127,149	28	1
7	9,948	611	3,770	47.50	79.36	179,067	19	1
8	9,062	633	4,046	40.31	100.37	163,118	21	1
9	5,637	442	2,770	36.63	75.62	101,461	18	1
10	9,902	591	3,681	48.42	76.03	178,233	29	1
11	5,930	454	2,819	37.87	74.45	106,746	15	1
12	7,397	521	3,183	41.84	76.07	133,147	18	1
13	4,291	427	2,626	29.41	89.28	77,238	16	1
14	6,851	503	3,083	40.00	77.08	123,321	21	18
15	8,193	529	3,372	43.73	77.12	147,471	17	8
16	6,095	462	2,803	39.15	71.59	109,716	17	14
17	12,893	731	4,673	49.67	94.08	232,082	22	7
18	5,510	434	2,660	37.29	71.34	99,184	18	14
19	6,955	520	3,165	39.55	80.04	125,198	20	13
20	9,256	563	3,637	45.81	79.38	166,601	24	6
21	4,099	379	2,257	32.70	69.02	73,787	14	13
22	4,916	450	2,794	31.67	88.24	88,485	17	13
23	5,797	434	2,817	37.03	76.08	104,337	20	18
24	4,421	436	2,732	29.12	93.82	79,573	17	13
25	5,938	466	2,847	37.55	75.81	106,892	26	18
26	5,180	418	2,657	35.09	75.74	93,244	18	11
27	7,073	513	3,234	39.37	82.14	127,311	28	20
28	4,214	402	2,384	31.81	74.94	75,849	16	13

ISSN: 2473-4901
v10 n6152

29	6,412	492	3,055	37.78	80.87	115,413	22	18
30	6,169	480	2,990	37.14	80.50	111,039	26	18

Table 6: ICE 04 Student Data