Tool Assisted Identifier Naming for Improved Software Readability: An Empirical Study

Phillip A. Relf
Faculty of Engineering
University of Technology, Sydney
Sydney, NSW, Australia
Phillip.A.Relf@uts.edu.au

Abstract

This paper describes an empirical study investigating whether programmers improve the readability of their source code if they have support from a source code editor that offers dynamic feedback on their identifier naming practices. An experiment, employing both students and professional software engineers, and requiring the maintenance and production of software, demonstrated a statistically significant improvement in source code readability over that of the control.

1. Introduction

Programmers find the concurrent writing of source code and the selection of meaningful identifier names difficult. The practice of allowing software to enter production, that has employed poor identifier naming practices, increases the cost of future maintenance activities over the costs if poor identifier names had been removed during coding. This paper reports the effects on the source code of dynamically reporting identifier-naming style flaws during editing. The investigation also considers whether or not the reduction in these identifier-naming style flaws then results in improved source code readability as indicated by the use of meaningful identifier names.

1.1. Paper Organisation

Section 1 introduces the empirical study, describes the main objectives of the study and provides a literature review in support of the study. Section 2 describes the experiment design and describes the research variables. Section 3 describes the design of a source code editor from the perspective required to support the experiment and describes the identifier-naming style guidelines implemented by the source code editor. Section 4 describes the investigation conducted and presents the results collected from the experiment. Section 5 summarises the results collected and presents the conclusions drawn from these results. Section 6 and 7 acknowledges those persons who have been directly or indirectly involved in supporting this research.

1.2. Literature Review

The programming culture that has grown out of programming in the small (i.e., 100k SLOC and smaller) has created software maintenance issues when these practices are applied to the development of large computer systems (i.e., 1M SLOC and larger). A single programmer may be consistent with their identifier naming and may consistently choose the same identifier-naming style in the development of their software. However this may not necessarily be sufficient for programming in the large. By way of an example, a programmer may choose to name a constant identifier, which represents some maximum value, with one of say the following possible identifier-naming styles: *mValue*, *valueM*, *maxValue*, *valueMax*, *maximumValue*, *valueMaximum*, *m*, *max*, *maximum*, *mxmm* etc. Any one of these identifier names could potentially be considered as acceptable practice when used within small systems but when multiple programmers choose their own different identifier-naming styles the cognitive effort required of a maintenance programmer increases with consequent deterioration in the results of the maintenance task. Software maintenance is an enduring problem which has existed for over 40

0-7803-9508-5/05/$20.00 (c)2005 IEEE

years and which will most likely rise further in importance over the next decade [1].

Currently, on average, over 70% of the total software lifecycle cost of a software system is allocated to the maintenance phase [2]. Software maintenance is becoming of critical interest to customers as any software that is useful typically stimulates user-generated requests for capability enhancement [1]. However many of these user-generated requests would not have been apparent to the original software developers and hence the realisation of these maintenance requests requires that the software be maintainable [1].

Software maintainability is contingent on program comprehension and hence program readability [8]. However there is substantial evidence that programmers find the concurrent specification of source code and the derivation of identifier names that are generally understandable by other programmers to be a difficult task. These difficulties result from cognitive limitations [3, 4, 5, 19, 22]; education issues [6, 9, 11, 14, 16, 17, 18, 23]; cultural practices on the part of the programmer [2, 6, 12, 19, 25]; and cultural practices on the part of the programmer's management [7, 13, 19, 21]. These same researchers suggest that programmer characteristics such as the number of years programming experience will influence the degree of difficulty experienced by the programmer in devising identifier names. This may confound the collection of data in any experimental evaluation of this phenomenon. Investigation of this programmer characteristic is addressed further in section 4.5.

2. Experimental Method

Based on the above discussion, the following hypothesis (H1) is proposed:

> *Providing <u>heuristic-based dynamic reporting</u> of identifier-naming style flaws, as identified by an expert as affecting source code readability, during source code editing <u>can reduce the occurrence of these flaws</u>.*

An experiment consisting of three programming exercises was defined i.e.: an exercise which introduces the test subjects to the capability of the source code editor and requests the test subjects to replace poor identifier names with more meaningful identifier names; a maintenance exercise which requests the test subjects to make simple modification to an existing software unit that has been purposefully written with poor identifier-naming; and a production exercise which requests the test subjects to write a simple software method. Maintenance and production exercises were chosen as they are representative of both main phases of the software lifecycle.

2.1. Experiment Design

Weinberg [23] possibly exaggerates that 99% of programming studies have been conducted on individual programmers. However, the average programmer only spends one-third of their time working alone [23]. Hence, the test subjects were asked to form groups of one, two or three members, at their own prerogative, with one person 'driving' the keyboard and mouse. The test subjects were encouraged to communicate both within their groups and between groups.

Weissman [24] discovered that allowing test subjects to work at their own pace could result in inadequate data collection when multiple tasks were required of the test subjects as the students' time management skills were not adequate to managing the timely completion of each task. However, it was not considered prudent to force the students to work at a pace that they were not comfortable with, as the intent of the experiment was to identify whether dynamic reporting of identifier-naming flaws would result in modification to the source code. If the test subjects were not given adequate time to respond to the exercise then the results would not be able to distinguish between a test subject who chose to make no directed modifications to the source code files but who had completed the task with alacrity, and a slower test subject who was prematurely requested to move on to the next exercise and hence was not afforded the time necessary to modify the current source code file.

The control group differed from the experimental group in that the dynamic reporting of identifier-naming style flaws was disabled for the maintenance

and production exercise. This allowed all test subjects to start the maintenance and production exercises with the same level of awareness of specific poor identifier naming practices, as reported by the source code editor during the introduction exercise.

2.2. Introduction Exercise

The introduction exercise was defined to be a very simple exercise to demonstrate the capability of the source code editor to the test subjects. The test subjects were requested to replace three identifier names and to declare a named constant within a 12 SLOC Java source code file. The file contains a single class, a class attribute used to hold a time index, a method to increment the time index by one and to reset the time index to zero when a maximum value is set, and a literal corresponding to this maximum value. The file is also supported by a number of comments describing the intended meaning of the identifiers declared within the source code. As the logic complexity of the source code was trivial and the meanings of the declared identifiers were well commented, it was not anticipated that editing the file would present any difficulty for the test subjects other than the inherent difficulty that the test subjects would experience in devising suitable replacement identifier names.

2.3. Maintenance Exercise

The maintenance exercise used contrived source code, however the identifier-naming style practices used in its construction were typical of those seen in production software. The identifier names represented in the source code were mostly abbreviations of English words that corresponded to their actual function, with the remainder consisting of meaningful identifier names. The test subjects were requested to make the following modifications to a 118 SLOC Java source code file: (1) insert a new colour i.e., *violet* into the existing list of colours and update the "Successor" function as appropriate; (2) extend a *switch* statement to return the 4th prime number; (3) increase an array size; (4) change a secondary sort key; (5) reinstate commented-out code and correct a typographical error in a variable name; (6) remove the declaration of an unused identifier; and (7) modify a function result, which required the declaration of a Boolean flag, a test to set the flag and a test to use the flag to increment the function result by one. The test subjects' instructions contained considerably greater detail than that presented above and as part of the instructions the test subjects were given the identifier name(s) applicable to the maintenance action, so as to aid the test subject in finding the exact line(s) requiring modification. The logic complexity of the maintenance exercise source code was not expected to result in any difficulty of understanding for the test subjects and only one maintenance action was directed per source code method.

2.4. Production Exercise

The production exercise required that the test subjects code a specified method. The test subjects were given a source code file that contained one comment per required identifier declaration which described the need for the identifier, and one comment per main control statement that described the required function to be implemented. The statement comments were appropriately indented so as to show the scope of the previous statements. As the test subjects should be familiar with the required algorithm from their previous studies and they had just been asked to make a minor modification to a similar method in the maintenance exercise it was expected that the logic should not create any difficulty for the test subjects.

2.5. Research Variables

The dependent variable corresponds to the percentage of meaningful identifier names devised by the test subject(s) in each group. Any identifier name that was not ambiguous or misleading, as identified by an expert programmer, was deemed to be a meaningful identifier. The independent variable corresponded to whether or not the source code editor was dynamically reporting identifier-naming style flaws. As the dependent variable is a continuous value (i.e., 0 through 100%) any

statistical significance tests should use the Mann-Whitney test for comparing two unpaired groups.

3. Tool Design

The tools used in support of the research consisted of a source code editor which could dynamically report identifier-naming style flaws and could also assist the programmer in the replacement of the identifier name. The set of identifier-naming style guidelines are similarly considered to be a tool supporting this research and are also described in this section.

3.1. Identifier-Naming Style Guidelines

The identifier-naming style guidelines consist of a selected set of nineteen guidelines. These have been either identified from the research literature as leading to improved source code readability or have empirical evidence indicating that the guidelines can effect an improvement in source code readability. No claim is made concerning the completeness of the set. The following identifier-naming style guidelines were used during this research: Unnamed Constant, Multiple Underscore, Outside Underscore, Numeric Digit(s), Short Name, Long Name, Word Count, Identifier Encoding, Class/Type Qualification, Constant/Variable Qualification, Abstract Words, English Words, Numeric Name, Plural Word, Naming Convention, Duplicate Names, Similar Names, Unused Identifier and Same Words.

3.2. Source Code Editor

The programmer can be assisted in the task of devising meaningful identifier names by the source code editor. Such assistance can be given by the dynamic reporting of identifier-naming style flaws, as identified by a set of identifier-naming style heuristics designed for improved source code readability. Such reporting can be made non-intrusive by the highlighting of the identifier name at the point of declaration or the highlighting of a numeric literal at the point of use.

The source code editor uses a standard text editor interface (using Microsoft Rich Edit controls) incorporating File, Edit, View and Help main menu selections (see Figure 1). Deviations from the identifier-naming heuristics are highlighted to the programmer by rendering a red 'squiggly' line underneath the identifier name at the point of declaration or the numeric literal at the point of use. Figure 1 illustrates the source code editor's user interface populated with the Java source code that is the subject of the introduction exercise.

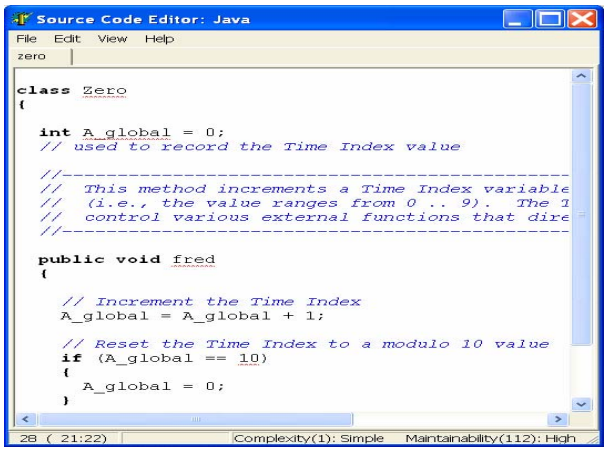

Figure 1. Source Code Editor Main Window

The status information displayed at the bottom of the window corresponds to the total number of lines in the file, the line and column number of the edit cursor, the McCabe [10] Cyclomatic complexity calculated for the file and the Pearse and Oman [15] Maintainability Index calculated for the file.

At the programmer's leisure, they may right-click on the highlighted text and be presented with a pop-up menu that allows the programmer to open a correction dialog box (see Figure 2), replace the identifier name with a suggested correction within the program scope of the current class/method, ignore the guideline for this instance only or ignore the guideline for the entire project.

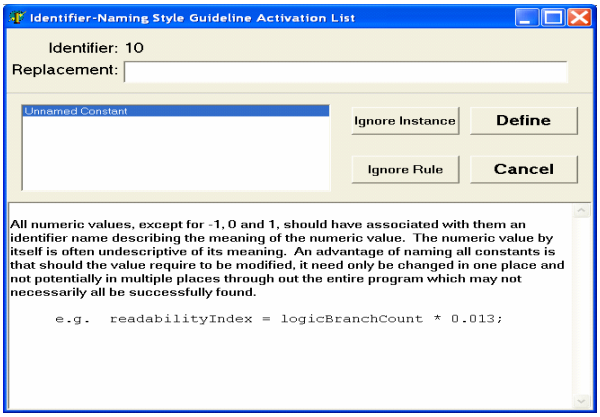

Figure 2. Identifier Name Replacement Dialogue

The correction dialog box (see Figure 2) supports the selections available from the pop-up menu and supplies additional information pertinent to the identifier-naming style flaw(s) that resulted in its launch. The original identifier name or numeric literal is displayed; a suggested replacement identifier name is presented, if appropriate; the list of relevant identifier-naming style flaw(s) presented; and a description of the currently selected identifier-naming style flaw presented. The dialog box can accept the editing of the identifier name and on user request can result in the automatic declaration of the named constant in the case of a numeric literal, and the replacement of all instances of the current numeric literal or current identifier name, as appropriate.

3.3. Strengths and Weaknesses

It was expected that the test subjects would be familiar with the concept of dynamic fault reporting and correction, as employed by the source code editor, from their successful attempts to enter spelling and grammatical errors into a Microsoft Word document. The user interface was chosen to emulate the usual look-and-feel characteristics of most contemporary Interactive Development Environments and has been limited only to the necessary editing functions so as to minimise the test subject's learning task.

The source code editor can readily execute on a standard PC and edit source code files in the order of 100 SLOC. However, as the source code file increases in size, typically so too does the combinatorial checking required on the declared identifier names. The processing requirements generally grow exponentially as the source code file increases in size and hence the source code editor will appear slow as the file size approaches 500 SLOC. Fortunately, this slow-down had little impact on the maintenance and production experiment.

4. Investigation and Results

This section describes how the test subjects were selected and how the test subjects performed during the experiment.

4.1. Test Subject Selection

The test subjects chosen were postgraduate and advanced undergraduate Software Engineering students enrolled at the University of Technology, Sydney (UTS) in the Software Systems Analysis, Software Systems Design and Software Analysis & Design courses. The experiment was conducted as a class tutorial for the students. Each class tutorial acted as a complete session with the introduction, maintenance and production exercise completed in that order and in a single sitting within the environment of a computer lab. The introduction, maintenance and production exercises were completed with a mean of 10.0 minutes, 20.9 minutes and 14.3 minutes respectively. The Software Systems Analysis and the Software System Design students were allocated to the experimental group and the Software Analysis & Design students were allocated to the control group. Six of the student groups contributed data to the control. In total 69 students, including mature age students, participated forming 35 groups distributed over three separate sessions.

In addition to the above, ten professional programmers, with two drawn from academia and eight drawn from industry, also participated in the maintenance & production experiment. Seven of the industry programmers contributed data to the control group, with the remainder contributing data to the experimental group. The professional programmes did not form groups, worked in isolation and completed the experiment in one sitting but did so in their normal office environment. The introduction,

maintenance and production exercise were completed with a mean of 8.5 minutes, 95.4 minutes and 80.0 minutes. The greater time spent with the maintenance and production files open, over that of the student group, was due to normal office distractions that the student group was isolated from in the lab environment.

4.2. Introduction Exercise

The test subjects were requested to qualify the numeric literal "10" with a named constant, and to replace the method name "fred", variable name "A_global" and the class name "zero" with more meaningful identifier names (see Figure 1). Table 1 lists the total number of test subject groups that suppled data for analysis; the number of unique identifier names devised by the test subjects; the total number of identifier names that did not result in the source code editor reporting the identifier name as an identifier-naming style flaw i.e., the identifier names that were accepted by the source code editor; and the total number of identifier names that were considered meaningful to the intent of the identifier by an expert programmer.

Table 1. Introduction Exercise Results

Identifier Kind	Test Subject Groups	Unique Id	Accepted Id	Meaning -ful Id
Constant	44	33	50%	18%
Method	43	31	70%	40%
Variable	44	27	70%	48%
Class	43	27	58%	42%

Using the source code editor to edit a file did not appear to present any difficulty to the test subject but as expected choosing a suitable identifier name resulted in considerable difficulty for some of the test subjects. One test subject suggested nine different class names before finding a name that was accepted by the source code editor. Another test subject, apparently unable to accept that the abbreviation "*idx*" was unacceptable in any part of a variable name, attempted seven different combinations before choosing to ignore the reporting on the identifier name offered. Other test subjects similarly exhibited difficulty in devising identifier names but required fewer iterations before their final selection.

The test subjects followed the recommendations for acceptability of an identifier name supplied by the source code editor for 61% (107/174) of the identifier names. However, the test subjects demonstrated some difficulty in devising meaningful identifier names in that only 38% (66/174) of the identifier names offered were meaningful to the actual use of the identifier. Given that the test subjects were constrained by the identifier-naming style heuristic implemented by the source code editor and were subject to a high level of suggestion from the imbedded comments, an impressive 68% (118/174) of the identifier names were unique for the introduction exercise. This result is surprising as the Java class, which can only be described as trivial in its simplicity, contained four separate comments containing in total six references to a "Time Index" and even with this level of prompting only 32% (14/44) of the test subjects choose to label the corresponding identifier with the name "*timeIndex*".

4.3. Maintenance Exercise

The test subjects were requested to conduct a number, of what were believed to be simple maintenance actions. Table 2 shows the percent of test subject groups, separated into student group and professional programmers that correctly responded to the maintenance activities described in section 2.3.

Table 2. Correct Maintenance Actions

Maintenance Activity	Student Group	Professional Programmer
(1)	14%	70%
(2)	94%	100%
(3)	94%	90%
(4)	49%	80%
(5)	83%	100%
(6)	43%	80%
(7)	57%	70%

Table 3 characterises the identifier names declared during the maintenance exercise and presents data in the same format as Table 1.

Table 3. Maintenance Exercise Results

Identifier Kind	Test Subject Groups	Unique Id	Accepted Id	Meaning-ful Id
Colour constant	20	3	85%	85%
Flag	32	32	53%	9%
Numeric constant	30	3	3%	3%
Store	4	4	50%	25%

The maintenance exercise demonstrated, but did not attempt to quantify, the level of difficulty that programmers experience when modifying source code written with poor identifier names. Quantification of the level of difficulty or completeness of understanding of the difficulties faced by programmers who are exposed to poor identifier names was not attempted, as this activity is outside the scope of the current research. However, the maintenance exercise did demonstrate that the inconsistent mixing of named constants with numeric literals affected the exercise in that only 29% of the test subject groups were successful in correctly completing this maintenance activity. Where as, 95% of the test subject groups were successful in correctly completing a slightly simpler maintenance activity that did not mix named constants with numeric literals. As both maintenance activities were considered very simple activities for the test subjects, the difference in complexity of the two activities is not considered overly relevant to this result.

4.4. Production Exercise

The test subjects were requested to declare a class, array, method, Boolean flag, temporary store and an array index as well as defining the size of the array in the construction of the Exchange Sort method. The test subjects were expected to generate a file of approximately 16 SLOC, but were not required to compile their resultant source code. Two test subjects copied the Exchange Sort method source code from the maintenance exercise and hence their data was removed from further consideration as the identifier name declarations were not of their own invention. Table 4 characterises the identifier names declared during the production exercise and presents data in the same format as Table 1, except that the results of the control group and experimental group are separated with the results of the control group placed first, before the results of the experimental group.

Table 4. Production Exercise Results
(control group, experimental group)

Identifier Kind	Groups	Unique Id	Accepted Id	Meaningful Id
Class	9, 16	21	11%, 19%	67%, 31%
Size	8, 19	9	13%, 5%	13%, 5%
Array	11, 21	23	18%, 71%	9%, 38%
Method	11, 17	12	55%, 76%	55%, 76%
Flag	7, 13	15	29%, 46%	57%, 77%
Store	10, 14	11	10%, 7%	10%, 7%
Index	10, 19	7	80%, 79%	0%, 16%

Not only did the test subjects continue to devise a considerable variety of identifier names but the method coded by the test subjects also demonstrated individuality and hence not all identifiers that were expected to be declared were actually declared. The most common identifier name chosen by the test subjects for the class name was “ExchangeSort”, with three instances declared. Two other class identifier names were duplicated by the test subjects with two instances each being declared i.e., “ExchangeSortClass” and “Sort”. All other class names declared were unique and included the following identifier names: “ArraySorterClass”, “at”, “Exchange”, “Exchange_Sort”, “exchangeSort”, “ExchangeSortClass”, “exchangeSortClass, “ExchangeSortInteger”, “ExhangeSortMethod” (sic), “IntArray”, “Integer”, “MainClass”, “numericSort”, “Sort”, “SortArray”, “SortClass”, “SortEngine”, “Swap”, “two” and “Xsort”. The other identifier declarations were similarly diverse.

4.5. Programmer Characteristics

The test subjects demonstrated considerable inventiveness in the construction of identifier names, with 60% of the identifier names being unique. Two identifiers in particular (i.e., a Boolean flag and a named constant which replaced a numeric literal) were given different names by every group that offered data for analysis. The test subjects had the potential to create at least seven identifier declarations during the production exercise and at least three identifiers during the maintenance exercise. Comparison between 19 groups (i.e., 11

control and 8 experimental) was possible as not all test subjects supplied data for both exercises or there was evidence that an exercise had not been completed hence making a comparison questionable because the names of the same identifier types could not be compared. The mean percentage of meaningful identifier names generated over the combined maintenance and production exercises was 25% for the control group and 48% for the experimental group, with a statistical significance of 0.0059 using a Mann-Whitney directional test. This result supports the hypothesis H1. However, this result alone does not confirm whether or not an increase in the ratio of meaningful identifier names is desirable.

The behaviour of experts is generally perceived to be desirable and the behaviour of expert programmers in their identifier naming practices in particular may be useful for novice and intermediate programmers to emulate. Simon [20] states that 50,000 chunks of knowledge distinguishes an expert and that at the rate of learning appropriate to a university student, the period required to consolidate this knowledge will be ten years. Hence the transition up to ten years of experience was of interest to the research. Figure 3 shows the percentage of meaningful identifier names suppled by the test subjects against the number of years that they have worked as a programmer. The figure shows an initial decline in the proportion of meaningful identifier names, returning to the initial level prior to the ten-year point but then improving past that of a novice as one would assume that further experience is gained.

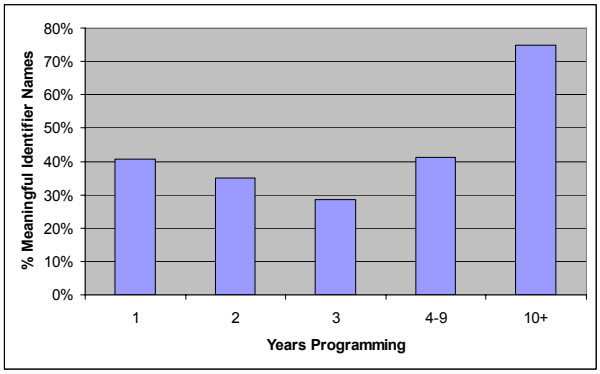

Figure 3. Years Programming vs. Meaningful Identifier Names

The results for the number of years programming, starting with their first university course, would suggest that new programmers rapidly pick up bad identifier-naming practices, during their first three years of exposure to programming and it then takes them a decade to recover to the same level as when they were new to programming. This hypothesis has however not yet been fully explored.

5. Summary and Conclusions

Test subjects consisted of university students and professional programmers. They were engaged under controlled experimental conditions to maintain existing and produce new Java source code. The experiment introduced test subjects to a source code editor that dynamically reports poor identifier naming practices based on the evaluation of arbitrarily selected identifier-naming style guidelines. The specific identifier-naming style guidelines were chosen because researchers had asserted that the guideline directs towards improved source code readability or because there is empirical evidence supporting that the guideline directs towards improved source code readability.

5.1. Programmer Support

Poor identifier naming practices can be reported during code inspections. However, this practice is expensive in terms of the expert programmer's time. Alternatively, a source code editor can be used as a quality gate that requires the programmer to correct poor identifier naming practices before a software unit can be checked in. In addition the source code editor has the following advantages over that of an expert programmer with regards to reporting poor identifier naming practices: (a) it is a cheaper resource to employ, (b) it is always available, (c) it is consistent and (d) it does not tire.

By reporting poor identifier naming practices to the test subjects, the test subjects demonstrated a propensity to modify the identifier name, to satisfy the identifier-naming style guidelines. This propensity extended to a willingness to modify their own choice of identifier name as demonstrated by the

progression of candidate identifier names entered by the test subjects until a final name was selected by the test subject.

As noted above, the dynamic reporting of identifier-naming style flaws resulted in almost a two-fold increase in meaningful identifier names being generated. This increase was shown to be statistically significant, and hence the hypothesis H1 is supported by the data collected from the population sample. This result has been achieved with a source code editor that supports nineteen identifier-naming style heuristics and it may be possible to further improve on the effect by adding further identifier-naming style heuristic. In particular the addition of identifier-naming style heuristics that are intended to result in greater consistency in the naming of identifiers during software production could further improve the readability and hence maintainability of software.

5.2. Maintenance Legacy

The increase in meaningful identifier name usage is a practice mirrored in the transition of a programmer as they obtain expert status. However, the progression along this path initially results in a reduction of meaningful identifier names from that of the neophyte prior to the programmer reaching expert status. Programmers who work in sizable teams will normally progress to a position of team leader after approximately five to seven years in industry. This results in the production of software by programmers who are at the stage of their development that results in a smaller proportion of meaningful identifier names entering the software. Left unchecked, this practice increases the maintenance issues due to identifier naming that will be inherited by future maintenance programmers. As already established, poor identifier naming can severally influence the ability to correctly modify source code, in some cases even when these modifications are trivial. This in turn results in an increased cost due to rework.

5.3. Identifier Name Standardisation

The test subjects showed considerable difficulty in devising suitable identifier names (i.e., ones that were acceptable to the source code editor) and the identifier names that were chosen infrequently coincided with the same choices made by their peers. The magnitude of this result is remarkable as in some cases there were a considerable number of constraints that should have limited the potential choice of identifier names. However the test subjects did not appear to be so limited.

The readability of source code would benefit from the use of a constrained set of English words in the construction of identifier names and a programmer would benefit from greater assistance, possibly supplied by a source code editor, in the construction of identifier names. Such a requirement may be necessary if software engineering is to adopt a standard for software interfaces and reap the benefits that standardisation can offer.

6. Acknowledgements

The author thanks Zenon Chaczko for allowing access to his students, and also to the students of UTS and the professional software engineers who participated in the experiment.

7. References

[1] Bennett, K. H.; Rajlich, V. T., *Software Maintenance and Evolution: A Roadmap*, ACM: Proceedings of the Conference on the Future of Software Engineering, pp: 75-87 & 73, May 2000

[2] Boehm, Barry; Basili, Victor, *Software Defect Reduction, Top 10 List*, IEEE: Computer Innovative Technology for Computer Professions, Vol: 34, No: 1, pp: 135-137, January 2001

[3] Brooks, Frederick P. Jr., *The Mythical Man-Month, Essays on Software Engineering, Anniversary Edition*, ISBN: 0471154059, Addison-Wesley, 1995

[4] Brooks, Ruven E., *Studying Programmer Behavior Experimentally: The Problems of Proper Methodology*, ACM, Communications of the, Vol: 23, No: 4, pp: 207-213, April 1980

[5] Detienne, Francoise, *Software Design - Cognitive Aspects*, ISBN: 1852332530, Springer, 2001

[6] Glass, Robert L., *Facts and Fallacies of Software Engineering*, ISBN: 0-321-11742-5, Addison-Wesley, 2002

[7] Haneef, Nuzhat J., *Software Documentation and Readability: A Proposed Process Improvement*, ACM SIGSOFT Software Engineering Notes, Vol: 23, No: 3, pp: 75-77, May 1998

[8] Koenemann, J.; Robertson, S., *Expert Problem Solving Strategies for Program Comprehension*, ACM: Proceedings of the SIGCHI Conference on Human Factors in Computer Systems: Reaching through Technology, pp: 125-130, 1991

[9] Laitinen, Kari; Mukari, Timo, *DNN - Disciplined Natural Naming: A Method for Systematic Name Creation in Software Development*, System Sciences. Proceedings of the Twenty-Fifth Hawaii International Conference on, Vol: ii, pp: 91-100, 1991

[10] McCabe, Thomas, J., A Complexity Measure, IEEE: Transactions on Software Engineering, Vol: 2, pp: 308-320, October, 1976

[11] Mengel, Susan A.; Yerramilli, Vinay, *A Case Study of the Static Analysis of the Quality of Novice Student Programs*, ACM: SIGCSE Bulletin, Proceedings of the Thirteenth SIGCSE Technical Symposium on Computer Science Education, Vol: 31, No: 1, pp:78-82, March 1999

[12] Nakashima, T.; Oyama, M.; Hisada, H.; Ishii, N., *Analysis of Software Bug Causes and its Prevention*, Elsevier: Information and Software Technology, Vol: 41, pp: 1059-1068, December 1999

[13] Neumann, Peter G., *Risks to the Public in Computers and Related Systems*, ACM: SIGSOFT Software Engineering Notes, Vol: 27, No: 2, pp: 5-19, March 2002

[14] Oman, Paul W.; Cook, Curtis R., *Typographic Style is More than Cosmetic*, ACM, Communications of the, Vol. 33, No. 5, pp: 506-520, May 1990

[15] Pearse, Troy; and Oman, Paul. Maintainability Measurements on Industrial Source Code Maintenance Activities, IEEE: Proceedings of the International Conference on Software Maintenance, pp: 295-303, October, 1995

[16] Poole, Bernard John. Meyer, Timothy S., *Implementing a Set of Guidelines for CS Majors in the Production of Program Code*, ACM SIGCSE Bulletin, Vol: 28, No: 2, pp: 43-48, June 1996

[17] Rising, Linda, *Removing the Emphasis on Coding in a Course on Software Engineering*, ACM: SIGCSE Bulletin, Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education, Vol: 21, No: 1, pp: 185-189, February 1989

[18] Rotenstreich, Shmuel, *The Study of Programming Standards in Computer Science Programming Courses*, IEEE Computer Standards Conference, Computer Standards Evolution: Impact and Imperatives, Proceedings of the, pp: 80-82, March 1988

[19] Sage, Andrew P.; Rouse, William B. editors, *Handbook of Systems Engineering and Management*, ISBN: 0471154059, John Wiley and Sons, 1999

[20] Simon, Herbert A.; *A Behavioral Model of Rational Choice*, The Quarterly Journal of Economics, Vol: LXIX, pp: 99-118, February 1955

[21] Slaughter, Sandra, *Assessing the Use and Effectiveness of Metrics in Information Systems*, ACM: Proceedings of the 1996 ACM IGCPR/SIGMIS Conference on Computer Personnel Research, pp: 384-391, April 1996

[22] Spinellis, Diomidis, *Reading Writing and Code*, ACM: Queue, Vol: 1, No: 7, pp: 84-89, October 2003

[23] Weinberg, Gerald M., *The Psychology of Computer Programming, Silver Anniversary Edition*, ISBN: 0-932633-42-0, Dorset House Publishing, New York, 1998

[24] Weissman, Larry, *Psychological Complexity of Computer Programs: An Experimental Methodology*, ACM SIGPLAN Notices, Vol: 9, No: 6, pp: 25-36, June 1974

[25] Yourdon, Edward, *Programmer Attitudes and Reactions Towards Programming Productivity Techniques*, ACM: Special Interest Group on Computer Personnel Research Annual Conference Proceedings of the Thirteenth Annual SIGCPR Conference, pp: 72-84, 19