Source Code Readability Improvement Using Heuristic-Based Dynamic Error Reporting During Editing

Phillip Anthony Relf

Doctor of Philosophy in Engineering

2007

CERTIFICATE OF AUTHORSHIP/ORIGINALITY

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Signature of Student

_____

Acknowledgement

I thank my supervisors David Lowe and John Leaney for their patient direction and pertinent review comments, and particularly early on for their restraint when I was struggling to understand the full implications and the nature of the research process. I owe these two gentlemen much for their gentle direction in steering my paradigm shifts that have occurred during the prior years. In addition, I thank the following people for their support during my research: Susan Cockshell for reviewing and offering advice on questionnaire layout and wording, and also for Human Factors advice relating to the instructions for the experiment test subjects; Chris Coreless for removing the Java syntax errors from my translation of the original Ada experiment source code files into the Java computer language; the 2005 UTS classes: Software Systems Analysis, Software Systems Design and Software Analysis & Design, and also to Mark Denford and Jared Berghold for their participation as experiment test subjects. I also acknowledge the participation of the many corporate programmers who responded to questionnaires and supplied data from their participation in the experiment, and in particular to Anthony Anger, Chris Coreless, Martin Fowel, Grant Hensley, Ken Lau, Patrick Lee, Sun-Young Kim and Amir Sadross for their participation in the case studies. Finally, I thank my employer for allowing me to collect and report data as part of the industrial case studies.

I thank those anonymous reviewers who supplied comment on my research papers.

I wish to again acknowledge David Lowe for supplying editorial input.

For my parents: Dorothy, Gertrude, Ron and Sid.

Table of Contents

List of Tables

List of Figures

Abstract

This research considers whether dynamically reporting poor identifier-naming practices at the time when the source code is written can improve readability and hence maintainability. Poor identifier-naming practices have little effect on the production phase of the software lifecycle. However, poor identifier-naming practices can have a substantial impact during the maintenance phase of the software lifecycle, particularly for the maintenance of large (i.e., 1M SLOC) computer programs. Of the nineteen identifier-naming style guidelines employed to support the research and used to identify poor identifier-naming practices, thirteen were found to be useful in improving source code readability. A questionnaire was employed to ascertain whether expert programmers accepted these guidelines; a textbook survey was used to identify the potential to transmit poor identifier-naming practices; a survey of contemporary source code was used to ascertain current identifier-naming practices; and a survey of dated source code was used to ascertain how identifier-naming practices have changed over an extended period of time. In addition, a controlled experiment was used to evaluate the effects of poor identifier-naming during a maintenance exercise and to evaluate the generation of poor identifier-naming during a production activity. A novice programmer case study and a programming team case study were executed to identify the longer term effects of dynamically reporting poor identifier-naming practices. The benefit of dynamically reporting poor identifier-naming practices was most pronounced for novice programmers with the percentage of meaningful identifier names increasing from 12% to 28%. The results for expert programmers were less pronounced with the percentage of meaningful identifier names correspondingly increasing from 53% to 60%. The identifier-naming style guidelines that proved to be the most useful to programmers required that identifier names should be composed of from two to four Natural language words or project accepted acronyms; should not be composed only of abstract words; should not contain plural words; and should conform to the project naming conventions.

List of Abbreviations and Acronyms

ACM	Association for Computing Machinery
API	Application Programming Interface
CATS	Computer Architecture Topography Simulator
CD	Compact Disk
COCOMO	Constructive Cost Model
df	Degrees of Freedom
DoD	Department of Defense
Exp.	Experiment
GUI	Graphical User Interface
Id	Identifier
IDE	Interactive Development Environment
IEEE	Institute of Electrical and Electronic Engineers
Lab.	Laboratory
LOC	Line of Code
N/A	Not Applicable
PC	Personal Computer
ROM	Read Only Memory
SDD	Software Design Description
SLOC	Source lines of Code
SPC	Software Process Consortium
SRS	Software Requirements Specification
UML	Unified Modelling Language
USA	United States of America
UTS	University of Technology, Sydney
WYSIWYG	What You See Is What You Get
Y2k	Year 2000

Glossary of Terms

Most technical terms in this thesis are used consistently with the definitions given in IEEE Std 610.12-1990: *IEEE Standard Glossary of Software Engineering Terminology*. Additional terms used within the thesis, but which are not given in this standard are defined below.

Mission Critical

Mission Critical, when applied to a computer system, defines that computer system as being critical to the successful completion of the mission.

SLOC

The term SLOC is potentially ambiguous when used to identify the size of a computer program, in that the count of source lines of code can mean: (1) the total number of source lines of code generated during the production of the computer program; (2) the total number of source lines of code delivered to the customer; or (3) the total number of source lines of code necessary to build the computer program, excluding any support software. The last usage of the term SLOC hence excludes any SLOC counts attributed to test harness software and test program software. This last usage of the term SLOC has been assumed and is used within the body of this thesis.

1 Introduction

This thesis presents an argument that source code readability can be improved by providing the programmer with immediate feedback when a poor identifier name is detected. Poor identifier names for the purposes of this research were recognised by the identifier name failing to satisfy any one of the arbitrarily selected identifier-naming style guidelines. The identifier-naming style guidelines were chosen from published work, where the author either claimed or had supporting empirical evidence that the identifier-naming style guideline directed towards improved source code readability. It is important to note that this research, while relevant to programming in the small is primarily intended to be useful to programming in the large (i.e., greater than 1M SLOC). The research is primarily concerned with the effect that dynamic reporting has on source code readability. To qualify the scope of the research further, the research is not a behavioural study, nor is it overly concerned with the actual choice of the particular identifier-naming style guidelines.

1.1 Research Problem

Programmers, particularly novice programmers, can generate software that is difficult to maintain due in part to their poor choices for identifier names. Resolution of this problem is of interest to industry due to the large proportion of the software lifecycle which is devoted to software maintenance; the paradigm shift within industry to consider software as a corporate asset and no longer as a corporate liability; and the urgency placed on corporations to rapidly release new versions of their software products in order to maintain market share – all of which is enhanced by having maintainable software. Given the importance that maintainable software has within industry, the disparity between the volumes of software engineering research directed to software production over that of software maintenance is astounding. This thesis considers one of the issues relevant to industry's interests in maintaining large software systems.

Typically, researchers have not had sufficient software quality training necessary to perceive the software quality issues associated with poor identifier-naming practices. Software quality itself is intangible and can only be inferred from quality attributes such as readability, maintainability, testability, etc. and software is not visualisable, which further hides software quality attributes from the untrained person. We would not tolerate haphazard layout of components on an electronic circuit board even though this layout transpires to have no effect on the functionality of the circuit board and we would argue that there are maintainability issues associated with

untidy layout. However, we tolerate the use of ambiguous, incomplete and meaningless words in identifier names. Granted, the choice of identifier name has no direct effect on the correct functionality of a computer program but a computer program is only correct until the first user desires more capability and so computer programs must also be maintainable for them to be useful as corporate assets.

Software systems of 1M SLOC and larger typically require tens or hundreds of programmers working over many years to develop the software system. When developing large software systems, in the order of 1M SLOC, the integration programmer and the maintenance programmer may need to access hundreds of identifier names when conducting their normal work tasks. The identifier names encountered may possibly be consistent in identifier-naming style within the source code generated by an individual programmer. However, without the employment of standard identifier-naming style guidelines, the likelihood of these identifiers being consistent in their identifier-naming style within the source code across the entire project will be remote.

Identifiers can be considered from at least three perspectives:

1. From the perspective of the **compiler**, any unique and syntactically correct character string can be used as an identifier name. Identifier names such as: A001, A002, A003 etc. would be acceptable to act as data place holders for say the number of apples, apricots and avocadoes respectively that can be packed in an appropriate shipping box. However, this identifier-naming style offers cognitive difficulties for the production programmer in the development of software for all but the most trivial of computer programs.
2. From the perspective of the **production programmer**, any unique and syntactically correct character string, that has some initial meaning for the production programmer, can be used as an identifier name. Identifier names such as: Max, Max_Apl, Apple_M etc. may be acceptable to act as data place holders for the number of apples that can be packed in an appropriate shipping box. However, this identifier-naming style offers cognitive difficulties for the integration and maintenance programmer in the execution of their specific tasks during the production of large software systems when the identifier-naming style differs between related identifiers. This is because the integration and the maintenance programmers must learn the idiosyncrasies of each production programmer's source code before system integration and maintenance can effectively proceed.
3. From the perspective of the **integration and maintenance programmer** (who will be faced with potentially hundreds of identifier names during the execution of their

specific tasks) identifier-naming style that would be perceived as adequate for the production programmer can result in cognitive difficulties for the integration and maintenance programmers. From the perspective of the integration and maintenance programmer, identifier names such as: Apple_Box_Maximum, Apricot_Box_Maximum, Avocado_Box_Maximum etc. would be acceptable to act as data place holders for the number of specific fruits that can be packed in appropriate shipping boxes. Knowing that all identifier names of this type follow a standard nomenclature, the programmer only needs to remember one simple rule to be able to near-immediately and correctly synthesise the relevant identifier name for many different fruits which would not be possible without the ability to formulate such a rule. However, this identifier-naming style offers cognitive difficulties for the production programmer in the execution of their specific tasks during the production of software systems of any size.

The nature of these cognitive difficulties that are apparent for the programmer will be identified and discussed in section 2.3.2.

The choice of identifier name is not overly important in small programs but becomes increasingly important in the construction of large programs, particularly where numerous programmers will integrate software modules over extended periods - often over a decade or more. Just as different construction techniques are required to build a mud hut (a non-trivial task that can be accomplished by a single generalist resource) from those required for the construction of a sky-scraper (also a non-trivial task but requiring teams of specialised resources), different construction techniques are required to build a small computer program from that of a large software system. Here the ability to readily understand the intended meaning of identifiers created by tens and hundreds of different programmers becomes critical for the programmer engaged in the software integration and maintenance of large mission critical software systems.

Identifier-naming research is sparse, which is surprising as identifiers are fundamental to programming; exist at the atomic level within the source code and are the most frequently used tokens within source code. In particular, we have little empirical data that describes what characteristics should be possessed by identifier names in order to aid source code readability. What we do know about identifier naming is that we are poor at devising meaningful identifier names: due to cognitive limitations on the part of the programmer, due to the programmer's culture, the programmer's manager's culture and also due to existing teaching practices. The ability to create universally meaningful identifier names is also effectively damaged by the programmer's own creativity. Our creative processes will result in a plethora of different

identifier names available for choice by the programmer. Once an identifier name has been chosen, the probability that it will correspond to the same name chosen by another programmer is remote, hence making the reading of another programmer's source code an exercise in internal translation requiring cognitive effort that would be best used for programming activities.

1.2 Research Overview

This thesis introduces the research problem, lists the research questions and describes the research method. The research method is described by the research hypothesis, a literature review, the necessary research tools are specified, the surveys that were required are described, an experiment specification is made and case studies are characterised. Execution of the research method resulted in the collection of data which was analysed and conclusions were subsequently drawn. The research also considers the ethics of supplying an identifier-naming style flaw reporting capability to a programmer.

The literature review uses both ad hoc and a systematic review process to find relevant research that is used to define the background theory. In the context of this background theory, the hypothesis has been decomposed into a number of research questions. Some of these research questions can be addressed either fully or partially by the background theory. The remaining research questions that could not be fully addressed are investigated further as part of the research. The research method required the development of a questionnaire, the execution of the maintenance and production experiment, and the employment of industry-based case studies using a novice programmer and experienced programmers. The questionnaire is intended to measure the attitude of programmers to specific identifier-naming style guidelines. The maintenance and production experiment is intended to identify the short-term effects of dynamically reporting identifier-naming style flaws. The intent of the industry-based case studies is to measure the long-term effects of dynamically reporting identifier-naming style flaws. In addition, a survey of production software was conducted to identify the pervasiveness of the identifier-naming style flaws to quantify industry tolerance to these flaws. The production software survey used recently written software to identify current practices and software written substantially in the past to identify in what way an individual has modified their identifier-naming style practices over time.

The conclusions drawn by this research are mapped back to the background theory which is re-evaluated in light of the new knowledge in order to identify what has now changed.

Conclusions that cannot be mapped back to the background theory are used to qualify as the foreground theory, which is expected to be unique to this thesis. In addition, new research which is justified by the results and conclusions of this thesis is presented for consideration.

1.3 Research Objectives

The research objectives of this thesis are primarily to discover and report new knowledge relevant to improving the software quality in large software systems by addressing source code readability. In particular the intent of the research activity was to: establish the attitudes of the software community to poor identifier naming practice; establish the extent to which production software contains related software quality defects; identify whether these defects could be removed; and to identify whether these defects would actually be removed by programmers when their presence was dynamically reported to the programmer. A positive outcome from this research would support the programming community in areas of software practice, software teaching, software theory and the qualification of software quality. The research contributes to an understanding of identifier naming and generates knowledge in this area of software engineering.

This new knowledge can be used to support the novice programmer without first requiring that they assimilate experiential knowledge which may require a decade or more to discover. The programmer could be supported with necessary explanation, describing an identifier-naming style flaw, when this knowledge is most relevant to them i.e., when they are actively editing their source code. Such a proof of concept opens the possibility to automate the delivery of other software engineering knowledge, potentially at the level of an expert. Hence saving organisations hours of expert programmer's time that would normally be consumed in code review, only to find a software issue that had previously been identified. Using dynamic reporting, the detection of programming issues can occur with the result of reducing programming effort necessary to find and potentially correct these programming issues.

1.4 Research Publications

I have also had my original work, which is relevant to this thesis, published, as follows:

- *Achieving Software Quality through Source Code Readability*, Conference Proceedings, Qualcon 2004

- *Tool Assisted Identifier Naming for Improved Software Readability: An Empirical Study*, Proceedings of the 4th International Symposium on Empirical Software Engineering, ISESE 2005

1.5 Document Structure

This thesis is supported by 'boilerplate', is composed of a number of chapters (comprising the bulk of the argument forming the thesis) and a collection of appendices used to support the argument. The text formatting and 'boilerplate' sections are mandated by the University of Technology, Sydney (UTS). The thesis structure is based on the five-chapter structure as suggested by Perry (1994) i.e.: introduction, literature review (i.e., background theory), research methodology, analysis of data (i.e., investigation and results) and conclusions. An additional chapter was found to be necessary to capture the results of the literature review which identified and specified the identifier-naming style guidelines that are integral to the research. The identifier-naming style guidelines are used by the research but of themselves they do not form part of the 'analysis of data' specified by Perry (1994). Hence, this additional chapter is placed after the research methodology chapter. In addition, Phillips and Pugh (2000, p65) were also useful in defining the thesis chapter content.

Boilerplate: This section includes the title page, which identifies the document; a certificate stating authorship and originality of the thesis content; acknowledgement of any and all assistance offered in the compilation of this document and in the research itself; a table of contents, list of figures and tables; a list of abbreviations and acronyms used in the thesis; and an abstract describing the work addressed by this thesis.

Chapter 1: The Introduction chapter defines the scope of the research, discusses the relevance of the research and identifies the accepted publication generated by this research. The chapter also describes the document structure.

Chapter 2: The Background Theory chapter surveys and evaluates the relevant contributions made by other researchers and states the relevance of this background theory within the context of the research.

Chapter 3: The Research Method chapter states the research hypothesis, and specifies the research methods relevant to testing the hypothesis. This chapter also states the nature of the research tools; the design of the research tools where applicable; and states how the research

tools were used in the collection and analysis of data as part of the research problem investigation. This chapter also discusses the ethical issues associated with the execution of the relevant research methods.

Chapter 4: The Identifier-Naming Style Guidelines Specification chapter presents the results of the literature review to find relevant identifier-naming style guidelines.

Chapter 5: The Investigation & Results chapter states how the research problem investigation unfolded, describes and analyses the raw data collected and establishes statistical significance of the data, where appropriate.

Chapter 6: The Conclusions chapter integrates the data analysis and draws conclusions appropriate to the research data. The chapter also identifies the research contribution by stating how the background theory has now changed, identifies the support for the foreground theory and suggests what further work is now appropriate.

Bibliography: Identifies each reference cited within the thesis.

Appendix A: Presents the identifier-naming guidelines and specifies how they were implemented in support of the research.

Appendix B: Presents the Identifier-Naming Style Guideline Programmer Acceptance questionnaire.

Appendix C: Presents the test subject questionnaire presented during the maintenance and production experiment.

Appendix D: Presents the instructions given to the test subjects as part of the maintenance and production experiment.

Appendix E: Presents the source code and software design description that test subjects used during the maintenance and production experiment.

Appendix F: Describes the software module high-level design of the source code editor used in support of the research.

CD-ROM: Contains an electronic copy of this document, and a copy of the source code and PC executables developed as research tools.

1.6 Chapter Summary

> *What is written without effort is in general read without pleasure.*
> Samuel Johnson

In summary:

- The research aim is to offer knowledge that can result in improving source code readability.
- The research will establish the attitudes and practice of the programmer community in relation to the identifier-naming style guidelines, and will identify whether the corresponding flaws can be removed from the source code.
- Identifier naming research is unpopular with researchers.
- Poor identifier naming practices do not effect small computer programs but have a detrimental effect on the integration and maintenance of large software systems.
- Production programmers have difficulty composing source code that contains meaningful identifier names and maintenance programmers have difficulty in maintaining source code that has not been written with meaningful identifier names.
- The thesis is argued in six chapters and includes six appendices which are used to support the argument.

Having stated the research problem, a literature review is now required to better ground the nature of the problem and to identify the background theory. In addition, further justification of the research problem is given to better establish the research context.

2 Background Theory

This chapter presents the results of the literature search as a statement of the background theory.

2.1 Software Maintenance

The software lifecycle is composed of two main activities i.e., software production and software maintenance. Software production corresponds to the initial development of a software system and Glass (2002) observes that Software maintenance consists mostly of supplying enhancements to the software system.

2.1.1 Inevitability of Software Maintenance

Software maintenance is unavoidable. Even if no new functionality is required, unforseen corrections (eg. Y2k), upgrades to the operating system or the release of a new compiler can cause what was previously 'working' software to no longer function as expected and hence software maintenance will be required. Additionally, any software that is useful typically stimulates user-generated requests for capability enhancement (Bennett and Rajlich, 2000). However, many of these user-generated requests were inconceivable to (or at least not conceived by) the original software developers and hence incorporation of these requests requires that the software be maintainable (Bennett and Rajlich, 2000). Software maintenance is an enduring problem, supported by 40 years of practical experience, which will most likely rise further in importance over the next decade (Bennett and Rajlich, 2000).

Source code that is difficult to maintain consumes additional resources, which industry cannot afford to squander if it expects to remain competitive and which the customer may similarly reject for economic reasons. As Keller (1990) notes, the fate of unreadable software is that it rapidly become too costly to maintain. Today (2006) the customer is very much concerned that their substantial investment in software be protected (through development of maintainable software) to support their position in a dynamic economic and technological environment. Hence, customers are becoming increasingly interested in the internal structure of software delivered to them as this has a flow-on effect to the cost of maintaining their software product. Customers may hence contractually impose software quality standards on their contractors. The contractor interprets these standards and typically further standards are derived such as the

project coding standard. However, no matter how prescriptive the software quality standards and the derived standards, the responsibility to deliver a defect free product ultimately rests with the programmer.

Elshoff and Marcotty (1982) surveyed General Motors and reported that 75% of the software professional's time, in the data processing office, were consumed in maintenance activities. In support of this figure, a Hewlett-Packard executive at the 1992 Software Engineering Productivity conference stated that 60 – 80% of their research and development staff were involved with maintaining 40 – 50M SLOC (Pearse and Oman, 1995). Software maintenance consumes from 40% to 80% of the total software lifecycle cost (Glass, 2002). Cordes and Brown (1991) qualify the mean cost as 60% of the total software lifecycle cost, and Boehm and Basili (2001) report a mean of 70%. The software industry now employs more people who are engaged in maintaining computer programs than are employed in producing completely new computer programs (Tilley, 1993). Hence arguably making software maintenance the most important phase of the software lifecycle (Glass, 2002).

2.1.2 Size Related Difficulties of Software Maintenance

Weissman (1974) was unable to show that meaningful identifier names had any effect on source code understandability in small computer programs. However, Haneef (1998) asserts that source code readability only becomes important for the maintenance of large software systems. As Tilley (1993) notes, there are significant differences in the ability of a programmer to understand software that is composed of 1k, 100k and 1M SLOC. Due to the enormity of information encapsulated within large software systems, maintenance programmers do not attempt full understanding and rely on the readability of the source code to conduct maintenance activities (Mohan, Gold and Layzell, 2004). In support of this statement, Shneiderman (1980, pp119-120) was able to show that meaningful identifier names were more useful to source code understandability as complexity increases. Correspondingly, Ramanujan, Scamel and Shah (2000) found that meaningful identifier names do not affect the maintainability of small computer programs but that poor identifier naming adversely effects the maintainability of larger software systems.

Software engineering theory does not define what constitutes a large software system. As Humphrey (2002) notes: "size means different things to different people." The issue of size is also confounded by researchers who qualify software system size differently, such as by the number of files, number of classes or by SLOC counts, which does not afford ready

comparisons. As the majority of researchers were found to define software system size, for large software system, in terms of SLOC counts, this measure has been adopted for this research as a basis for comparison between various researcher's pronouncements defining software system size. The definition of a large software system as used by the relevant literature depends on the production environment in which the computer program was written. Ramanujan, Scamel and Shak (2000) consider 300 SLOC to be a large computer program with reference to student capability. Boehm (1981) defines 128k SLOC to be a large software system from within his experience at TRW while the COCOMO was being developed. However, Pearse and Oman (1995) who conducted research in software maintenance used 10M and 50M SLOC as the basis for industrial sized software systems.

A search was conducted on 23Dec05 to catalogue the range of SLOC count values used by researchers in their definition of small and large software systems. The search employed the ACM Library and used the search strings: "program size", "large" and "LOC". The search found 66 'hits'. Few of these 'hits' resulted in an actual definition of software system size. In many cases it was not possible to confirm that the researcher were referencing a value in simple Lines of Code, which counts blank lines and comment lines or whether they were referencing SLOC, which does not count blank lines or comment lines. Hence, the conservative approach was taken to potentially over estimate the minimum value of SLOC for a software system to be considered to be large, by assuming that all values referenced were in SLOC. The values that identified a computer program as small were: 30 (Batory, Liu and Sarvela, 2003), 100 (Humphrey, 1995), and 15k (Dwyer et al., 2004). The values that identified a software system as large were: 1k (Shneiderman, 1980, p42), 10k (Ghan, 1971), 20k (Ryder et al., 2001), 26k (Schroeder, Clark and Saltzer, 1977), 30k (Batory, Liu and Sarvela, 2003), 100k (Rountev and Chandra, 2000), 140k (Das, 2000), 500k (Hartzman and Austin, 1993) and 1M (Tilley, 1993; and Heintze and Tardieu, 2001). In addition, the following researchers have also qualified the size of a large software system as: 40k (von Mayrhauser and Vans, 1994) and 50k (Humphrey, 1995). Figure 2-1 graphs the SLOC count values against the year that the above referenced researchers consider to correspond to the minimum size for a large software system. Note that the y-axis uses a logarithmic scale.

As none of these researchers offer a basis for the SLOC value quoted, computer program size appears to be a subjective measure. This research defines a large computer program in terms of the size appropriate to the production of software requiring multiple teams of programmers working over multiple years. This definition is appropriate as the issues relevant to this research only typically manifest during software integration and software maintenance in software

systems of this size. Hence an arbitrary SLOC count of 1M SLOC[1] is defined as the minimum size for a software system to be considered as large.

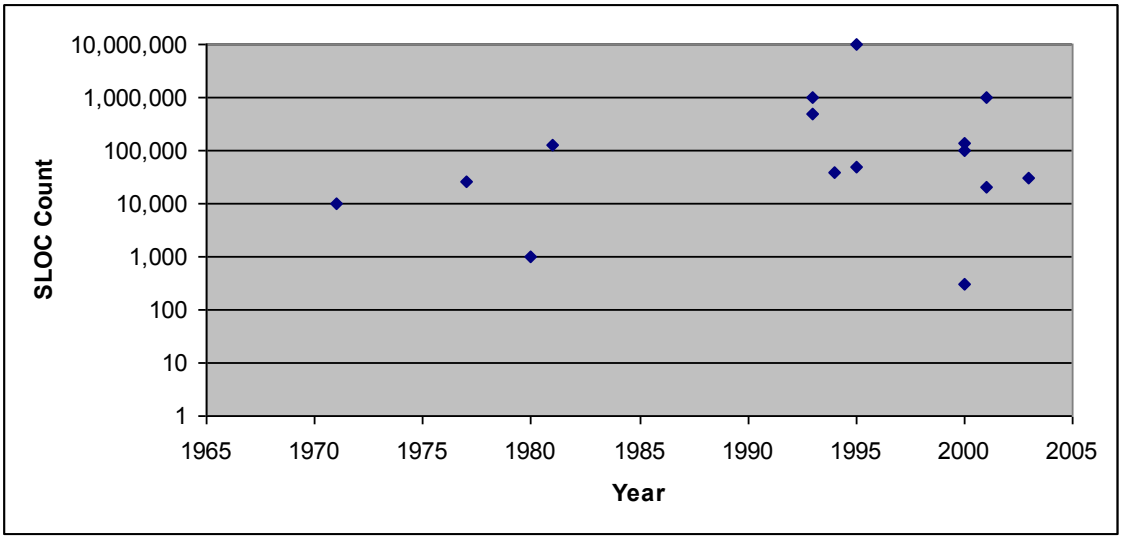

Figure 2-1 - Minimum SLOC Counts for Large Computer program

2.1.3 Software Process Scaling

Universities have been required to address problems of scale. Chemistry faculties have been forced to split into chemistry and chemical engineering departments. Similarly, computer science faculties are being transformed into computer science and software engineering departments. The difference between a pure science department and an engineering department is, at least in part, that of process scale. What can be controlled readily in a test tube does not always scale up to industrial applications. Similarly, the process of software development of small computer programs does not necessarily scale up for the efficient production of large software systems. Humphrey (2002) illustrates this problem of scaling with the following analogy:

[1] This value has been derived from the calculation of software engineer productivity in a software development project arbitrarily requiring say 100 software engineers working for a minimum of five years at 1760 hours per year and with a mean productivity of 1.1 SLOC per hour. The figure of 1760 hours per year is an industry recognised metric which is based on a forty hour week with the discrepancy in the number of hours assumed by annual leave, training and other absences. The mean productivity of 1.1 SLOC per hour is an industry metric which has been collected over many years from a software engineer base of tens of thousands of software engineers working in the development of large mission critical software systems.

If the largest program you have written is a couple of thousand SLOC then you have built yourself a mud hut. If it is a couple of hundred thousand SLOC then you have built yourself a modern 3-bedroom home. If it is a couple of million SLOC then you have built yourself a small office block. If it is a couple of tens of million SLOC then you have built yourself a sky-scraper. The techniques used in the construction of a mud hut do not scale well to the construction of a sky-scraper.

Software development processes, which do not cause cognitive overloading during the development of small computer programs, are capable of exceeding the cognitive abilities of programmers who are engaged in the construction of large software systems. In particular, programmers engaged in the software integration and software maintenance of large software systems require, amongst other software attributes, that the identifier-naming style will be consistent over the software modules. This goes beyond a need for the consistent use of character case and the presence or absence of separation characters in the formation of an identifier name. The programmer must be able to assume a greater level of identifier name standardisation than is current common practice for the development of small computer programs.

2.1.4 Source Code Readability and Software Maintenance

Fang (2001) noted that not only is the source code read by the author but also by the maintainers. The Software Productivity Consortium (SPC, 1995) observes that over the complete software lifecycle, source code will be read potentially many times but written only once. They further assert that the added cost in ensuring that source code is readable and understandable is acceptable. When programmers, working on the same software product, use different coding style conventions the lack of consistency makes the continual maintenance of the software product difficult (Vaughan-Nichols, 2003). Deibenbock and Pizka (2005) conducted experiments relating to the use of homonym identifiers (i.e., the same identifier name but used for different purposes within the source code) and found that as the number of poor identifier names increased, there was a greater tendency by the maintenance programmers to disregard software quality, and to introduce arbitrary and misleading identifier names into the source code in support of maintenance activities. Meaningless identifier names add complexity to the process of understanding the source code and consequently burdens the cognitive abilities of the programmer (Shneiderman, 1980, p72). In particular, reengineering actions will fail unless poor identifier-naming practices can be removed from legacy software (Sneed, 1996).

Elshoff and Marcotty (1982) assert that modifying source code in accordance with identifier-naming guidelines that generally direct towards more readable source code, is a worthwhile activity given the proportion of time that software remains in the maintenance phase of the software lifecycle. Further to this, Cordes and Brown (1991) asserts that the practice of rendering source code readable should be encouraged. Today (2006) programming (particularly for large software systems) is usually team based; necessitating that software must now be readable by all members of the team. Oman and Cook (1990b) observe that approximately half of a maintenance programmer's effort is spent in reading source code and that maintenance programmers identify their largest problem as understanding the intent and style of another programmer.

A 1983 survey of USA Air Force sites identified that the second biggest problem facing software maintenance, past high staff turnover, was the understanding of source code (Glass, 2002). Koenemann and Robertson (1991) assert that program comprehension and hence program readability is a central activity to program maintenance. More generally, Laitinen and Mukari (1991) assert that source code understandability depends largely on the understandability of identifier names. Siy and Votta (2001) suggest that identifiers are useful to program understandability as they act as 'beacons' that guide the programmer's understanding of the source code. Similarly, Tenny (1988) considers program readability to be an important aspect of maintenance and defines readability within the context of program maintenance. Understandably, if the source code readability presents difficulty to the maintenance programmer, the maintenance function will similarly suffer as source code must first be read and understood before maintenance is possible (Pearse and Oman, 1995).

Spinellis (2003) asserts that the single factor responsible for source code readability is programming style; such as program structure, indentation, comments, identifier-naming style etc. The style of writing text affects the readability and understandability of the text just as the programming style effects readability and understandability of the source code (SPC, 1995). Researchers investigating program understanding hypothesise that a meaningful identifier name acts as a semantic index to a knowledge structure or schema. For example, Detienne (2001) suggests that the "counter schema" would be activated in an expert programmer on encountering an identifier name such as "count" and that the expert programmer would be able to infer a missing initialisation statement i.e., "count := 0;", more easily than a novice. Hence, the choice of a meaningful identifier name can assist the expert programmer during source code debugging. Laitinen (1996) cites work conducted form 1974 through to 1994 which found that

mnemonic identifier names improves understandability of the source code but that the results have not been statistically strong.

2.1.5 Economic Case for Software Maintenainability

Software maintainability requires that software modification be possible in order to correct defects, improve performance and adapt to a changed environment (IEEE Std 610.12-1990). A software defect (including incorrect functionality or inadequate performance) can result in the computer system failing to successfully complete the mission that the computer system was designed to support. However an inability to adequately modify the software to remove the software defect in a timely manner or an inability to increase performance to an acceptable level may further exacerbate the problem with potential economic loss as a consequence. Similarly, the inability of the software to be modified to adapt to a new mission can result in economic loss, as the computer system may not survive against a competitor's product.

Weinberg (1998, p247) notes that there is no relationship between the magnitude of a software defect and the consequences of the defect occurring. This observation was inspired by the loss of an $18 million rocket, by the USA in 1962 and was due to the absence of a single character (i.e., a hyphen) from a software load of 100,000 instructions. Similarly, the consequences of a computer system crashing will be different depending on the current usage of the computer system. For instance, a software crash requiring a system restart may result in the loss of a few minutes to re-boot the PC in the case of a student exercise, financial loss in the case of a banking system and unintentional loss of life in the case of a Defence platform. In the case of a student exercise, the consequences of such a software crash may be momentary embarrassment to the student who is demonstrating their software. The consequences of a batch processing banking system failing to transfer money between banks could result in the inability to transfer monies until the batch program is next run and the consequent financial costs which could range into millions of dollars. The consequence of the combat system failing on a submarine during wartime could result the loss of an asset costing over a billion dollars; the loss of in excess of fifty lives onboard the submarine; and in the reduced ability to support national defence. However, knowing that a software defect has occurred is insufficient to ensure continued economic viability of the software product. The location of the software defect must also be identified in the computer system's software and the software must also be modifiable in order to remove the software defect. Hence, software must be maintainable in order to support ongoing economic viability of the product.

Industry, particularly the Defence industry, is acutely aware of the potential cost of a mission critical computer system failing to support the mission it was designed to support. The Defence industry does not have the advantages of large numbers of users to test their computer systems as many of these computer systems are classified and hence access is necessarily restricted to a small number of users in most cases. Similarly, the Defence industry does not have the ability to repeat test scenarios under real-world conditions due to the cost associated with deploying military resources to support real-world testing (eg. deployment of surface ships comes with crewing costs of upwards of hundreds of personnel) or the ability to expend multiple ordinance under control of the computer system (eg. a torpedo can cost upwards of a million dollars). Hence software maintainability must be demonstrated prior to allowing testing under real-world conditions as there must be considerable confidence that if a software defect is found then the software is sufficiently maintainable to allow the software defect to be removed. Hence the software must be extensively maintainable in order to safeguard against potential economic loss. Industry is concerned to ensure that software maintainability is achieved and more generally that software quality is high.

2.2 Software Quality

The issues appropriate to large software systems are not widely known or acknowledged outside of the industry in which they manifest. As Lipaev (2005) notes, there are few software quality assurance textbooks that address complex software systems. However, software professionals who are not trained in quality methods will not believe that software quality is important to the production of software (Humphrey, 2002). Due to the environment that mission critical software systems support, Lipaev (2005) further states that insufficient software quality can cause greater damage than the possible benefits that the use of the software can bring.

The interest in software quality grew during the 1980's due to increased availability of competing software products and the development of software systems to address increasingly complex problems (Vollman, 1993). Not only is the complexity of software systems increasing but as Gibbs (1994) notes, the size of these software systems are increasing ten fold each decade for some industries. In order to maintain competitive and strategic advantage in a world market, software quality is a key issue which must be addressed further (Wheeler and Duggins, 1998).

2.2.1 Software Quality Practice

The development of software, like any other manufacturing process, can introduce defects into the resultant product; but unlike other manufacturing processes software is manufactured extensively once and then modified until sufficient defects have been removed such that the customer will accept the product. The issue may be that there is little incentive to improve our software quality practices. Vaughan-Nichols (2003) reports that the Standis group chair has publicly stated that many software vendors continue to make money despite selling software products that contain poor quality and bug-laden software. Chillarege (1996) noted that a decade prior, hardware and related technology had improved 10,000 fold in dependability but that over the same period, software had only marginally improved or had possibly regressed in dependability. This led Chillarege (1996) to wonder whether software engineering research was heading in the right direction. The DoD report (DoD, 2000, pp20-34) recommended a back-to-the-basics approach and in particular recommended: (1) increased software process maturity; (2) the use of expert reviews; (3) the improvement of software engineering skills; (4) the collection, dissemination and employment of best practices; (5) the adoption of commercial contracting practices; and (6) the funding of Defence specific software research.

Vaughan-Nichols (2003) reports a senior Microsoft researcher as stating that we have been developing software for 50 years, our software development tools have improved but our software quality does not reflect improvement. Last decade the record of success for software systems was that 33% were cancelled and on the average projects are 68% over schedule (Gibbs, 1994). The Standish group Chaos Study 1, published in 1999 which surveyed commercial and government software development, as cited by the Report of the Defense Science Board Task Force on Defense Software (DoD, 2000) stated that 31% of the software systems surveyed were cancelled. The DoD report (DoD, 2000, p12) concluded that the ability to deliver software systems, as specified, remains appalling in both the commercial and Defence environments.

In order to improve software quality some objective measure is required to categorise acceptable practices (Vollman, 1993) and software quality will not improve until there is a comprehensive definition available (Dromey, 1995). However, Voas (2004) observes that software quality remains subjective in nature and what is important to one individual may be unacceptable to another. Due to the non-physical nature of software, we are frustrated by our attempts to measure non-measurable software quality attributes (Voas, 2004).

2.2.2 Software Quality Categorisation

Advances in computer hardware, and problem-solving techniques; the specification of software processes; and changing programmer backgrounds and practices have modified our understanding of what constitutes software quality (Rajlich, Wilde and Buckellew, 2001). Standards that offer top-level guidance that specify software quality exist, e.g. ISO-9126:1991, which defines a set of six quality attributes i.e.: efficiency, functionality, maintainability, portability, reliability and usability (Vollman, 1993). However, these software quality standards can not prescribe how to insert quality into software as software does not exhibit quality attributes (Dromey, 1995). Instead, software shows characteristics that add to or infer quality attributes and defect attributes (Dromey, 1995).

Haneef (1998) states that software quality is affected by program structure, identifier-naming conventions, function-prototypes, comments and external documentation. Mengel and Yerramilli (1999) also investigated source code quality characteristics and found the following characteristics to be useful in communicating their understanding of software quality to their students: module length, identifier name length, comments, indentation, blank lines, line length, embedded spaces, constant definitions, use of reserved words, use of include files and the presence of GoTo statements. Shneiderman (1980, pp:66-74) found source code commenting to be beneficial to understanding for some programmers, and other programmers found comments to be distractions to understanding at best and harmful to understanding at worst. However, Laitinen (1996) demonstrated that the 'natural naming' (see Keller, 1990) of identifiers negates the need for comments in source code.

Humphrey (2002, p43) found that even experienced programmers typically insert a software defect, on average, every ten lines of source code. Half of these defects are syntax related and can be identified by a compiler (Humphrey, 2002, p46). The other half require greater effort to identify and are discovered during a source code review, testing or subsequently by the customer (Geis, 1998). Software defects can be classified as major defects or as minor defects. Gilb and Graham (1993, p75) define major defects as defects which will have a large cost associated with fixing them if they are not addressed immediately. Conversely, minor defects are typically unimportant and are considered uneconomic to address at the present time. However, Siy and Votta (2001) believe that these minor defects should be corrected in conjunction with major defect correction and that correcting these minor defects will also increases software maintainability. Boehm and Basili (2001) found that approximately 80% of software defects are found in only 20% of the software modules comprising a large software project and that about 50% of the software modules are software defect free. Boehm and Basili

(2001) suggest that identifying these software defect rich modules within a project would be useful to software engineering practice.

2.2.3 Software Quality Improvement

Current projects spend 40 – 50% of their development budgets on re-work which could be avoided using techniques such as source code inspections and testing (Boehm, 2001). Zeller (2000) concurs that the two best practices for removing software defects are testing and source code inspection. The programming community generally agrees that source code inspections detect software defects (Gilb, 1993). In addition to finding software defects, source code inspections result in more readable source code (Zeller, 2000; and Siy and Votta, 2001). However, unlike testing, when source code inspection discovers a software defect it also identifies the position in the source code of the software defect and the situation under which it will manifest (Geis, 1998). Further to this, Humphrey (1989) states that source code inspections are more efficient in finding software defects than testing as software inspections discover six – ten software defects per hour compared to two – four software defects per hour during test. Boehm and Basili (2001) assert that up to 60% of software defects can be discovered by source code inspection and that the cost of the source code inspections is less than the cost of the testing required to find these software defects. However, Siy and Votta (2001) concluded that advances in software engineering technology have reduced the value of source code inspections for the removal of software defects.

Poole, and Meyer (1996) assert that the use of appropriate software quality standards, will significantly improve the quality of software produced, by both individual and team efforts. This can be achieved by using a software quality standard to communicate current best practices and by offering a definition of common terms to aid in the communication of software concepts between individuals (Schneidewind and Fenton, 1996). In addition, McConnell (1993, p196) cites other benefits, which include: the standardisation of source code style so that the programmers do not potentially need to learn a different style applicable to each programmer who is or has worked on the project. The eccentric identifier-naming practices that are common to some programmers can be limited by the use of coding standards that prescribe identifier-naming style which hence result in more readable source code (Rotenstreich, 1988).

Coding standards are intended to reduce the complexity of developing large software systems and are intended to support the programmers in this task (Rotenstreich, 1988). Rotenstreich (1988) further observes that coding standards are required for the development of large software

systems but are of lessor importance to the development of small computer programs. Dromey (1995) observes that programmers, particularly inexperienced programmers, don't like to use coding standards and may ignore them as being irrelevant to their task because: the content and presentation is not supportive of ease of use by the programmer; and their use can be frustrated by ambiguous, contradictory or inconsistent guidelines that make the consistent application of the coding standard impossible. DeYoung and Kampen (1979) state that coding standards often contain guidelines which are based on the individual beliefs of the authors and are not generally supported by empirical evidence. Oman and Cook (1990a) further assert that theory has not been used to develop these coding standards.

Schneidewind and Fenton (1996) evaluated a number of software standards against their stated software quality criteria i.e., "a software standard is effective if, when used properly, it improves the quality of the resulting software products cost-effectively". Schneidewind and Fenton (1996) did not find a software standard that satisfied their software quality criteria. However, Schneidewind and Fenton (1996) observe that software standards may not be perfect but there is sufficient evidence to suggest that software standards do improve software quality.

The issues raised at a source code inspection have been catalogued by Porter, Siy and Votta (1995) as:

- **True Defects** – issues that address incorrect requirements implementation, design limitations and system efficiency;
- **Soft Maintenance** – issues affecting interpretation of the coding standards and program readability; and
- **False Positives** – issues that subsequently had no effect.

In an experiment to assess the cost/benefit of source code inspections in large software development projects, Porter, Siy and Votta (1995) measured the proportion of True Defects, Software Maintenance issues and False Positives as being 21%, 55% and 24% respectively. Siy and Votta (2001) later obtained corresponding results of 18%, 60% and 22% respectively. Geis (1998) obtained corresponding results of 29%, 43% and 28% respectively when an automated tool was used to analyse student source code. This last result suggests that automated tools may be useful to improving software quality, if they can duplicate the reviewing capability of an expert programmer.

2.2.4 Manual Source Code Inspection

Porter, Siy and Votta (1995) have observed that a programmer can effectively inspect a limit of 300 SLOC over a maximum time period of two hours. However, Glass (2002) suggests that the required level of attention to conduct a source code inspection can only be maintained by a programmer for one hour and that during this time only 100 SLOC will be inspected. Allocating the most effective person to participate in a source code review can also be difficult. Meyers (1988) found large variations in software defect detection capability between experienced programmers. Further to this, Land, Sauer and Jeffery (1997) conducted empirical research which indicated that experienced programmers do not perform substantially better than student programmer at detecting software defects. Irrespective of the actual number of SLOC that can be effectively reviewed, Haneef (1998) notes that source code that is difficult to read further reduces the productivity of the programmer during a source code inspection. Fisher and Cukic (2001) further notes that source code inspections are non-deterministic and can suffer when the reviewer has not been appropriately trained to conduct a source code inspection. Porter, Siy and Votta (1995) observe that software defects, discovered by an individual and presented to a group, tend to be lost when the source code review is documented. In addition to the loss of documented software defects, project resources can also be lost. The number of programmers involved in a source code inspection can threaten a schedule slip as a consequence of the schedule bottleneck occurring on a critical path (Porter, Siy and Votta, 1995). This is a significant cost which must be reduced or the effectiveness of source code inspections increase substantially before we can expect software projects to use code inspections (Fisher, 2001).

The source code review comments, instead of necessarily improving software quality, can result in degrading the programmer's morale due to the social effects of the code inspection (Glass, 2002). The negative social effects of a code inspection can be countered somewhat by using an independent team to conduct the code inspection. However, this practice results in reduced productivity resulting from an us/them mentality on the part of the two teams (Haneef, 1998). Allowing another team to modify existing source code to make it more readable, may cause the original developer to reduce the quality of their source code on the assumption that the other team will take responsibility for the production of a quality product (Haneef, 1998). Depersonalising the source code inspection can alleviate these issues.

2.2.5 Automated Source Code Inspection

Brooks (1995, pp128-137), in reference to the large separation between excellent software engineering practices and the average software engineering practice, states that an automated software tool that propagates good software engineering practices would be important to the software industry. DeYoung and Kampen (1979) state that automated software tools that reports relative source code readability would be useful to educators in the grading of student work and also to programmers in general to improve their software quality. Rees (1982) developed an automated software tool that marked student programming exercise and gave a higher score to more readable source code. Rees (1982) gave the students access to this automated software tool, and noted that identifier names consisting of a single character reduced in frequency and that the average length of identifier names increased, particularly for procedure names. Similarly, Jwiz, which was developed by Geis (1998) saved students a significant amount of time that normally would have been spent during test, by reporting software defects prior to test.

Glass (2002) asserts that defect removal from source code can benefit greatly from the use of software tools that encapsulate programming knowledge. Humphrey (2000) adds that clear and timely feedback is important to improving worker performance and in particular, Khwaja and Urban (1993) states that the immediate feedback generated by the Visual Basic software development environment is useful in reducing the effort required to correct syntax errors. Vollman (1993) found that organisations that routinely conduct source code inspections have found the development of an automated source code inspection tool to be cost-effective. Geis (1998) found that student Java programmers all inserted the same soft maintenance software defects, which would suggest that recognisable and recurring source code flaws could be effectively reported by an automated software tool.

Lint was originally developed to locate C computer language source code defects and potential inefficiencies in the source code (Johnson, 1977). Despite this promising start, little consideration has been given to the automatic detection and removal of software defects (Lieberman and Fry, 2001). Glass (2002) speculates that the reason why effective automated software tools have not been developed is for economic reasons, in that few vendors are making a profit from automated source code inspection tools. Teleman (1996) suggests that the low acceptance of automated software tools by programmers is due to a lack of consideration and understanding of the cognitive needs of programmers. Glass (2002) also speculates that the cognitive processes used during a source code inspection have not been catalogued which has resulted in few new insights into source code inspections being discovered. Brooks (1995) is

even more pessimistic and asserts that automated software tools will always be limited to discovering syntactic errors and simple semantic software defects at best.

Siy and Votta (2001) agree that some soft maintenance software defects can be identified by an automated software tool but add that source code readability remains a human judgement. Hence, the responsibility for software quality is extensively placed on the programmer and that this places an unnecessary burden on the programmer (Dromey, 1995). This burden can be alleviated somewhat by automated tools and particularly by the computer language compiler (Baker, 1997). Baker appears to be saying that denying the programmer the ability to execute source code containing detectable software quality defects, will result in a software quality improvement.

2.2.6 Identifier-Naming Style Standardisation

Deibenbock and Pizka (2005) observe, very little work has been conducted dealing with identifiers. This surprised Deibenbock and Pizka (2005) as they note that software contains 33% identifiers by symbol count and 72% of all characters composing the source code of a computer program are found in the identifier names.

During the 19th century, the manufacturing industry discovered the advantages of standardisation of parts that employed standard interfaces (Drucker, 1961). However, it was not until the beginning of the 20th century before machining techniques were precise enough to result in the actual standardisation of parts (Langlois, 2003). This discovery enabled manufacturing to migrate from a cottage industry to an efficient manufacturing industry and removed the need for maintenance activities to be conducted by artisans as unique actions. However, the enabler was not sufficient in itself, and the technical revolution could not have occurred until manufacturing progressed from a craft to a technological discipline (Drucker, 1961). Unfortunately, the software engineering industry has not fully learnt this lesson and programmers must contend with identifier naming that may be consistently applied by individual programmers but is typically inconsistent between programmers. For instance, one programmer may consistently name an identifier which is used to hold a maximum value, as any one of the following possible identifier names: M, Max, Maximum, M_Value, Value_M, Max_Value, Value_Max, Maximum_Value, Value_Maximum, etc., where the word “value” is replaced by some meaningful name. It is easy to conceive that the use of any of these identifier names would allow the parent software module to be coded, tested and debugged. However, if each programmer arbitrarily chooses an identifier-naming style the cognitive effort required to

separate these different identifier-naming styles can become excessive. Cognitive overload situations, like the 'attention resource' being exhausted, can result in the programmer experiencing difficulties during system integration and future maintenance activities. Where the programmer must contend with relatively few identifier names in the case of a small computer program, the number of identifier names in the case of a large software system can become considerable due to the need to address the software unit interfaces alone. The normal cognitive tasks required of the programmer are substantial and should not need to be further stressed by the handling of perverse identifier-naming styles.

2.3 Programmer

McConnell (2001) recounts previous claims that the need for programming had gone because Fortran has removed the need for programmers by allowing formula to be directly written. This same argument was resurrected for Visual Basic, when it was claimed that drag-and-drop of graphics components had also eliminated the need for programmers. In the early 1970's Weinberg (1998) recalls that executives desired the elimination of programmers and financed this desire with "staggering funds". Twenty-five years latter, the funds allocated remain staggering (Weinberg, 1998). The need for programmers continues and will probably remain for many years to come. At best we can improve their education, provide support which addresses their cognitive limitations and modify deviant cultural practices inherited while developing small computer programs (Weinberg, 1998).

2.3.1 Programmer's Education

Programming knowledge can be gained by education and by experience. Experiential learning is a slower process than training and requires discovery on the part of the programmer, and hence can be incomplete (Curtis, 1984). Curtis (1984) also notes that experiential learning is the most usual way of teaching programming. In support of this statement, Glass (2002) observes that in many universities the stages of the software lifecycle after coding are ignored and that the student must discover test, integration and maintenance activities by themselves. In addition, Baker (1997) notes that there is little evidence that the programming styles being taught to programming students are improvements over those taught in the past.

Rising (1989) comments that software professionals are concerned that most programming graduates have little understanding of how to develop software for large computer systems.

Rising (1989) further notes that programming graduates have been taught that maintainability, modifiability and readability may be sacrificed towards the path to produce a correct program. More than a decade later, Weinberg (1998) echoes this concern and adds that university software projects are typically not required to be maintainable, testable or even usable. Rising (1989) asserts that students should be taught that the 'ilities (eg. maintainability) are more important than correctness alone as industrial programs are at best only fleetingly correct and that they are correct only until the user wants greater or modified capability.

Laitinen and Mukari (1992) observe that the limit of instruction presented for naming identifiers in most programming texts is that names should be "clear and understandable" without giving guidance on how this can be achieved. Oman and Cook (1990a) observe that programming students may be introduced to a few simple coding style rules but that they are typically the personal preferences of their instructors. Rotenstreich (1988) claims that depriving programming students of the use of coding standards does not allow the development of programming skills necessary for large program development and that this practice does not fully prepare the student for future work in industry. Li and Prasad (2005) conducted a survey using programming students and found that the students did believe coding standards are important to programming, and the strength of this belief increases with the number of years of programming experience but that the students tend not to comply with the coding standards. This result lead Li and Prasad (2005) to infer that there may be flaws in the teaching strategies used to educate the students in software quality issues. Hiburn and Towhidnejad (2000) support this inference and add that software quality is treated as a postscript to programming during curriculum development.

Mengel and Yerramilli (1999) observe that assisting students to appreciate the relative quality of their source code is a difficult activity, which is further degraded by the limited time that staff have allocated for exercise grading. What may appear as readable to the author of the source code may be difficult to read by others (Haneef, 1998). Hence, students should be taught what characteristics of source code result in improved readability of the source code. However, currently, teaching assistants may be required to assess the source code listings of students without adequate supervision from the faculty member responsible for the course. Poole and Meyer (1996) believe this practice to be ineffective as undergraduate teaching assistants, as a rule, neither have the maturity nor the competence to give quality feedback on source code without faculty supervision. Hence, expert knowledge relevant to identifier naming should be made available to students through some other mechanism. This thesis considers one possible mechanism, which is described in the Research Method chapter in section 3.8.

2.3.2 Programmer's Cognitive Limitations

Programmers, particularly novice programmers, find the task of devising meaningful identifier names while attending to the task of writing source code exceedingly difficult. Hence the initial production of source code can result in software that does what is required when executed but is difficult and costly to maintain. Any maintenance task would require the meaning of the identifiers to be established through effort of understanding before maintenance can progress. Hence potentially the most effective time to replace poor identifier names with meaningful identifier names is immediately after the identifier has been declared and its usage established, as the understanding of the actual usage is still current with the programmer. Brooks (1995) asserts that software is unable to be visualised, hence depriving the programmer of one of our most powerful cognitive tools. During programming, the programmer must imagine how the program should behave dynamically and capture this behaviour statically in source code. This activity places a large burden on working memory (Lieberman and Fry, 2001). Spinellis (2003) observes that programmers are poor at choosing meaningful identifier names because they find it cognitively difficult to do so. Concurrent expression using programming constructs to write source code and expression using natural language to invent identifier names is taxing on the programmer. This process requires rapid context shifting between two different cognitive processes that use the same underlying apparatus with consequent degradation to working-store memory (Spinellis, 2003).

Elshoff and Marcotty (1982) note that source code can be composed to satisfy an arbitrary set of identifier-naming guidelines that generally result in the development of readable source code but may even so result in unreadable source code being produced. Even for familiar objects, the Bellcore researchers who researched in a variety of subject domains found that the likelihood of two people choosing the same name for an object was between 10% and 20% (Oliver, 1995). The same Bellcore researchers showed that naming standards do not assist in the uniform naming of an object (Oliver, 1995). What is considered to be an appropriate identifier name by one individual may be incomprehensible to another (Siy and Votta, 2001).

Sackman (1970, p47) measured ten performance variables of "highly experienced" programmers and found an order of magnitude difference in capability was typical in separating the best and worst performers. The most extreme difference (i.e., 1:28) was found for the time required to debug Cobol source code which was intended to evaluate an algebraic equation entered as a text string. Fix, Wiedenbeck and Scholtz (1993) found that less skilled maintenance programmers derive their hypotheses of source code function using information such as identifier names but that they do not first verify their assumptions. Such a practice

disadvantages the less skilled maintenance programmer when the source code contains identifier names that are not meaningful to their function. Sage and Rouse (1999) suggest that the large difference in cognitive abilities apparent between novice and expert programmers is due to their differences in the problem-solving methods that they employ, the amount of problem domain knowledge and the completeness of the schemata that they hold. Kline (1995) suggests that complex activities (such as programming) can not be addressed in the absence of schemata and that this places the novice at a cognitive disadvantage as they would not have had sufficient time to learn appropriate programming schemata. In addition, Detienne (2001) states that a novice programmer will prefer to use a bottom-up strategy that involves frequent review of prior efforts. This practice typically results in source code statements being written before proper consideration is given to naming the imbedded identifiers and this often results in poor identifier names being chosen by the novice programmer. In addition the programmer must also contend with 'namespace pollution' due to the use of a large API (Spinellis, 1998). By way of an example, the Microsoft Windows API identifier-naming style is not self consistent making the names of the constants and functions difficult for the programmer to remember (Spinellis, 1998).

2.3.3 Management's Culture

For a typical software lifecycle, low-dependability software costs the same per SLOC as high-dependability software but savings from the high-dependability software are not realised until the maintenance phase of the software lifecycle (Boehm, 2001). This is because high-dependability software costs 50% more to develop than low-dependability software; low-dependability software costs about 50% more to maintain than to develop; and high-dependability software costs about 15% less to maintain than develop (Boehm, 2001). Hence management will sacrifice source code readability, as the extra cost can only be recovered in the next production cycle, which is a future problem that can be ignored in the present (Haneef, 1998). Incentive to produce high-dependability software is low, as Spinellis (2003) notes, we will typically accept a payoff in the near-term rather than accepting a larger payoff in the more distant future, particularly when this payoff may be collected by another individual.

Slaughter (1996) reports that 80% of software quality programs fail within the first year and that the root cause is due to cultural resistance on the part of the programmers and their management. Software quality attributes such as source code readability are neglected because management sees the pursuit of readability in terms of capital expenditure which delays market product placement (Haneef, 1998). The impact that poor source code readability has on the next

production cycle is a long term problem which is easy for management to ignore in the present (Haneef, 1998). Haneef (1998) observes that most organisations do not reward their programmers to produce readable source code. Neumann (2002) further observes that not only don't programmers care that they produce a quality product but that their managers don't care either. Glass (2002) also observed that this culture is a product of successful conditioning, which will be difficult to correct.

2.3.4 Programmer's Culture

Spinellis (2003) notes that in the 1950's and 1960's programming was an individual endeavour with little regard to ensuring that software was readable. In the early years of programming, various coding tricks were employed in order to insert the compiled object code into the limited computer memory common during the time period. This culture resulted in the development of software that was so complex that it was often found to be cheaper to rewrite the entire program than to attempt modification (Weissman, 1974).

Many programmers felt that the failure of management to understand software development made developing quality software more difficult (Wilson and Hall, 1998). Glass (2002) conducted an experiment where programmers were requested to progress a task deliberately designed to be too ambitious for the time allocated to complete the task. The programmers produced 'shoddy' products that had the semblance of completeness but could not possibly work (Glass, 2002). Boehm (2001) observes that programmers behave as though the only thing of matter is the initial generation of the source code and Glass (2002) concludes that in our current culture, programmers will ignore quality in the pursuit of a working product. Nakashima et al. (1999) concluded that 50% of the design errors in a software development project that they investigated had a root cause of programmer carelessness. By adopting disciplined personal practices a reduction of 75% of the software defects, initially inserted due to undisciplined practices, can be achieved (Boehm, 2001). However, most programmers are averse to accepting change and will actively "pervert, ignore or destroy any programming standards with which he does not agree" (Yourdon, 1975). Yourdon (1975) further states that after hours spent in discussion, programmers will accept that their productivity can be improved but then argue that there is no incentive for them to do so if some form of reward is not offered.

More generally, Dumas and Parsons (1995) discovered that programmers are resistant to accepting new programming environments. However, as Barnes (2001) asserts, an automated software tool (in particular a compiler for a computer language that mandates strong typing) is

best positioned to discover certain source code flaws but that most programmers would reject this capability and that a perceived loss of control, by the programmers, may be paramount in explaining the programmer's behaviour.

Text editors such as EMACS and vi were used in the past to edit both source code and natural language documents. Today WYSIWYG editors, such as Microsoft's Word, encapsulate extensive, powerful and sophisticated natural language text processing capability. However, editors with little more capability than the primitive editors of twenty years ago remain popular with programmers for the development of source code (Vanter and Gosling, 2002). Khwaja and Urban (1993) note that language-based editors have been available since the 1970's but that they have not been accepted by programmers who incorrectly perceive the editor interfaces to be inefficient, inflexible and restrictive. Sage and Rouse (1999) observe that researchers have defined cost-effective software development methods that result in high quality software, but these methods have received "loud and clear" industry objection and are ignored by programmers because they are not perceived to be cost-effective.

2.4 Chapter Summary

> *Quality is never an accident; it is always the result of intelligent effort.*
> John Ruskin

In summary, the background theory was specified by the following:

- Software maintenance is unavoidable.
- Industry interest in software quality is increasing.
- Poor source code readability is the second biggest problem facing software maintenance, past that of high staff turnover.
- Identifier name choice including style is an important aspect of source code readability.
- Identifier name choice is very much more important during the development of large software systems than it is for the development of small software systems.
- Small computer programs are just as easy to maintain when poor identifier naming practices are employed as for when good identifier naming practices are employed. The same is not true for large computer programs.
- A qualification of software size has not been formalised and size means different things to different people.

- Programmers experience difficulty in devising meaningful identifier names due to cognitive limitations, education practices, and cultural practices on the part of themselves and their management.

The literature has provided guidance but which has not always been clear or complete. Clear guidance has been provided which describe identifier-naming style guidelines, the impact that identifier-naming style flaws can have and the need to report these flaws. The literature has not discussed the impact that reporting these identifier-naming style flaws will have or whether the flaws or suggested identifier name replacements will be accepted. The literature review has succeeded in defining the programmer's problems with devising meaningful identifier names. However, the literature has been unable to provide a quantifiable definition for software quality or specifically for source code readability. Hence it is now appropriate to state a method which can investigate poor identifier-naming practices engaged in by programmers.

3 Research Method

This chapter presents the presents the foreground theory, specifies a hypothesis that is predicted by the foreground theory and describes the research method used to test the hypothesis. The foreground theory is the theory that is new or at least that component that modifies the background theory (i.e., theory that is already known). The concepts relevant to the hypothesis are extracted and defined in order to qualify an understanding for the hypothesis scope and to enable the research method to be unambiguously defined. The methods for testing the hypothesis by experiment and also by case study are defined. The foreground theory derives from issues raised within the Introduction chapter and in response to the research questions discussed in the Background Theory chapter. However, not all of the research questions could be addressed completely using the background theory, hence further investigation is required.

This further investigation, as has previously been stated, requires that the following activities be conducted: survey of computer programming textbooks; survey of production software; execution of a maintenance and production experiment; and the execution of case studies. The research method describing these activities is detailed within this chapter. This chapter also specifies the research tools necessary to support the research. These research tools consist of a Programmer Characteristics questionnaire, an Identifier-Naming Style Guideline Programmer Acceptance questionnaire, a source code editor, and a source code identifier-naming style flaw analysis and report generator. These research tools are used variously to support the production source code survey, the maintenance and production experiment data analysis, and the analysis of the source code generated during the case studies. This chapter also identifies the metrics and variables that will be employed during the test of the foreground theory.

3.1 Foreground Theory

Programmers gain software engineering knowledge through learning which may be supported by programming experience, lectures, mentoring and by the use of textbooks. Programmers are typically writing source code before they have been taught software quality aspects that are relevant to the construction of large mission critical computer systems. It can reasonably be assumed that a poor understanding of software quality issues can lead to the development of poor identifier-naming practices that could effect the software system integration, software system test and the subsequent maintenance of these computer systems. Hence, it is postulated that if programmers are introduced to relevant software quality considerations affecting the

software readability of large mission critical computer systems, and are asked to consider improving the readability of their source code, the source code readability will improve. This theory, as stated, remains broad and qualifying it further by limiting the software quality considerations to consist of an arbitrary set of identifier-naming style guidelines that effect source code readability allows a prediction to be made in the form of a hypothesis.

3.1.1 Hypothesis Statement

The research hypothesis can be stated as:

> *Providing dynamic reporting of identifier-naming style flaws during source code editing can reduce the occurrence of these flaws, where the identifier-naming style guideline, used to detect the flaw, has been identified by an expert as affecting source code readability.*

The intent of the research described within this thesis, is to investigate and to seek verification of this hypothesis. In order to evaluate the truth of the hypothesis as stated requires decomposition into simpler concepts, which in turn require definition. These concepts that are apparent from the decomposition of the hypothesis are:

- **Dynamic reporting** – Within the context of this research, dynamic reporting refers to the near immediate identification of an identifier-naming style flaw at the point of declaration in the case of an identifier name or at the point of usage in the case of a numeric literal, to the programmer as they enter text into the source code editor.
- **Expert** – An expert by definition is a person who has gained special knowledge. Simon (1981, p107) states that 50,000 chunks of knowledge distinguishes an expert from a non-expert and that at the rate of learning appropriate to a university student, the period required to consolidate this knowledge will be ten years. However, ten years of experience gained while writing only small programs, without exposure to the issues associated with the development of large mission critical computer systems will result in a different experiential basis than a person who has had a sustained exposure to the development of large mission critical computer systems. Hence, an expert is defined, for the purpose of this research, to be a programmer who has had a minimum of ten years experience developing large mission critical computer systems.
- **Guideline** – In the most general sense, a guideline is a rule-based statement that is generally true but may not necessarily be true.

- **Identifier-naming style** – The identifier-naming style of an identifier describes the way in which an identifier name is constructed in terms of word arrangements within the identifier name, character arrangements within a word and the placement of special characters (eg. The underscore character).
- **Identifier-naming style flaw** – The identifier name itself may be flawed, in that the name may be meaningless for the task of correctly naming the actual use of the identifier. However, the intent here is that an instance of an identifier name that deviates from one or more of the accepted identifier-naming style guidelines, by definition, is flawed.
- **Source code editing** – The action, conducted by a programmer, which results in the generation or modification of source code.
- **Source code readability** – Source code readability is defined as the ease with which a programmer can read and understand source code that has been written in a computer language syntax with which the programmer is familiar. However, for the purpose of this research, source code readability is limited only to considerations regarding the readability of the identifier names used within the source code under consideration.

3.1.2 Hypothesis Verification

Crotty (1998, p25) states Wittgenstein's verification principle as: 'no statement is meaningful unless it is capable of being verified'. Crotty (1998, p25) further observes that logical positivism, which separates statements into two categories i.e., analytic statements and synthetic statements, describes how a statement can be verified. Analytic statements are closed in that what is predicted by the statement is already contained within the statement (eg. 'A doe is a female deer'). Synthetic statements are open and are verified by experience i.e., what can be either sensed directly or indirectly with the assistance of instruments, (eg. 'It's raining'). The hypothesis requires that source code readability can be affected and hence must be measured in order to evaluate the truth of the hypothesis. This defines the hypothesis as a synthetic statement requiring verification by means of experiential observation.

What we experience results in verified knowledge, which is factual (Crotty, 1998, p25). However, the discovery of facts that support a hypothesis does not prove the hypothesis as being necessarily true, no matter how many facts are found that support the hypothesis. It only takes one fact that contradicts the hypothesis to invalidate the hypothesis (Carr and Kemmis, 1986, p63). For a scientific theory to be legitimate the theory must be testable and some imaginable observation stated that would refute the theory (Carr and Kemmis, 1986, p119). The theory is

readily testable, for example conducting an experiment with programmers developing software can be used as a basis to test the theory and such an experiment is discussed in a subsequent section.

3.2 Research Questions

The research hypothesis naturally raises a number of questions, requiring a more rigorous treatment than the simple definition of terms that accompanies the research hypothesis statement. These research questions are presented, and addressed where possible, in the following subsections. The final subsection, to this section, summarises the responses that have been offered to the research questions.

3.2.1 Identifier-Naming Style and Software Quality

RQ01: Does identifier-naming style affect software quality?

Within the source code of software systems, identifier names comprise approximately 33% of all symbols and 72% by character count (Deibenbock and Pizka, 2005). Hence, the potential for identifier naming to affect software quality is significant. In support of this potential, Ghan (1971) asserts that software quality is affected by identifier names and in addition, Haneef (1998) asserts that software quality is affected by identifier naming conventions.

Siy and Votta (2001) conducted a code inspection experiment and found that 60% of all issues raised could not have been detected by subsequent testing. These issues included poor identifier naming practices and when attended to, the software quality improved (Siy and Votta, 2001). A characteristic of software quality is source code readability and Keller (1990) found that the use of meaningful identifier names in source code usually made the source code more readable and consequently improved the software quality.

3.2.2 Catalogue of Identifier-Naming Style Guidelines

RQ02: What identifier-naming style guidelines have been found?

Glass (2002) observes that many researchers recommend software engineering practices in preference to actual investigation and consequently many of these researchers' beliefs have little

basis in which to establish the relative worth of these practices as they are not based on evaluative research. An arbitrary guideline can be necessarily true, occasionally true or necessarily false. Guidelines that are necessarily true concerning the construction of identifier names, do not appear to exist, other than those rules mandated by the computer language syntax, as it is conceivable that any given arbitrary collection of characters forming an identifier name may have some significance to a programmer. An identifier-naming style guideline forbidding the construction of this particular identifier name would be invalid in at least this instance. Guidelines concerning the construction of identifier names that are necessarily false other than those rules mandated by the computer language syntax similarly do not appear to exist. Considering the identifier-naming style guideline: "An identifier name that is composed of characters that have apparent random character case (eg. ThIs_IS_an_iDENTifier) is a poor identifier name.", the guideline is not necessarily true because an identifier name that includes an acronym and hence may appear to contain random character case may be considered to be a valid identifier name under a specific project coding standard but rejected under another project coding standard. The best that can be achieved is to identify guidelines that support improved source code readability more often than not. Hence, the identifier-naming style guidelines that can be found will be heuristic in nature and therefore be likely to relate to statistical improvement rather than an absolute improvement.

Ledgard, Hueras and Nagin (1979, p30), Barnes (1989, p30), Horton (1998, p44) and Vermeulen et al. (2000, p15) suggest that identifier names should be meaningful or indicative of the data that they hold but do not suggest how this can be achieved. Conversely, Teasley (1994) states that poor identifier names are names based on the identifier's data type (e.g., Integer_Array), their function (e.g., Loop_Counter) or are arbitrary names (e.g., Wombat). Sneed (1996) identifies the following as poor identifier-naming practices: the use of girlfriend's and sportsmen's names, the use of synonyms where the same program identifiers are labelled with different identifier names, and the use of homonyms where different program identifiers are labelled with the same name. These definitions do not support the construction of guidelines that can be consistently implemented given only the definition above.

Turning to industry for suitable identifier-naming style guidelines also presents issue. Few, if any, commercial coding standards reference a researcher's work that would support the particular coding standard guideline. For instance, the Ada Programming Language, Military Standard (MIL-STD-1815) contains much advice and example source code, and has been used extensively as a coding standard by industry. However, the standard cites no reference when identifier-naming recommendations are presented. Hence, it is not generally possible to know if any one coding standard guideline is the product of a software professional's beliefs, has some

validity in that it is predicted by theory, is supported by a theoretical basis or has some empirical evidence supporting the guideline. Hence, the consideration of coding standards (commercial or otherwise) was rejected as a potential source of identifier-naming style guidelines within the scope of this research.

A stronger definition of what constitutes an identifier-naming style guideline was required. Crutchfield and Workman (1994) supply the following definition:

> "Software quality guidelines are unambiguous, language-sensitive rules or constraints relating to measurable features of a program's structure, semantics and syntax that affect its quality either positively or negatively".

It is clear that collecting a set of identifier-naming style guidelines, that direct towards improved source code readability, will be problematic in that there may not necessarily be empirical or theoretical evidence in existence that supports the guideline. Hence, the thesis will also consider guidelines that are supported by researcher's assertions that the guideline directs towards improved source code readability, in addition to those guidelines that have empirical support. The identifier-naming style guidelines used by this research were identified from peer reviewed published work. Identifier-naming style guidelines were chosen where it would be expected, as suggested by the author of the published work, that when applied to source code, the modification would result in improved source code readability. However, as noted above, there is no guarantee that the guideline thus produced will necessarily direct towards improved source code readability. The identifier-naming style guidelines are described in Chapter 4 and are further qualified in Appendix A – Identifier-Naming Style Guidelines.

3.2.3 Cognitive Complexity of Identifier-Naming Style Guidelines

RQ03: What kind of identifier-naming style guidelines can be evaluated by a programmer?

Programmers suffer from cognitive limitations which exist due to limited working memory and the degradation, over time, of the data held in working memory (see section 2.3.2). These cognitive limitations could potentially frustrate the programmer's ability to apply some identifier-naming style guidelines. If the programmer must manually apply the identifier-naming style guideline to the source code, then the cognitive effort required of the programmer may become significant to the task of improving source code readability. That is, identifier-naming style guidelines that require prohibitive cognitive effort on the part of the programmer will have little potential value in improving source code readability, if only applied manually.

Establishing which identifier-naming style guidelines will tax a programmer's cognitive abilities and under which conditions the cognitive limitations will be exceeded requires investigation.

Researchers have catalogued our cognitive limitations and a summary is presented below:

- Mauri and Williams (1982) state that we are capable of eighteen mental discriminations per second.
- Kline (1995, p34) cites the work of Simon and his co-workers that suggests that we can process about four chunks in our working memory at any instance of time with each chunk holding up to about seven bits of related information.
- Alsio and Goldstein (2000) cite research by Card, Moran and Newell (1986) that identifies the 'half-life' for one chunk only, in working memory as being about 70 seconds. This reduces to seven seconds when three chunks are placed in working memory (Card, Moran and Newell, 1986).
- Gleitman (1981, p305) cites work by Collins and Quillian (1969) measuring the time required to identify the validity of a simple statement as 1.3 seconds, 1.4 seconds and 1.5 seconds for each successive level in the knowledge hierarchy stored in long term memory. These values were measured from the times required to evaluate the validity of the following statements: "A canary can sing", "A canary can fly" and "A canary breaths".
- Our attention is limited in the same way that a flash-light can only illuminate a small area of focus at any given time. It is generally not possible to consider more than one activity at a time (unless the activities are similar) without degrading our cognitive performance (Wickens, 1992).

Hence, the programmer's cognitive limitations will generally not be exceeded if working memory does not need to concurrently hold more than four chunks of information to evaluate the identifier-naming style guideline, the identifier-naming style guideline can be evaluated within seconds and the programmer is only concerned, at that moment, with the naming of identifiers.

Table 3-1lists the categories of identifier-naming style guidelines which are used in chapter 4 – Identifier-Naming Style Guidelines Specification as a basis for the specification of the identifier-naming style guideline used in this research. The table suggests a cognitive effort against each of the identifier-naming style guideline categories. The cognitive effort corresponds to an estimate based on the mean number of characters in an identifier name and the mean number of words in an identifier name, which ever is relevant for the particular identifier-naming style guideline category. A survey conducted by the author on 5,122 SLOC

from a mission critical software product determined a mean identifier name length of 7.48 characters and a mean word count per identifier name of 1.61 words. Where as, Deibenbock and Pizka (2005) found an average identifier name length of 8.9 characters from the source code that they surveyed. As the metric values collected from the mission critical software potentially represent a least case, these values have been used in the discussion that follows. The least case is applicable to the discussion, as where the least case will cognitively overload the programmer in their attempts to apply the relevant identifier-naming style guideline, then so to will all other cases. The cognitive effort necessary to be expended by a programmer to apply a specific identifier-naming style guideline category is addressed for each of the categories in the table. The cognitive effort stated is my estimate which is justified by the accompanying discussion. However, the cognitive effort stated should only be considered to be an indicative qualification and is wanting for a more rigorous confirmation than that presented in the accompanying discussion.

Table 3-1 - Programmer Cognitive Effort

Cognitive Loading	Programmer Cognitive Effort - Discussion
Light	Single Identifier – Character Internal Relationship Evaluation of this category of guidelines requires the programmer to potentially allocate one chunk from working memory to the identifier name and one chunk to the target character. As the identifier name will also be stored in sensory memory, conditional that the programmer is currently looking at the identifier declaration, working memory can be constantly refreshed, unless destroyed by the effects of the attention cognitive resource being exhausted. As evaluation of the guideline would be expected to require 7.48 / 18 = 0.42 seconds, with two chunks from working memory consumed, the cognitive loading on the programmer is considered to be light.
Light	Single Identifier – Character Count Relationship Evaluation of this category of guidelines requires the programmer to allocate one chunk from working memory to the identifier name. As the identifier name will also be stored in sensory memory, conditional that the programmer is currently looking at the identifier declaration, working memory can be constantly refreshed, unless destroyed by the effects of the attention cognitive resource being exhausted. In addition, one chunk from working memory must be reserved to hold the character

Table 3-1 - Programmer Cognitive Effort

Cognitive Loading	Programmer Cognitive Effort - Discussion
	count. As the evaluation of the guideline would be expected to require 7.48 / 18 = 0.42 seconds, with two chunks from working memory consumed, the cognitive loading on the programmer is considered to be light.
Light	Single Identifier – Word Count Relationship Evaluation of this class of guideline requires the programmer to allocate one chunk of working memory to the identifier name. As the identifier name will also be stored in sensory memory, conditional that the programmer is currently looking at the identifier declaration, working memory can be constantly refreshed, unless destroyed by the effects of the attention cognitive resource being exhausted. In addition, one chunk from working memory must be reserved to hold the word count. As evaluation of the guideline would be expected to require 1.61 / 18 = 0.09 seconds, with two chunks from working memory consumed, the cognitive loading on the programmer is considered to be light.
Light	Single Identifier – Word Qualification Relationships Evaluation of this category of guidelines requires the programmer to potentially allocate one chunk from working memory to the identifier name and one chunk to the set of target qualification words. As the identifier name will also be stored in sensory memory, conditional that the programmer is currently looking at the identifier declaration, working memory can be constantly refreshed, unless destroyed by the effects of the attention cognitive resource being exhausted. As evaluation of the guideline requires that the target word is applied to one end of the identifier name only for identifier-naming style guidelines 08 and 09 and both ends for identifier-naming style guideline 10. However, the identifier-naming style guidelines 08, 09 and 10 will require 9, 1 and 8 separate determinations respectively, corresponding to the appropriate set of target qualification words. As evaluation of the guideline would be expected to require, in the worst case for identifier-naming style guideline 10, 2 * 8 / 18 = 0.889 seconds, with two chunks from working memory consumed, the cognitive loading on the programmer is considered to be light. See Appendix A – Identifier-Naming Style Guidelines for a description of the identifier-naming style

Table 3-1 - Programmer Cognitive Effort

Cognitive Loading	Programmer Cognitive Effort - Discussion
	guidelines.
Moderate	Single Identifier – Word Meaning Relationships Evaluation of this category of guidelines requires the programmer to allocate one chunk from working memory to the identifier name and search long-term memory to make a determination of the guideline's validity. As the identifier name will also be stored in sensory memory, conditional that the programmer is currently looking at the identifier declaration, working memory can be constantly refreshed, unless destroyed by the effects of the attention cognitive resource being exhausted. Evaluation of the guideline requires that a determination, using long term memory be made for each word composing the identifier name. Programmers would not typically associate the word "count" with the concept that "the word count is an abstract word". This concept would be separated by at least two levels in their knowledge hierarchy, hence requiring 1.5 seconds per determination of the validity of the statement to be evaluated. As evaluation of the guideline requires that a determination be applied to every word in the identifier name, until all words have been considered or a determination allows early termination an expectation of 1.61 * 1.5 = 2.42 seconds, with one chunk from working memory consumed the cognitive loading on the programmer could be considered to be light. However, the attention cognitive resource may play a role due to an inability of the programmer to make a determination rapidly. For instance, anecdotal evidence places programmers as poor spellers and poor in written communication in general. These limitations hinder some programmers and a cognitive loading of moderate has been placed on the evaluation of these guidelines.
Moderate	Single Identifier – Naming Conventions Evaluation of this category of guidelines requires the programmers to allocate one chunk from working memory to the identifier name, and one chunk from working memory may also be required to store the identifier kind and make a determination of the guideline's validity which may require multiple tests depending on the nature of the coding standard. As the identifier name will also be stored in sensory memory,

Table 3-1 - Programmer Cognitive Effort

Cognitive Loading	Programmer Cognitive Effort - Discussion
	conditional that the programmer is currently looking at the identifier declaration, working memory can be constantly refreshed, unless destroyed by the effects of the attention cognitive resource being exhausted. As evaluation of the guideline typically requires that a determination be applied to every word in the identifier name, until all words have been considered or a determination allows early termination, evaluation of the guideline would be expected to require 1.61 / 18 = 0.09 seconds, with two chunks from working memory consumed, the cognitive loading on the programmer could be considered to be light. However, the kind of identifier may need to be determined and the appropriate rule for this identifier kind selected and the attention cognitive resource may play a role due to the different cognitive processing required to make a complete determination for the guideline. Hence a cognitive loading of moderate has been arbitrarily placed on the evaluation of these guidelines.
High	Multiple Identifier Relationships Evaluation of this category of guidelines requires the programmer to allocate one chunk from working memory to each identifier name to be compared. As the programmer must fixate on each identifier name in turn, update from sensory memory will only be possible for one identifier name at a time. If the source code contains more than four identifier declarations then the guideline must be evaluated successively for smaller groupings of at most four identifier names at a time. The attention cognitive resource is expected to have a considerable role due to the need to search a source code listing for the identifier names to be compared. Hence a cognitive loading of high has been arbitrarily placed on the evaluation of these guidelines.

The identifier-naming style guidelines that result in low or medium cognitive loading may possibly allow the programming tasks to occur in parallel as the programmer maintains some spare cognitive resources. However, the programmer may have difficulty concurrently attending to a programming task while evaluating an identifier-naming style guideline that results in high cognitive loading. This is not to say that this exhausts the programmer's cognitive capacity but only that, momentarily the entire programmer's cognitive capacity is being used to the exclusion of any other task (eg. programming). This discussion assumes that

the guidelines are readily available to the programmer from long-term memory and that the programmer does not require further cognitive resources in order to represent the identifier-naming style guideline in working memory. However, if the programmer must attempt to hold more identifier names than their working memory can concurrently hold, this will impact on their cognitive ability to evaluate the guidelines. Hence identifier-naming style guidelines exist that will exceed the programmer's cognitive ability to evaluate them in at least some instances.

The above discussion assumes that only one identifier-naming style guideline is being applied at a time. When multiple identifier-naming style guidelines must be recovered from long-term memory and then applied, the programmer's cognitive abilities will become exhausted hence frustrating their ability to provide cognitive resources to the programming task. As it would not be useful to expect a programmer to evaluate identifier-naming style guidelines that exceeds their cognitive ability, a software tool may be ideally placed to automatically evaluate the identifier-naming style guideline. The source code editor used to support the research is described in section 3.8.

3.2.4 Detection of Identifier-Naming Style Flaws

RQ04: How can guideline evaluation be used to dynamically find identifier-naming style flaws in source code?

Mengel and Yerramilli (1999) assert that static analysis of source code can assist in revealing software quality, including the understandability of the source code. However, static analysis can also be allied dynamically as the source code is read from file or entered from the keyboard. This can be achieved by parsing the source code according to the computer language syntax and a token list generated. Traversing the token list and finding an identifier token forming the subject of an identifier declaration allows the identifier-naming style guideline to be applied. However, some identifier-naming style guidelines may require information concerning the names of other identifiers within the same program scope before they can be evaluated. By building a program construct tree, where each branch corresponds to a program declaration or a statement, the program construct tree can then be searched for identifiers in scope and the identifier-naming style guideline applied when an identifier token is encountered. In either case deviations from the identifier-naming style guidelines can be attached to the identifier token for subsequent reporting by the source code editor. Having previously attached the line and column number, which identifies the location of the token within the edit file; this will allow the source code editor to highlight the identifier token instance for the programmer. Hence, the

construction of a program construct tree will both allow the detection of identifier-naming style flaws by evaluation of the corresponding identifier-naming style guideline against the identifier name and the subsequent display of the flaws to the programmer. The source code editor used to support the research is described in section 3.8.

3.2.5 Automation of Identifier-Naming Style Guidelines

RQ05: What identifier-naming style guidelines can be automated?

An identifier-naming style guideline that requires understanding of the natural language meaning would require access to the defining software artefacts (eg. software requirements and software design documentation) and potentially to the application domain knowledge. The software artefacts and the application domain knowledge due to their unconstrained format may not necessarily lend themselves to automated access. Irrespective of whether automated access is possible, human intervention may be required to match an identifier declaration in the source code against the relevant name used in a software artefact or in the application domain knowledge. As the intent of the research is to identify whether dynamic reporting of identifier-naming style flaws will lead to source code readability improvement, identifier-naming style guidelines that cannot be reliably automated have been excluded from consideration. It is not believed that this will invalidate the result since the identifier-naming style guidelines previously stated should be sufficient to demonstrate an effect. However, the use of identifier-naming style guidelines that have the potential for inconsistent application could create confusion in the test subject and this confusion could conceivably result in disrupting the collection of result. Appendix A – Identifier-Naming Style Guidelines describes how the identifier-naming style guidelines can be automated requiring only the source code to be parsed in order to identify an identifier-naming style flaw against the identifier-naming style guideline.

3.2.6 Perceptions of Identifier-Naming Style Flaws

RQ06: Are the identifier-naming style guidelines universally perceived as necessarily identifying an identifier-naming style flaw?

As all programmers were once engaged in learning how to program, it could be expected that programming practices depicted in their course texts may have had some effect on the novice programmers. The effects of experience may have modified their programming practices,

depending on the actual experiences gained by the programmers. Evaluating the progression from novice to expert programmers with regard to their acceptance of the identifier-naming style guidelines necessitates that the computer languages of the era, in which the programmer gained their education, be considered. Hence a rudimentary survey of texts that teach programming may offer some incite into programmer's acceptance of the identifier-naming style guidelines, particularly considering the changing attitudes towards identifier naming over the years. The scope of this survey is to identify instances where the author's text supports or detracts from the identifier-naming style guidelines and is described further in the Textbook Survey section of this chapter.

If programmers do not exhibit an attitude of acceptance towards a specific identifier-naming style guideline then the reporting of a corresponding identifier-naming style flaw may result in the reporting, in this instance, being largely ignored by the programmer as the report may be perceived to have no effect on source code readability by the programmer. This does not mean that the programmer will necessarily fail to act to subsequently remove the identifier-naming style flaw as an alternative reason to modify the source code may be apparent. However, establishing whether specific identifier-naming style guidelines are accepted by programmers may lead to an understanding of why the reporting of specific identifier-naming style flaws are ignored by the programmer to no effect on source code readability. Hence, a questionnaire was required to identify the attitudes of programmers to the identifier-naming style guidelines. The questionnaire is described in section 3.6.

3.2.7 Display of Identifier-Naming Style Flaws

RQ07: How can identifier-naming style flaws be reported to a programmer?

Information may be communicated to a computer user by many different means. This includes but is not restricted to: sound, dialog box, display highlighting and printed reports. The current knowledge regarding auditory outputs is inadequate for the design of effective information transfer methods (Dix, Finlay, Abowd and Beale, 1998). The use of sound to alert the user to a potential error situation can be embarrassing (Sommerville, 1995) and annoying (Cox and Walker, 1993). The use of dialog boxes is not desirable for alerting the user to effectively harmless error situations as this would pre-empt the user instead of allowing them to pre-empt the computer system (Dix, Finlay, Abowd and Beale, 1998). However, display highlighting allows immediate feedback to the user without necessarily resulting in a disruption to their current task, as immediate acknowledgement of the potential error situation is not necessary.

Industry products are in existence that use this particular method to highlight a potential issue to a computer user, without undue disruption to the primary task. Microsoft Word dynamically underlines spelling errors, as they are entered by the user, with a red squiggly line. Similarly, the editor interface of the J-Builder Interactive Development Environment underlines Java computer language syntax errors. Both Graphical User Interfaces then support correction of the error via the use of a pop-up menu which offers suggested actions that the user may select from in order to correct the error reported. Hence, the reporting of identifier-naming style flaws to the programmer can also be satisfied by the use of display highlighting. The source code editor used to support the research is described in section 3.8.

There remains the concern that the specific choice of mechanism used to highlight the identifier name could affect the outcome in a manner that would differ should an alternative method of highlighting the identifier name be chosen to support the research method. Hence, the results of this research must be qualified by specifying the actual mechanism that was used to highlight an identifier-naming style flaw.

3.2.8 Benefit of Reporting Identifier-Naming Style Flaws

RQ08: Should identifier-naming style flaws be reported to a programmer?

Dromey (2003) asserts that a preventative approach to the introduction of software quality issues is preferable to that of a curative approach. Dromey (2003) supports this assertion by stating that other engineering disciplines strive to collect codify and systematically investigate failures in order to minimise the risk of repeating past mistakes in future applications. Dromey (2003) further suggests that the validity of this statement can be confirmed by consulting civil engineering handbooks on bridge construction. Boehm (1981, p39-40) also found that the cost of preventing a software quality defect was much cheaper than applying a remedial fix.

Dromey (2003) supplies a list of eight source code characteristics important to source code readability and software quality in general, with the first on this list being identifier-naming style. McConnell (1993, p211) asserts that programmers should be encouraged to put effort into generating ‘good’ identifier names as they are integral to source code readability. In support of this assertion, Siy and Votta (2001) developed a cost/benefit model in which they assume maintainability is defined as the relative cost reduction associated with modifying source code with or without minor defects removed (eg. poor identifier names). Siy and Volta (2001) argue

that the modification cost is proportional to the comprehension time required by a programmer before the source code modification can occur. Using data collected from one of their projects, Siy and Votta (2001) conclude that the difference in comprehension time of a professional programmer required for a software unit that does not contain minor defects verses that for a software unit that contains minor defects makes the repair of minor defects cost effective.".. Similarly, Haneef (1998) found that source code readability is highly related to the ability to sustain software productivity. If in addition an automated tool provides the bulk of the effort necessary to identify and subsequently correct an identifier-naming style flaw, then the modification cost is substantially reduced.

Programmers may not necessarily be aware that they are using poor identifier-naming practices. Truong, Roe and Bancroft (2004) found that introductory programming course students commonly engaged in the practice of using 'magic numbers' and defining variables that are not used. Bringing these poor identifier-naming style practices to the programmer's attention and assisting them in the removal of the poor identifier-naming style practice could result in improved source code quality. Laitinen and Mukari (1992) built a tool which replaces abbreviations in source code identifier names with natural language words. Laitinen and Mukari (1992) discovered that the activity consistently addressed approximately 1k SLOC per hour, as user interaction was required, and concluded that the staff cost was not excessive given the potential to improve software understandability for the maintenance programmer. A tool that employs a larger set of identifier-naming style guidelines, past that of only considering the presence of abbreviations within an identifier name, could show greater utility and hence may result in a potentially greater increase in source code readability than a more limited tool.

Humphrey (1995) has found that experienced programmers inject about 100 software defects per 1k SLOC. In a small program the removal of these software defects can be addressed at any time even during software maintenance with little impact to cost. However, a 1M SLOC software system will have 100,000 software defects injected by the programmer producing the software. The sheer size of this number of software defects requires that a considered approach to their removal is required (Humphrey, 1995). The cognitive resources that the programmer must expend to evaluate identifier-naming style guidelines would be substantially reduced by automation. Hence, a novice programmer need not constantly refer to the project coding standards thus consuming their attention cognitive resource, when the coding standards have been automated for their benefit. An expert programmer, who potentially has consigned the project coding standards to long term memory, need not expend cognitive resources searching long term memory for a particular project coding standard guideline. The programmer's expended cognitive effort will hence be substantially reduced by automating the identifier-

naming style guidelines. The ethical issues associated with reporting identifier-naming style flaws to a programmer are discussed in section 3.14.

3.2.9 Effect of Reporting Identifier-Naming Style Flaws

RQ09: What is the effect of reporting identifier-naming style flaws to a programmer?

A number of factors can affect software development, such as programming style, program complexity and size, and the professional competence of the programmer. However, of all these factors, programmers have the most control over programming style (Gorla, Benander and Benander, 1990). Hence, reporting programming style issues to the programmers has the potential to affect change by the programmer with greater ease than can be achieved by the reporting of other programming issues.

Theory does not support accurate prediction of how an individual programmer will respond to the reporting of identifier-naming style flaws. Hence a mechanism to record the reporting of identifier-naming style flaws must be devised and programmers are required to be placed in a controlled situation where their responses to identifier-naming style flaw reporting can be measured and recorded. Cohen, Swerdlik and Phillips (1996) report that psychology researchers argue for assessment in preference to testing when the test subject is engaged in a problem-solving activity and in particular argue that the evaluation of writing skills is best undertaken by asking the test subjects to write some text (Cohen, Swerdlik and Phillips, 1996). Similarly the investigation of a software task to affect source code readability should involve the programmer in the actual construction of software. As the software lifecycle, which describes software development, is typically sectioned into two main phases i.e., software production and software maintenance, an investigation into source code readability should consider the development of source code under both of these phases. Hence, source code maintenance exercise is required to identify the affects on the source code of reporting poor identifier names to the test subjects. Similarly, a production exercise is required to establish the effects on the source code of the dynamic reporting of identifier-naming style flaws as the test subjects enters source code into a source code editor. Hence, a source code maintenance and production experiment was required that afforded a group of programmers the opportunity to respond to the dynamic reporting of identifier-naming style flaws. Such an experiment required the existence of a source code editor that could both address the dynamic reporting of identifier-naming style flaws and also record the actions of programmers. The source code editor is discussed further in section 3.8 and the experiment design is discussed further in section 3.12.

3.2.10 Suggestion of Replacement Identifier Name

RQ10: Can a suggested action be presented to the programmer in order to correct the current identifier-naming style flaw?

The identifier-naming style guidelines defined allow for the potential, in some cases, to supply a suggested correction. There is no guarantee that a suggestion is possible in the case of a meaningless identifier (eg. “mmmmq”) or that the suggestion will result in a meaningful identifier name being generated (eg. “counts” reduced to “count” in order to convert to singular form). When a replacement identifier name suggestion is possible, the number of different identifier-naming style flaws that have been identified against the original identifier name will reduce but not necessarily reduce to zero as human intervention may still be required. Hence any suggestion for a replacement identifier name must be editable by the programmer. The specific suggestions that can be made to a particular identifier name are addressed further in Chapter 4 – Research Method.

3.2.11 Effect of Suggested Replacement Identifier Name

RQ11: Will a programmer accept identifier names that are suggested as replacements?

In order to present suggested replacements to the programmer, a software tool is required to be constructed which can function as a source code editor and also present the suggested identifier name replacements to the programmer. Khwaja and Urban (1993) suggest that novel user interfaces will require retraining that may need to counteract habits gained over many years and hence some disruption to normal productivity can be expected. Glass (2002) asserts that not only does a new programming tool lower programming productivity but it also lowers quality, at least initially. Glass (2002) suggests that any experiment that introduces a new tool must be patient when measuring outcomes.

Further investigation of this question is required by the execution of an experiment that uses multiple subjects undertaking the same task. As noted by Shepperd (1988) an issue in programming research is the trivial size of programs (i.e., 300 SLOC and less) used during experiments and the results extrapolated for larger computer systems. Also, due to the short time that the experiments were to be individually conducted, it was possible that the Hawthorne

Effect[2] could have skewed the data collected during the experiment. This data skew was expected to be in the direction of the test subject's prediction of the experimenter's interests. That is the test subjects were expected to respond in a fashion similar to the way they believed the experimenter wished them to act. As such, a case study was considered necessary to be run, were the programmers were given access to the source code editor over an extended period and their responses to an extended software production activity analysed. A software production activity allows the programmer to supply their own identifier naming and hence they would be expected to have greater investment in the identifier names than if a maintenance activity was being undertaken. Hence, a software production activity was considered appropriate as the subject of the case study. The maintenance and production experiment is discussed in section 3.12, and the case studies are discussed in section 3.13.

3.2.12 Production Software Identifier-Naming Style Flaws

RQ12: What identifier-naming style flaws exist in production software?

Identifier-naming style guidelines have been identified but the researchers, who recommend the particular guidelines make no reference to the frequency with which the corresponding identifier-naming style flaws occur within production software or occur during the initial development of software. If programmers rarely produce source code that would activate a particular identifier-naming style guideline then there may be little value in considering that particular guideline for its potential in improving source code readability. Hence a survey of production source code will be required to discover the relative frequency of the identifier-naming style flaws. This survey is described in section 3.11.

A survey of production software requires the analysis of large amounts of source code in order to remove project biases. As critics of open software and critics of closed software argue that one form of software production demonstrates superior software quality to the other (Raghunathan et. al., 2005) both forms of software will be surveyed. Due to the size of the analysis task an automatic tool was required to parse the source code and to apply the identifier-

[2] The Hawthorne Effect was discovered by Prof. Elton Mayo et al. at the Western Electric Company Hawthorne plant in Cicero, Illinois USA over the period of their research (1927 – 1932). The major finding of the research was that effectively, regardless of the experiment controls used, worker productivity appeared to improve. It was discovered that productivity improved because the workers felt empowered by the attentions given to them by the researchers and were thus motivated to please the researchers by displaying increased productivity.

naming style guidelines to each identifier declaration before generating a summary report. The identifier-naming style flaw reporting tool is described in section 3.9.

3.2.13 Measurement of Identifier-Naming Style Flaws

RQ13: How can a reduction in identifier-naming style flaws be measured?

A change in the number of identifier-naming style flaws that a given software module supports can result from editing the source code. A new declaration that introduces an identifier-naming style flaw, the deletion of a current declaration that contains a flaw or the modification of an identifier name that either introduces or removes a flaw will affect the number of flaws that a software module contains. In addition, the modification of the source code to use an identifier that has not been declared or the deletion of the last use of a particular identifier name but without the subsequent deletion of the now unused identifier declaration will also change the number of identifier-naming style flaws that a software module contains. However, a simple count of the number of identifier-naming style flaws in a software module is inadequate to measure source code readability as it can be argued that a small software module with the same number of flaws as a large software module will be less readable as the frequency of flaws is greater in the smaller software module. Similarly, the frequency say per SLOC of identifier-naming style flaws in a software module is inadequate to measure source code readability, as it can be argued that a software module with the same number of flaws as a second software module of the same size but where the first software module has fewer opportunities to introduce a flaw will be less readable than the second software module. Hence, calculation of the identified flaws as a percentage of the possible opportunities to introduce a flaw, gives a relative account of the pervasiveness of flaws. Appendix A – Identifier-Naming Style Guidelines describes, for each of the identifier-naming style guidelines in turn, how the identifier-naming style flaws were identified and how the opportunities to demonstrate a flaw were counted.

3.2.14 Source Code Readability Measurement

RQ14: How can source code readability be measured?

Deimel and Naveda (1990) state that we do not have a complete theory that can be used to measure the readability of a computer program. Hence we must consider broader software

quality attributes such as maintainability and subjective measures of source code readability. Halstead's (1977) software science metrics and McCabe's (1976) cyclomatic complexity metrics have appeared prominently in research papers that discuss software complexity, thought it is important to note that these metrics do not directly measure readability. Curtis et al. (1979) conducted experiments that empirically demonstrated a correlation between the difficulty with which maintenance programmers can understand source code and both the cyclomatic complexity and the software science metrics. A review of various researchers' work identified that correlations of 0.90 and higher have been reported between the software science metrics and software quality attributes (Curtis et al., 1979). However, Curtis et al. (1979) concede that these metrics are insufficient to predict maintenance programmer performance and that the cognitive aspect of the programming task need also to be considered.

Crutchfield and Workman (1994) state that the cyclomatic complexity metrics and software science metrics are too general in that they concentrate on lexical and syntactic features and that for software quality to be measured, structural and semantic measures must be made. Shepperd (1988) catalogues a list of criticisms that have been levied against the software science metric. These criticisms have questioned the validity of the base cognitive model, the validity of many of the empirical studies and the inconsistency with which symbol counting rules have been applied. Shepperd (1988) similarly questions the validity of the cyclomatic complexity metric and in particular states that the approach to counting program branches is simplistic and that the metric is insensitive to program constructs that are accepted as reducing the source code complexity eg. decision tables and program modularity. Shepperd (1988) also notes that attempts at metrics validation has occurred with programs that are less than 300 SLOC, which by software engineering practice, are considered trivial in size. Given the range of opinion regarding the usefulness of the cyclomatic complexity metrics and the software science metrics it is apparent that software metrics that measure specific attributes of software quality must also be considered.

Those researchers who have investigated specific software quality attributes include DeYoung and Kampen (1979) who developed a predicted readability metric, and Pearse and Oman (1995) who developed a maintainability index metric. Both the predicted readability metric and the maintainability index metric have empirical evidence supporting them. DeYoung and Kampen (1979) used multiple regression (which employs the sum of the measurable/calculated values, that are then individually normalised by an arbitrary weighting) to generate a readability predictor. The choice of the appropriate measurable/calculated values resulted from hypothesis testing which indicated the optimum readability characteristics selected from a large set of the potential characteristics considered. The resultant equation was shown by DeYoung and

Kampen (1979) to be a good predictor of source code readability. DeYoung, Kampen and Topolski (1982) report that the predicted readability metric showed a 0.64 correlation to program readability, as identified by expert programmers, with significance at the 0.001 level. Pearse and Oman (1995) state that the maintainability index metric is more stable to fluctuations in value during the maintenance activity than the individual metric values used in the construction of the maintainability index. However, Pearse and Oman (1995) warn that reliance on a single metric value alone could result in a reduced understanding of the internal changes that are due to the maintenance activity. Kearney et al. (1986) note that establishing a technique that consistently leads to a correlated change in some metric, does not necessarily support the validity of the technique. In a pathological case, the technique could favourably affect the metric but result in an actual reduction in software quality. What must be demonstrated is that a directed change in the metric ultimately results in improved software quality (Kearney et al., 1986).

It may not be possible to absolutely measure source code readability through some automated process and ultimately the engagement of an expert programmers will be required to pass judgement on whether an identifier name is meaningful for its actual use. However, the metrics identified may be useful in identifying some aspects of software quality and further investigation is warranted. The method of evaluating these metrics is further discussed in section 3.7.

3.2.15 Research Question Summary Findings

The research questions have been discussed and the relevant supporting background theory identified. In some cases the background theory was adequate to address the research question sufficiently such that further investigation is not required. However, in some cases the background theory is inadequate to fully address the research question and further investigation was required. Table 3-2 summarises the investigation discussion and status of the research questions. Where the research question has been addressed within this chapter, the Question Status is identified as “Addressed”, where the investigation is continuing, the section numbers where the research question is further addressed are listed.

Table 3-2 - Research Question Investigation Summary

Research Question Id	Investigation - Discussion	Research Question Status
RQ01	Identifier-naming style does affect software quality. For instance, differences in capitalisation schema used to construct identifier names significantly affects programmer's source code reading speed and poor identifier-naming styles can result in the potential for confusion. However, poor identifier naming style generally has no effect on the development of small computer programs; with severe economic consequences only apparent for large computer programs.	Addressed (3.2.1)
RQ02	Nineteen identifier-naming style guidelines have been identified, see chapter 4. This list cannot be considered to be complete nor can any one particular identifier-naming style guideline be considered to be necessarily completely qualified. The purpose of the literature review was to find identifier-naming style guidelines that could be used to support the dynamic reporting of identifier-naming style flaws and not necessarily to find the 'best' set of guidelines. As such, any further investigation to modify the current set of identifier-naming style guidelines or to modify individual guidelines is outside the scope of the research.	Addressed (3.2.2)
RQ03	Programmers suffer from cognitive limitations making the evaluation of some identifier-naming style guidelines under certain conditions, a difficult task for the programmer. Hence, desirably the evaluation of these identifier-naming style guidelines, under these conditions, should be relegated to automation. Investigation as to the nature of the cognitive limitations of programmers is outside the scope of this research and as such further investigation by the current research is not required.	Addressed (3.2.3)
RQ04	Dynamic reporting of poor identifier-naming instances can be supported directly by a source code editor. Investigation of an exhaustive list of reporting methods is not required as one adequate	Addressed (3.2.4)

Table 3-2 - Research Question Investigation Summary

Research Question Id	Investigation - Discussion	Research Question Status
	reporting method is sufficient to progress the research.	
RQ05	Identifier-naming style guidelines that require access to the source code as the only input and can be automated without a natural language understanding of the words composing the identifier name have been defined. The research scope does not extend to the development of an understanding of the source code that would require artificial intelligence support.	Addressed (3.2.5)
RQ06	The researchers who reported the identifier-naming style guidelines used to support this research do not similarly report whether the general programmer actually accepts these guidelines as being meaningful to them as directing towards improved source code readability. Hence, the research will endeavour to collect programmer attitudes to these identifier-naming style guidelines.	See (5.1.1) (5.1.2)
RQ07	Identifier-naming style flaws can be reported by underlining the offending identifier name at the point of declaration or the numeric literal at the point of use by a read 'squiggly' line. Investigation of an exhaustive list of methods to report an identifier-naming style flaw is not required as one adequate reporting method is sufficient to progress the research.	Addressed (3.2.7)
RQ08	Programmers will benefit from at least the partial removal of cognitive effort necessary to identify identifier-naming style flaws during coding. Similarly, integration, system test and maintenance programmers will benefit from inheriting source code that is easier to read. These benefits are more important for large programs than they are for small programs but none-the-less is also important for the education of programmers who may subsequently work on large programs. Hence, the non-disruptive reporting of identifier-naming style flaws is of benefit to all programmers and should	Addressed (3.2.8)

Table 3-2 - Research Question Investigation Summary

Research Question Id	Investigation - Discussion	Research Question Status
	consequently be reported to programmers.	
RQ09	The effect of reporting identifier-naming style flaws to a programmer is possibly unknown as the literature does not discuss this situation. Hence, the actions of test subjects will be required in order to support the evaluation of this research question.	See (5.8.2) (5.8.7) (5.10.3)
RQ10	The identifier-naming style guidelines having been previously defined, allow for the potential to supply a suggested correction to the identifier name. However, an improvement in the identifier name is possible in specific instances only.	See (3.8.3)
RQ11	Whether or not a programmer will accept a suggestion for an identifier name correction is unknown as the researchers who reported the identifier-naming style guidelines do not provide suggestion for the removal of any identifier-naming style flaw detected by their guideline. Consequently, these researchers do not identify the effects that supplying a potential correction will have on the programmer's actions. Hence, the actions of test subjects will be required in order to respond to this research question.	See (5.8.7)
RQ12	The researchers who have suggested the identifier-naming style guidelines used to support this research do not similarly report the pervasiveness of the corresponding identifier-naming style flaws in production software. Hence a survey of contemporary software is required to identify current software engineering practice and a survey of dated software is required to indicate how the artefacts of the same programmer have changed over time.	See (5.3.1) (5.4.1) (5.4.2)
RQ13	A reduction in the number of identifier-naming style flaws can be identified, and hence measured, by the reduction in the ratio of the number of actual flaws to the number of opportunities to generate	Addressed (3.2.13)

Table 3-2 - Research Question Investigation Summary

Research Question Id	Investigation - Discussion	Research Question Status
	the flaw for any specific identifier-naming style guideline.	
RQ14	Source code readability with primary reference to identifier naming can be measured to some extent using the Readability Predictor (DeYoung and Kampen, 1979) and the Maintainability Index (Pearse and Oman, 1995). However, an absolute measure of source code readability is not available using these measures and the assistance of an expert programmer will be required to subjectively measure the source code readability.	See (5.8.1)

3.3 Research Variables

The test of the hypothesis considers the effect of dynamically reporting identifier-naming style flaws on source code readability. As such, the generation of a software tool i.e., source code editor, which automates an arbitrary set of identifier-naming guidelines, will be required. Detienne (2001) argues that the methods used by experimental psychology are applicable to the evaluation of the use of software tools by programmers. As such, an independent variable is varied and a dependent variable, which is normally a performance indicator, is measured (Detienne, 2001). For the purpose of this research, the independent variable is whether or not dynamic reporting of identifier-naming style flaws has been enabled. The dependent variable will be the source code readability, as identified by an expert programmer.

Comparing results from two different test subjects remains problematic as different programmers will exhibit different programming characteristics. Various programmer characteristics have been shown by researchers to have some influence on programmer capability and potentially on their ability to produce readable software. These programmer characteristics can contribute to confounding the results of an experiment and hence are referred to as confounding variables. These potential confounding variables will be addressed below. In addition to researchers who claim specific programmer characteristics to be significant to affecting programming abilities, there are detractors too. Shneiderman (1980, p34) states that years programming, largest program written and work experience have not proven to be good

predictors of programming ability. Similarly, Pennington (1987) found no correlation in a programmer's ability to understand source code based on gender, education levels, different university courses taken or whether they had or had not taught programming, when the test subjects were otherwise equivalent.

The following subsections discuss potential confounding variables, where the literature search has identified some relationship between the programmer characteristic and source code readability. However, before the potentially confounding variables are discussed, measurement of the dependent variable is addressed.

3.3.1 Dependent Variable Measurement

The measure of source code readability does not lend itself to a quantifiable value that will result from a scalar value codification of source code readability. In order for source code to exhibit the software quality characteristic of readability, identifier names should generally be meaningful to the task in which they are employed. For an identifier name to be meaningful, the programmer should ideally be able to readily discover the meaning of the identifier using only the identifier name as a prompt. For instance, a meaningless identifier name such as "mmmmq" holds no clue as to its purpose and the programmer will need to visit the identifier declaration, read the attached comment if present and potentially search the source code to discover the identifier's usage. This may be acceptable for small programs but as the size of the program increases the task of visiting an increasing number of identifier declarations becomes tedious with subsequent loss in programmer productivity during system integration, system test and during maintenance activities. This definition continues to require the determination of an expert programmer but it has the advantage that it simplifies the evaluation of the dependent variable as only two values (i.e., 'not meaningful' and 'meaningful') need be decided for each identifier name. This result now allows the count of meaningful identifier names, to be used to compute the value of the dependent variable. Typically the dependent variable will be evaluated by recording the delta between the before and after state of the source code.

3.3.2 Professional Programmer

Most student programmers are not given the opportunity to develop skills necessary for software development of large programs during their initial training and as such it may be assumed that their ability to develop readable source code is usually less than that of a

professional programmer (see sections 2.3.1 and 2.3.3). Hence, it could be expected that programmers who have been professional programmers for a greater number of years may be expected to generate fewer identifier-naming style flaws in their source code than novice programmers.

3.3.3 Years Programming

The work conducted by Simon (1981, p108) requires an expert to be a practicing individual who has been in that field for a minimum of ten years. 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. Hence the transition up to ten years of experience was of interest to the research.

The large differences between novice and expert programmers, in particular the way they choose identifier names, suggests that expert programmers would be more likely to generate more readable source code than novice programmers (see section 2.3.2). However, Weinberg (1998) notes that the length of time working as a programmer is not necessarily indicative of an individual's programming strength. Weinberg (1998) speculates that this could be because programming experience can be diverse affording different opportunities to different individuals or that different individuals don't necessarily learn the same things from their common experience. Hence, it could be expected that programmers in general who have had a greater number of years programming may be expected to generate fewer identifier-naming style flaws in their source code but that this result may only weakly manifest.

3.3.4 Large Program Experience

Industry considers that coding standards, and hence identifier-naming style guidelines, are necessary to reduce complexity in large software projects (see section 2.2.3). However, large numbers of programming students are undisciplined in the use of coding standards (see section 2.3.1). If the programmer has not been exposed to large software projects, it is possible that they will not have gained exposure to identifier-naming style guidelines and hence may not appreciate the value in modifying their source code to follow the project identifier-naming style guidelines. As such, programmers who have had experience with large computer programmes may be more likely to generate more readable source code than those without this experience. Depending on the size of the largest program that the programmers had experience with, they

would correspondingly be expected to generate fewer identifier-naming style flaws with increased exposure to larger programs. Hence, it could be expected that programmers who have not been exposed to large software projects, compared to those who have, may generate more identifier-naming style flaws in their source code.

3.3.5 Code Review Experience – Subject

Programmers who have not had experience as the subject of a code review may ignore identifier-naming style guidelines as they see them as being a matter of personal style (see section 2.3.1) or as being irrelevant to the programming task (see section 2.2.3). They may also feel a reluctance to modify their source code as a consequence of the social disruption caused by the code review (see section 2.2.4). Typically this situation is more apparent in novice programmers. Hence, it could be expected that programmers who have only acted as the subject of a code review and have not acted as a reviewer themselves, may be more reluctant to modify source code as a consequence of a software tool reporting an identifier-naming style flaw.

3.3.6 Coder Review Experience – Reviewer

Programmers who are called upon to review source code are generally not themselves novice programmers. Intermediate and expert programmers typically have schemata which may map on to a problem solution apparent in the source code being reviewed. The programmer in this case knows what an identifier defined within this schemata is actually used for and hence may have access to a more meaningful identifier name than a novice programmer (see section 2.3.2). Hence, it could be expected that programmers who have acted as the reviewer in a code review may be more likely to modify source code as a consequence of a software tool reporting an identifier-naming style flaw.

3.3.7 Touch Typing Ability

Programmers who can touch type are able to allocate greater cognitive assets to the programming task than programmers who must use ‘hunt and peck’ to enter or modify source code. Alsio and Goldstein (2000) have shown that a skilled touch typist uses negligible cognitive resources to type on a QWERTY keyboard. However, a programmer who uses one or two fingers must assign cognitive resources to the typing task as they must alternate attention between the programming task and that of visually searching for the next required key on the

keyboard, and then moving their hand to this key. Hence, it could be expected that programmers who can touch type may also generate more meaningful identifier names as they have at their disposal a lesser depletion of cognitive resources than programmers who cannot touch type.

3.3.8 Identifier Naming Effort

Programmers are given minimal instruction during their undergraduate course in the importance of identifier-naming and their course texts are inadequate in supplying recommendation for identifier-naming (see section 2.3.1). The programmers themselves often ignore code review advice specific to identifier-naming recommendations as they see these recommendations as being a matter for personal style (see section 2.3.4) or as being irrelevant to the programming task (see section 2.2.3). The software project managers either perceive identifier-naming not to be a cost-effective issue or they see it as an additional cost to the software development budget that can be relegated to the software maintenance budget (see section 2.3.3). As such, programmers may be educationally disadvantaged to consider identifier-naming as a consideration in software development and they may not be motivated by their management to change this perspective. Hence, it could be expected that programmers who report dedicating effort in choosing identifier names may also generate more meaningful identifier names.

3.3.9 Flexible Work Practices

Programmers are given little incentive by their management to modify the way they work regarding the choosing of identifier names and are generally reluctant to modify their work practices (see sections 2.3.4). However, modification of a programmer's work practices is required in order to overcome these educational and management short-falls. Hence, it could be expected that programmers who report that they have flexible work practices may be more receptive to acting on the reporting of identifier-naming style flaws by a software tool.

3.4 Literature Review Method

During the early phase of the research, papers that appeared of interest and/or relevance to the research were sourced and read. A paper was considered highly relevant when it either directly supported or detracted from establishing a link between source code readability and identifier naming. A paper was considered relevant if it contained information that could support at least

a partial response to the research questions. Where an identified paper quotes other sources, in the areas of interest to the research, those other sources were obtained and in turn read. This activity was conducted to: (1) ensure that the first paper had interpreted the referenced paper correctly, which was not always the case and (2) to identify whether additional material of relevance was available in the referenced papers. In addition the papers appearing in the Empirical Studies of Programmers series of workshop were scanned for relevant material. This ad hoc method resulted in a number of relevant papers being identified but gave no understanding for the completeness of the search conducted.

Subsequent to the ad hoc approach and when an understanding of the research scope was available, a formal systematic review was conducted. The systematic review employed electronic libraries that have previously proven to be useful. Considering the journal articles found using the ad hoc method, a large proportion of these journal articles were found to be contained within the ACM, Elsevier, IEEE, Springer and Wiley families of journals. To facilitate the systematic review, the electronic libraries chosen for access were the ACM digital library; the IEEE Explorer library; the Science Direct (Elsevier) library with results limited to "Computer Science"; Springer Link with results limited to 'viewable' articles only and encompassing the following journals: "Automated Software Engineering", "Cognition, Technology & Work", "Cognitive Processing", "Education and Information Technologies", "Empirical Software Engineering", "Innovation in Systems and Software Engineering", "International Journal on Software Tools for Technology Transfer", "Journal of Computer Science and Technology", "Programming and Computer Software", and "Software Quality Journal"; and the Wiley InterScience library with results limited to "Computer Science" journals only.

Analysis of the research questions resulted in the following list of keyword phrases being used to search the electronic libraries: "naming convention", "naming style", "identifier naming", "code readability", "code maintainability", "code quality", "software readability", "software maintainability" and "software quality". However, this set of keyword phrases failed to find two highly relevant papers which had been previously found; hence the keyword phase set was increased to include the following keyword phrases: "program readability", "program maintainability" and "program quality". The resultant keyword phrase set was then able to find all papers that were previously found to be highly relevant. The search criteria was further extended to include the word phase "mnemonic name", when it because apparent that some early papers use this word phrase with reference to the readability of identifier names.

The systematic review was conducted from 27Nov05 through 16Dec05 and extended further on 20Jan06 to address additional searches. An anomaly appeared when it was discovered that the initial search using the Wiley InterScience library with the key word phrase "software quality" initially resulted in 30 hits but a week later on 07Dec05, 99 hits were received for the same search string. Investigation of the electronic library search engines discovered no anomalies with character case sensitivity but some of the search engines failed when a trailing space was used as part of the search criteria. However, the discrepancy in hit rates on the Wiley InterScience electronic library could not be duplicated. As a consequence all identified searches on all selected electronic libraries were re-run and on this occasion no further anomalies where encountered.

A search using the electronic libraries typically returned the paper's title, occasionally the abstract (or part there of) and a link to the paper. In total 12,058 papers were identified using the searches on the electronic libraries. Seven papers were found not to have links to the applicable paper's text. These papers were subsequently sourced from the physical journal. Duplicate papers were ignored with only the first instance of the paper considered. Paper titles that did not appear relevant (eg "An updated set of basic linear algebra subprograms (BLAS))" and in particular very old papers (eg "A proposal for Input-Output handling in ALGOL60" – 1965) were also discounted from further consideration. Similarly, papers that did not appear relevant from reading the attached abstracts were also discounted from further consideration. However, early on in the systematic review, this determination could not be readily made for fear of potentially missing a relevant paper. Hence, a number of papers were scanned containing abstracts suggesting that the paper was not relevant. No scanned paper, that had an abstract suggesting that the paper was not relevant, was actually found to be relevant. This new found confidence allowed papers to be considered and potentially rejected, from further consideration, considerably faster than would have been possible if each paper was scanned for relevant content. In addition to scanning the papers found, any paper cited within the paper that appeared relevant was also opened and similarly scanned.

Table 3-3 lists the numbers of papers found to be useful or were discounted at the indicated scanning step. Scanning a paper consisted of manually scanning the text for relevant content, reading the section headings and conducting an electronic search, within the document, for the words "identifier" and also "name" and then reading the text near each occurrence of the search words. In this way, two papers that were highly relevant but had not previously been sighted in addition to a number of less relevant papers were found.

Table 3-3 - Electronic Search Result Breakdown

Category	Papers
Discounted on Title	606
Discounted on Abstract	966
Discounted on Body	321
Previously not Seen and Relevant	47

The papers identified as being relevant along with the relevant papers previously found using the ad hoc process were read and relevant text copied to file. The purpose of this exercise was to facilitate electronic searches in preparation for the argument synthesis. In total 159 papers and 29 books were 'written up' to file. However, not all of the references initially considered as being useful to the thesis argument, actually proved to be useful to the thesis argument. Hence not all references that were written up have been cited in the bibliography.

3.5 Programmer Characteristics Questionnaire

The intent of the Programmer Characteristic questionnaire (see Appendix B – Programmer Characteristics Questionnaire) is to aid in the collection of programmer characteristics from programmers who have participated as test subjects so that values may be assigned to the confounding variables discussed in the previous subsections. The programmer characteristics data is needed to identify whether a disproportionate number of test subjects, with any one polarised programmer characteristic were represented in either of the control group or the experimental group which could hence skew the experiment result.

Once an experiment has been attempted by a test subject it can not effectively be re-run using a different value for the independent variable, due to the tendency of test subjects to learn from previous experience. As there is no ethical way to remove this experiential learning, it is appropriate to compare the results of test subjects that are the most similar to each other in order to identify whether a programmer characteristic has actually had some effect on the experimental result as predicted by the foreground theory.

Justification for an interest in the programmer characteristics has been previously made, however, discovery of a correlation between programmer characteristics (e.g., number of years programming experience) and behavioural factors (e.g. generation of superior source code readability) does not necessarily mean that there is a causal relationship between them

(Detienne, 2001). Hence, correlations between the final number of identifier-naming style flaws remaining in the source code generated during the maintenance and production experiment will not be calculated as no further analysis is appropriate in the absence of a foreground theory predicting correlation between the dependent variable and the confounding variables.

3.5.1 Questionnaire Statement Wording

The programmer characteristics to measured by the confounding variables would require some considerable amount of effort to extract using formal means, hence the questionnaire respondents were asked for a self assessment. As it can be argued that the ability of a test subject to articulare their responses may have some bearing on an open question, hence the questionnaire statements were specified as closed questions so as not to discriminate against the less articulare questionnaire respondents. Oppenheim (1992, pp99-100) notes that, the use of closed questions also facilitates quicker responses and is easier to code as the questionnaire respondent is given a list of response alternatives to select from. Oppenheim (1992, p106) also notes that experts do not agree on the number of response alternatives that should be provided to the questionnaire respondent. However as for attitude questions, the number of response alternatives has been limited to five.

The design of the question wording borrowed from de Vaus (2002, pp96-97) and Oppenheim's (1992, p187) suggestions for questionnaire statement wording, and the ordering of the individual questionnaire statements is in keeping with the suggestions made by de Vaus (2002, p110). The Programmer Characteristics questionnaire consisted of nine questions, which are discussed below.

Group Name

The group name was initially perceived as being a useful mechanism for data separation, should the programmer save their data to a common directory structure. As data separation did not become an issue, the responses to this question were ignored.

Professional Programmer

Requesting that programmer identify their current software engineering status was discounted as the question is potentially ambiguous without a universally accepted definition of software engineering professional categories. A response granularity of two values (i.e., yes or no) to identify whether the programmer was or was not a professional programmer was considered sufficient to code for this question.

Years Programming

As previously defined, the qualification for an expert programmer required as a minimum that ten years programming experience working to develop mission critical computers systems was required. Hence the transition to expert programmer status was of interest to the research. As such, the question was couched in terms of programmer years of experience gained. A variable range was used in the question to allow for the separation of novice, intermediate and expert programmers. The question was coded to correspond to the values: 1, 2, 3, 4-9 and 10+ years. On retrospect, additional graduations for the last two options may have been more useful for the purposes of data analysis.

Large Program Experience

As the word 'large', in its association with computer program size, has been found to be ambiguous as the definition depends on the actual experience of the programmer, the question was coded to correspond to five ranges of computer program size, increasing exponentially. A logarithmic scale was employed to specify computer program size ranges, as a linear scale would have resulted in a larger number of response alternatives being specified for no real apparent benefit. The response alternative offered consisted of 0-1k, 1k-2k, 2k-5k, 5k-10k and 10k+ SLOC.

Code Review Experience – Subject

Requesting that the programmer recount their prior code review experience was discounted as the question is potentially ambiguous without a universally accepted way of defining this experience. As such, the question was couched in terms of whether the programmer had had their source code reviewed at any time, was considered sufficient. Hence, a response granularity of two values (i.e., yes or no) was considered sufficient to code for this question.

Code Review Experience – Reviewer

As above, a response granularity of two values (i.e., yes or no) was considered sufficient to code for whether the programmer had ever reviewed someone else's source code.

Touch Typing Ability

Whether a programmer can touch-type was given low importance and a response granularity of two values (i.e., yes or no) was considered sufficient to code for this question.

Identifier Naming Effort

As it is unlikely that a definition unambiguously defining the 'proper' amount of effort required to be put into naming an identifier can be simply defined, responses to this question were enumerated due to the highly subjective nature of the question. A response granularity of five values was considered sufficient to code for this question. The response alternatives offered consisted of the enumerations: "much more than average", "more than average", "about average", "less than average" and "much less than average".

Flexible Work Practices

As there is no definition available to quantify what identifies as a flexible work practice, responses to this question were enumerated due to the highly subjective nature of the question. A response granularity of five values was considered sufficient to code for this question. The response alternatives offered consisted of the enumerations: "much more flexible", "more flexible", "about average", "less flexible" and "much less flexible".

3.5.2 Questionnaire Presentation

As the programmer characteristics are only relevant where the questionnaire respondent has participated in the maintenance and production experiment or in a case study, the source code editor was constructed to incorporate the Programmer Characteristics questionnaire. In addition to the questionnaire, help information was also available to the questionnaire respondents. Figure 3-1 depicts the questionnaire, as it initially appeared to the questionnaire respondent. The questionnaire respondent responded by using the mouse to click on a radio button. The questionnaire respondent's responses were saved to a log file when the Questionnaire window was closed. The log file was subsequently bundled with the source code generated by the test subjects to support future analysis of the data.

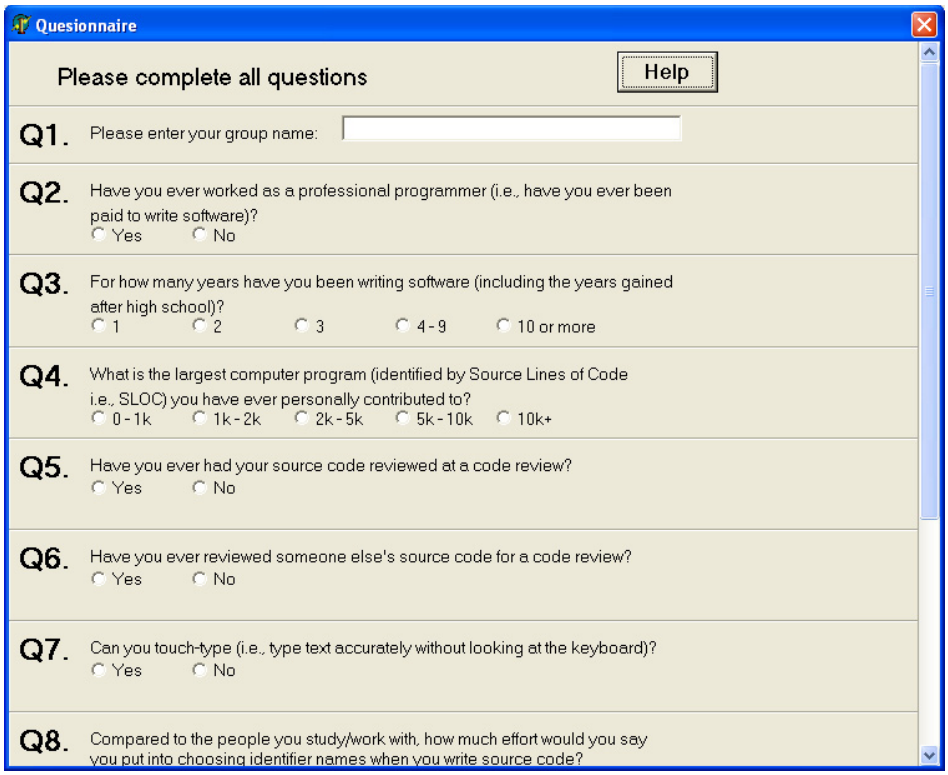

Figure 3-1 - Programmer Characteristics Questionnaire Presentation

Note: The test subject was required to scroll the window to see the ninth question.

3.6 Identifier-Naming Style Guideline Programmer Acceptance Questionnaire

The intent of the Identifier-Naming Style Guideline Programmer Acceptance questionnaire (see Appendix C – Identifier-Naming Style Guideline Programmer Acceptance Questionnaire) is to identify, through comparison against a standard naming style for a generic identifier name, whether the questionnaire respondent considers the standard naming style or an alternative identifier-naming style to be more readable. The questionnaire is not capable of definitively establishing the relative acceptance of the programming community towards the identifier-naming styles used by the questionnaire as only one example per guideline is presented to the subject. Rather, the intent of the questionnaire is to gauge the indicative acceptance of a generic identifier name over that of a similar identifier name that results in a deviation from one of the identifier-naming style guidelines.

3.6.1 Questionnaire Statement Wording

The Identifier-Naming Style Guideline Programmer Acceptance questionnaire's intent is to identify patterns amongst the attitudes of programmers and is not so much interested in group differences in attitude, attitude change or the hierarchical structure of attitude. Oppenheim (1992, p188) states that the Likert attitude scale is most appropriate when considering attitude patterns amongst groups of questionnaire respondents. The questionnaire respondents were required to indicate the degree with which they agree with or disagree with each of the identifier-naming style guidelines, i.e., they are being asked to express an opinion regarding their attitude to the guidelines. As an individual's opinions are typically dormant and are only expressed when the object of this opinion is perceived (Oppenheim, 1992, p174) source code examples accompanied each attitude question in order to elicit, by way of the example, the choice between accepting the identifier name that agreered with the identifier-naming style guideline or the identifier name that did not agree with the guideline.

The Likert attitude scaling method requires approximately 100 respondents and the responses to be scrutinised by a number of judges (Oppenheim, 1992, p195). However, Oppenheim (1992, p195) further states that the judges are no longer required when the primary concern is uni-dimensional (i.e., where the attitude statements all measure the same thing) and a trial sample is collected. As the research is not overly concerned with the ability to place an arbitrary individual's identifier-naming attitudes relative to another's but instead is interested to identify indicatively whether programmers will accept a specific identifier-naming style guideline as being relevant to source code readability, one attitude statement guideline was considered relevant for the questionnaire and in this case, a smaller sample may also be used.

Oppenheim (1992, p180) notes that attitude questions should offer the questionnaire respondent some slight room for interpretation as the intent of the question is to collect their opinion and not prescribe a 'desired' response, hence a selection of five responses was offered to the questionnaire respondent. Oppenheim (1992, p195) states that the responses are to be selected from the attitude continuum: 'strongly agree', 'agree', 'uncertain', 'disagree' and 'strongly disagree'. These attitude positions can be scored from 1 through 5, where, the scoring order is chosen depending on whether a high score or a low score is to be indicative of acceptance of the attitude statement (Oppenheim, 1992, p195). For the purpose of this research, a low score value has been associated with acceptance of the identifier-naming style guideline by the questionnaire respondent.

The next step in the Likert attitude scaling method is to conduct attitude statement analysis (Oppenheim, 1992, p200). Instead of the total score over all questionnaire respondents being used, during attitude statement analysis, the total score minus the score for the individual attitude statement is calculated for each individual questionnaire respondent against each of the attitude statements. This consideration becomes necessary when the number of attitude statements is small (Oppenheim, 1992, p199). Any attitude statement that attracts a Pearson correlation value of less than 0.85 is usually discarded from the final list of attitude statements, as they do not sufficiently correlate to the base consideration (Oppenheim, 1992, p200). As the expert programmer is potentially in the best position to provide a response to the attitude questions that best reflects true source code readability issues, the Pearson correlation was calculated using the mean expert programmer questionnaire score as the value corresponding to the base consideration. The identifier-naming style guidelines that would be removed by this process will identify those guidelines which may potentially be ignored by the expert programmer and as such any data collected on these particular guidelines may not necessarily be relevant to establishing support for, or against, the research hypothesis.

3.6.2 Questionnaire Presentation

The Identifier-Naming Style Guideline Programmer Acceptance questionnaire was prepared as a program executable, as this medium allowed the attitude statement to be readily presented in pseudo-random order for each questionnaire respondent, something that is not generally possible for a paper based questionnaire. This was done to limit the affect of attitude statement ordering to the questionnaire response. This medium also allowed the respondent to select the computer programming language (i.e., either Ada or Java) that they were most comfortable with. The questionnaire respondent was presented with two example skeleton methods, both containing an identifier declaration and an example use of the identifier. One method declared an identifier where an identifier-naming style guideline was not violated and the complementary skeleton method declared an identifier where the guideline was violated. The identifier name was bolded in the identifier declaration and also for the identifier use, in both skeleton methods. This was done to draw the attention of the questionnaire respondent to the particular identifier under consideration. These two otherwise identical skeleton methods were displayed horizontally next to each other to aid the questionnaire respondent to make comparison between the two identifier names declared within the skeleton methods. The skeleton method that violated the identifier-naming style guideline alternated sides, from one question to the next. This was considered necessary to reduce the affects of left-right bias to the questionnaire response. Figure 3-2 depicts one of the attitude statements as it would appear to a questionnaire

respondent. The questionnaire respondent was able to visit prior selections and was required to indicate their selection by clicking on one of five radio buttons.

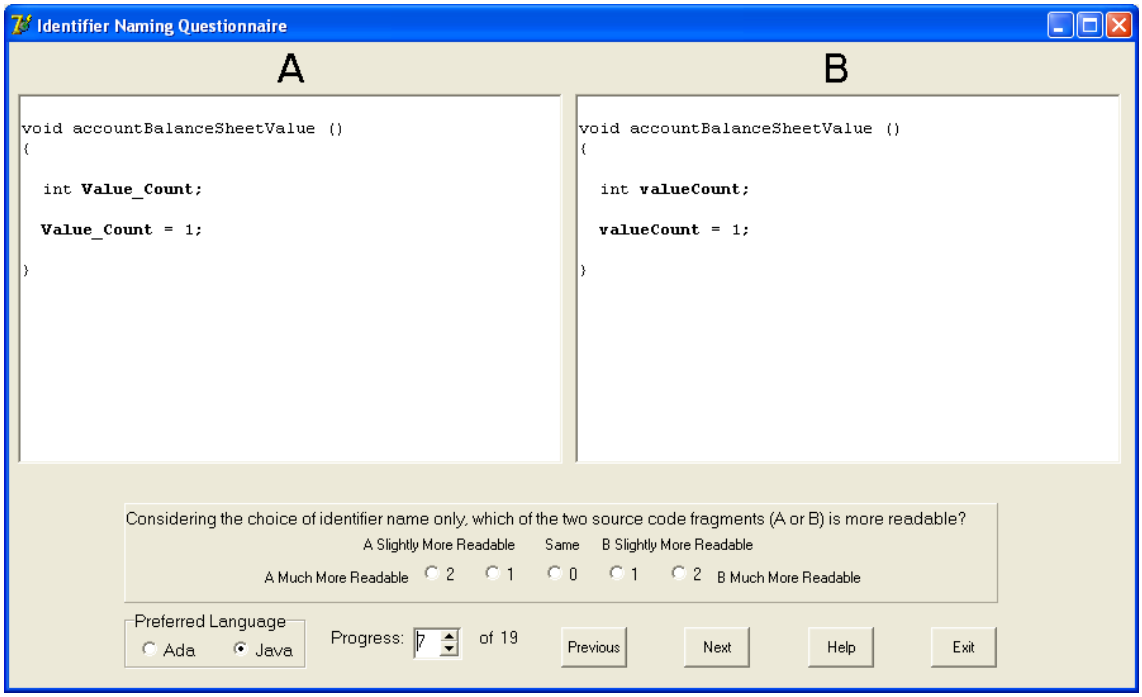

Figure 3-2 - Programmer Acceptance Questionnaire Presentation

Care was taken to provide a generic identifier name i.e., "Value_Count" in the Ada source code and "valueCount" in the Java source code, for the questionnaire respondents to consider in comparison with a second identifier declaration. However, due to the nature of three of the identifier-naming style guidelines, i.e., Class/Type Qualification (09), Constant/Variable Qualification (10) and Numeric Name (13), the artificial use of the generic identifier name was not considered desirable as the use of the generic identifier name could be considered as damaging to the source code readability in these cases and hence would result in an invalid test of the attitude statement. As both method skeletons had the same form and only differed in that identifier names that were more meaningful for the kind of identifier names chosen, this situation was not considered to present an issue for the questionnaire respondents. In addition, there were two attitude statements, i.e., Similar Name (17) and Same Words (19), that required declarations of the generic identifier and in addition a second identifier in both method skeletons. As both method skeletons had the same form, it was not expected that a questionnaire respondent would show bias to either of the method skeletons other than by virtue of the particular identifier names differing between the skeleton methods. The questionnaire presented the questionnaire respondent with a choice between a skeleton method with an identifier name that did not deviate from the identifier-naming style guidelines and a skeleton

method with a similar identifier name that deviated from one of the identifier-naming style guidelines of interest. There is little reason to suggest that generally the questionnaire respondent's selection would not also be repeated for different identifier name choices that similarly deviated from the identifier-naming style guideline.

Of the remaining fourteen attitude statements, ten required the questionnaire respondents to choose in preference of the generic identifier name and four required the questionnaire respondents to choose in preference of another identifier name in order to support the expectation that the corresponding identifier-naming style guideline directs towards improved source code readability. This heavy bias towards selecting in favour of the generic identifier declaration was of concern. However, this concern proved to be unfounded as analysis of the questionnaire results subsequently identified standard deviations recorded against the two categories of less than 1.0 and with similar mean values of 2.15 where selection for the generic identifier name was desired in order to support the corresponding identifier-naming style guideline and 2.35 for the other identifier. These results were reported for the expert programmers.

3.6.3 Questionnaire Respondent Selection Criteria

Industry typically categorises software engineers as Graduate Engineers, Engineers, Senior Engineers, Principle Engineers and Engineering Fellows. Graduate Engineers typically remain in that category for about two years as about two years is required for them to become familiar with company software engineering procedures and for them to gain sufficient domain knowledge specific to the industry that they are employed into. Hence Graduate Engineers correspond to the novice status. Engineers and Senior Engineers are in transition to expert status, and Principle Engineers and Engineering Fellows are considered to have reached expert status. There is one other category of software engineering professional, the Software Quality Engineer, who is responsible for software quality assurance on large software projects. The Software Quality Engineer has typically also reached expert status too and their focus is directed towards software engineering process.

Novice, intermediate, expert and quality assurance software professionals were selected from a Defence company, with the expectation that the acceptance of the identifier-naming style guidelines would be stronger with greater software development experience due to their exposure to a customer who imposes high software quality constraints on the development of their software. The attitudes of software quality engineers were sought as these individuals are

responsible for assuring the software quality produced by their organisations. In addition, the attitudes of software engineering academics were also sought as these individuals are in some part responsible for instilling the initial attitudes into their student programmers. The questionnaire data was grouped to establish the relative acceptance of the identifier-naming style guidelines by software professional category. In particular, meaningful comparison between novice and expert software professionals' attitudes was also sought.

3.7 Research Metrics

The software quality metrics relevant to software readability were introduced in the Background Theory chapter and the method of their calculation is specified within the following subsections. The expectations of the original researchers, for their software quality metrics, were to report software quality. The software quality metrics were tested for their responsiveness to indicate an improvement in source code readability due to the dynamic reporting of identifier-naming style flaws.

3.7.1 Cyclomatic Complexity

The Cyclomatic Complexity metric (McCabe, 1976), is the number of distinct source code execution sub-paths within a method plus one. A sub-path is created when an exception trap statement, If statement, Loop statement, procedure call, or selection statement (eg. Ada – Case, Java – Switch) is present in the source code. An If statement and a selection statement only increase the sub-path count by one, irrespective of whether the If statement has an Else part and irrespective of how many labels the selection statement supports. However, an additional sub-path is appropriate in calculating the extended Cyclomatic Complexity when an If statement or a selection statement has a condition expression containing at least one occurrence of the Boolean operators: "and", "or" or "xor".

The Cyclomatic Complexity value is calculated for each method within a software unit. The values are totalled for a software unit and averaged over all software units to give an average value per software unit for the software project. The Cyclomatic Complexity metric can be converted to a more meaningful enumeration for display using the value ranges suggested by the Carnegie Mellon Software Engineering Institute (http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html, accessed: 30Jan04) as identified in Table 3-4.

Table 3-4 - Cyclomatic Complexity Conversion

Cyclomatic Complexity Value	Enumerated Complexity
Value <= 10	Simple
10 < Value <= 20	Moderate
20 < Value <= 50	Complex
50 < Value	Un-testable

As the calculation of the Cyclomatic Complexity metric ignores all characteristics of identifier names, the use of the Cyclomatic Complexity metric is expected to be a poor predictor of source code readability. Hence calculation of the Pearson r_s value is expected to return a statistical result that is not significant for a correlation between the average Cyclomatic Complexity for a software project and the mean percentage of identifier-naming style flaws for the same software project.

3.7.2 Software Science Metrics

The Software Science metrics (Halstead, 1977) use the following four parameters in the calculation of the metric:

- n1 Number of distinct operators
- n2 Number of distinct operands
- N1 Total number of operators
- N2 Total number of operands

Identification of which program tokens should be classified as operands and which tokens should be classified as operators is not obvious. The convention used during this research was to label all computer language identifiers (i.e., classes, constants, enumerations, exceptions, methods, types and variables) and all literal values (i.e., the Boolean values “False” and “True”, character strings, characters and numbers) as operands. All other program tokens (eg. “=”, “and” etc.) are hence classified as operators, with the following exceptions: the source code comment, new line and semicolon tokens are ignored for the purposes of counting operands and operators. These recommendations are in line with those made by Mauri and Williams (1982).

The Software Science metrics are calculated using the following formula:

Difficulty (D) = 0.5 * N2 * n1 / n2

Volume(V) = (N1 + N2) * Log2 (n1 + n2)

Effort (E) = D * V

The Difficulty and Volume metric values are calculated for each method (i.e., constructor, function and procedure). The Effort metric is then calculated for each method and totalled for the software unit, and then totalled over all software units, to form the effort estimate for the software project. The Effort estimate indicates the total number of mental discriminations that would be expected to be made by a programmer in the development of the software project. In order to derive the software development time, Halstead (1977) measured the time required to write twelve machine-code programs of different length and derived a value of eighteen mental discriminations per second (Fitzsimmons and Love, 1978). This metric appears to be more useful and Fitzsimmons and Love (1978) report that Halstead (1977), who was set a second task (different from the task described above) of programming twelve algorithms in three different computer languages, found a very strong correlation between the actual development time and the predicted development time. Mauri and Williams (1982) support this value and state that a number of researchers assume the value eighteen for the number of mental discriminations per second. Hence, development time (in seconds) can be calculated as:

Development Time = E / 18

As the calculation of the Software Science metrics ignore all characteristics of identifier names, with the exception of the actual number of identifiers defined, the use of the Software Science metric is expected to be a poor predictor of source code readability. Hence calculation of the Pearson r_s value is expected to return a statistical result that is not significant for a correlation between the average Development Time for a software project and the mean percentage of identifier-naming style flaws for the software project.

3.7.3 Readability Predictor

The Readability Predictor metric (DeYoung and Kampen, 1979) uses the following three parameters in the calculation of the metric:

CC Cyclomatic Complexity

SLOC number of SLOC

VAR normalised identifier name length

These parameters are calculated for the method, with the normalised identifier name length calculated for each identifier declaration within the body of the method and averaged to construct a single value for the method. The normalised identifier name length is calculated by dividing the number of characters, composing the identifier name, by the arithmetic difference of the source line number where the identifier was last used within the body of the method and the line number where the identifier was first used. The number one is added to the difference value to protect against the possibility of a division by zero occurring when the first and last occurrence of the identifier name occurs on the same source code line.

The Readability Predictor metric is calculated using the following formula:

$$R = 0.013 * CC - 0.499 * SLOC + 0.295 * VAR$$

The Readability Predictor value is calculated for each method within a software unit. These values are averaged over all software units within the software project to form the average Readability Predictor metric for the software project.

The Readability Predictor metric value was initially used to calculate the delta between the examiner's solution and the student's solution, hence making the use of the metric questionable as an absolute measure of source code readability. These variables and normalising weightings may in addition only be appropriate to the source code that shares the same characteristics as the source code originally used to develop the Readability Predictor function and hence the choice of these variables and of the normalising weightings, may not necessarily be universal. However, an expectation remains that the metric may be useful for identifying step-wise improvements in source code readability from one edit action to the next; conditional that the delta summation is reset when an identifier declaration is either deleted or inserted as such an action on the part of the programmer would disrupt the basis for comparison of metric values. Hence calculation of the Pearson r_s value may return a statistically significant result for a correlation between the average Readability Predictor for a software project and the mean percentage of identifier-naming style flaws for the software project. However, this statistical result would be questionable in the absence of a foreground theory connecting the Readability Predictor metric with the presence of identifier-naming style flaws. The existence of a favourable correlation could however act as a catalyst to search for a suitable foreground theory.

3.7.4 Maintainability Index

Helander (1988) states that the Cyclomatic Complexity and Software Science metrics report differences in complexity for small programs but their capability degrades when they are applied to larger software systems composed of many software units. Hence, for a software quality metric to be useful for large mission critical software systems, some other metric may be required. The Maintainability Index metric (Pearse and Oman, 1995) was developed from data collected during the maintenance of software systems that range in size from 40M – 50M SLOC and is reported as being responsive to measuring the software quality of large software systems. The Maintainability Index metric uses the following four parameters in the calculation:

average V	average Volume per method
average V(g')	average extended Cyclomatic Complexity per method
average SLOC	average SLOC per method
per CM	average percent of lines of comments relative to the total number of comment plus SLOC per method

The Maintainability Index metric is calculated using the following formula:

$$MI = 171 - 5.2 * \ln(\text{average V}) - 0.23 * \text{average V(g')} - 16.2 * \ln(\text{average SLOC}) + 50 * \sin(\sqrt{2.4 * \text{per CM}})$$

The Maintainability Index value is calculated for each software unit and averaged to give a Maintainability Index per software unit for the software project. The Maintainability Index metric can be converted to a more meaningful enumeration for display using the value ranges suggested by Pearse and Oman (1995) as identified in Table 3-5.

Table 3-5 - Maintainability Index Conversion

Maintainability Index	Complexity
< 65	Low
[65 .. 85]	Moderate
> 85	High

As the calculation of the Maintainability Index metric ignores all characteristics of identifier names, with the exception of the actual number of identifiers defined, the use of the Maintainability Index metric is expected to be a poor predictor of source code readability. Hence calculation of the Pearson r_s value is expected to return a statistical result that is not

significant for a correlation between the average Maintainability Index for a software project and the mean percentage of identifier-naming style flaws for the software project.

3.8 Source Code Editor

The source code editor was constructed to support normal editing functions, and to detect and report identifier-naming style flaws in source code. The source code editor was readily configurable to either supply no reporting of the identifier-naming style flaw or dynamic reporting (i.e., as the programmer enters text). The source code editor's capability is further discussed in terms of the editing interface, the identifier-naming style flaw reporting function, assistance offered to the test subject to aid in the correction of an identifier-naming style flaw, the computer language subset parsed by the source code editor and the logging of test subjects' actions.

3.8.1 User Interface

Welsh, Rose and Lloyd (1986) suggest that an alternative to supplying a text based interface is to present the source code, being edited, as a syntax tree. However, the assumption that all programmers are capable and willing to work exclusively with syntax trees is questionable (Welsh, Rose and Lloyd, 1986). Further to this Teleman (1996) used keystroke models to demonstrate that text recognition was more efficient than syntax tree building for the editing of source code. Hence the display of a syntax tree was not considered further for the implementation of the user interface.

Dumas and Parson (1995) suggest that programmers want software tools that are easy to use and provide new capability. However, Humphrey (2000) cites early research indicating that greater choice leads to greater control. Further research bounded this prior statement when it demonstrated that too much choice overwhelms and may result in a perception of reduced control (Humphrey, 2000). Sage and Rouse (1999) observe that adding functionality to a software tool without reference to the user often results in confusion. Welsh, Rose and Lloyd (1986) believe that, in general maintaining a reasonable response time is important to the acceptance of a source code editor. Given these observations, the presentation of the source code editor user interface was limited to the usual look-and-feel characteristic of contemporary Interactive Development Environments and was developed as a Microsoft Windows application.

The application supports many of the standard editing functions available to current Interactive Development Environments.

The source code editor supports the file functions: open, close, save and save as; and the edit functions: copy, cut and paste, via the main menu. These functions were also available from standard keyboard short-cuts. The main menu, view submenu also allows the selection of the default project identifier-naming convention. The source code editor allowed multiple edit files to be simultaneously open and supported character insertion and deletion under keyboard control. The source code editor has been limited to supplying the necessary editing functions so as to minimise the test subject's learning task. The source code editor supported a single font only (i.e., 12 point Courier New); and highlighted the computer language constructs such as reserved words and comments, as well as all identifier-naming style flaws. Figure 3-3Figure 3-1 depicts the user interface to the source code editor and shows the source code, used by the Introduction exercise of the maintenance and production experiment, as it was initially presented to the test subjects.

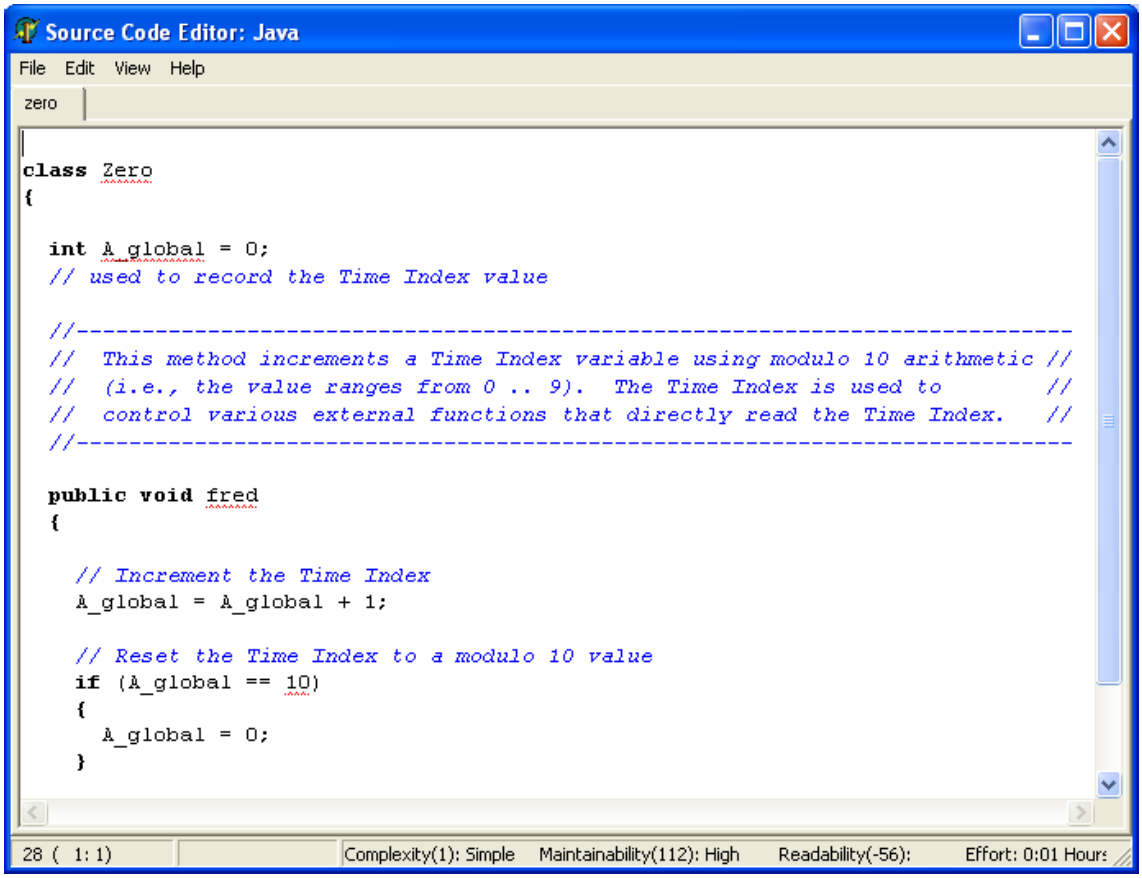

Figure 3-3 - Source Code Editor User Interface

3.8.2 Dynamic Reporting

There is some discussion as to when errors should be reported eg., when the source code editor focus leaves the program token, declaration/statement, block or on programmer demand. Welsh, Rose and Lloyd (1986) suggest that the editor should be configurable to allow the programmer to select their own preference and should allow the programmer to correct the error when convenient to them. The source code editor was implemented to periodically check, once every second, after there has been no keyboard or mouse activity and report identifier-naming style flaws as they are found. The programmer may respond to these reports at their leisure and are not required to respond immediately on the reporting of an identifier-naming style flaw. When an identifier-naming style flaw is detected, the identifier name is highlighted on the display. The test subject can then right-click on the identifier name or numeric literal as appropriate and be presented with an Identifier-Naming Guideline Activation List dialog that allows the test subject to: modify the identifier name or to define a named constant for the numeric literal as appropriate; ignore the guideline for this instance only; ignore the guideline for the project; or cancel the dialog.

On activation of the Identifier-Naming Style Guideline Activation List dialog, a list of identifier-naming style flaws that were detected is presented to the programmer and if applicable a suggested correction, using the current identifier as a basis for this suggestion, is also displayed. McIver and Conway (1996) identify that novice programmers (like all novices) have difficulty extracting the concept intended to be taught. Weinberg (1998) suggests that programming error messages should contain greater detail than is current practice. Given that the purpose of a coding standard is to guide the development of quality software, then the coding standard should include justifications and examples where this clarifies meaning (Shanklin, 1991). This suggests that a text description accompanied by an example that deviates from the identifier-naming style guideline would reinforce the guideline when reported by the source code editor. Figure 3-4 presents an example as it would be generated for a declaration of the Class identifier name “fred”.

The test subject may decide that the dynamic reporting of identifier-naming style flaws either for an individual case (i.e., single identifier declaration) or globally for any or all of the identifier-naming style guidelines is distracting to the source code editing activity. The test subject has the ability to disable the identifier-naming style guideline for the individual case by pressing the Ignore Instance button on the Identifier-Naming Guideline Activation List dialog. The programmer may also disable an identifier-naming style guideline globally by pressing the Ignore Rule button on the Identifier-Naming Guideline Activation List dialog.

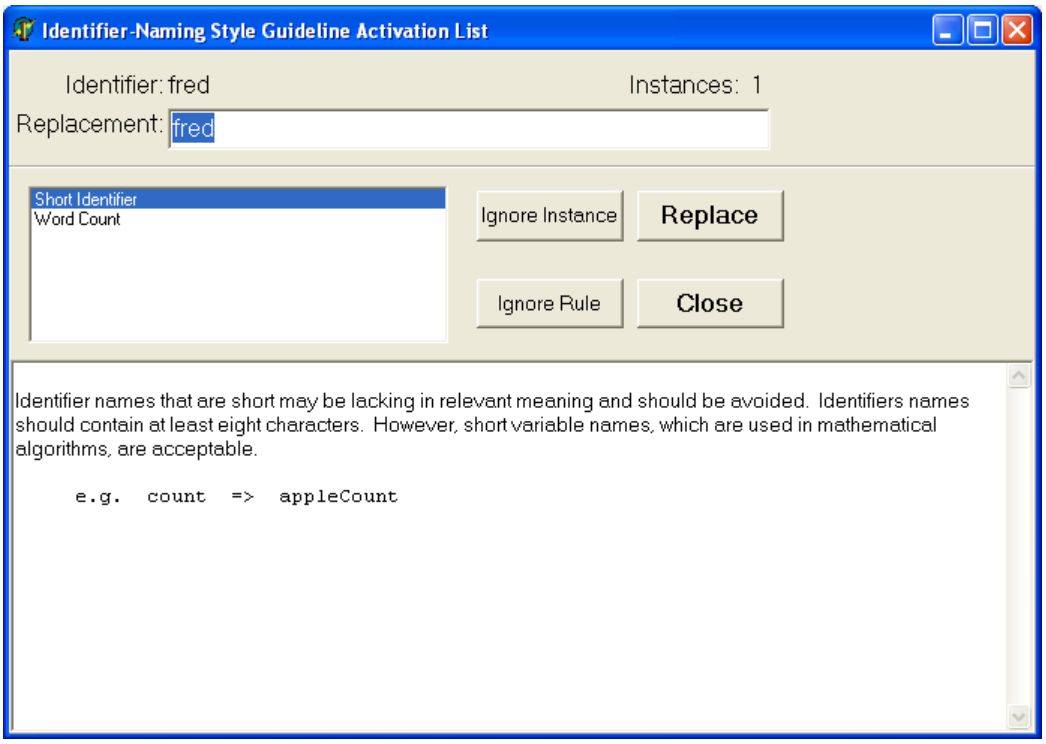

Figure 3-4 - Identifier-Naming Guideline Activation List Dialog Box

3.8.3 Identifier Name Replacement Suggestion

The nature of an identifier-naming style flaw must be known for the flaw to be detected in the first instance. Hence, there is a potential for a procedure to exist, which could modify the original identifier name such that the original identifier-naming style flaw is no longer present in the suggested identifier name. Suggested replacements for an identifier name can be offered for some but not all identifier-naming style flaws. Chapter 4 describes the identifier name conversion procedures that may be applied to modify the identifier name, as appropriate. Simple replacement of all occurrences of the identifier name that raises an identifier-naming style flaw, with another identifier name that does not raise the same flaw could result in the raising of other flaws. The identifier name conversion procedures, discussed in Chapter 4, may result in the removal or addition of characters from/to the current identifier name in the formation of the suggested identifier name, which could then result in the raising of common issues. These common issues will be addressed collectively below:

- There is potential for identifier names to clash when one identifier name is shortened by the deletion of one or more characters. Considering the example where the following two identifiers have been declared: “a_b” and “a__b”. The deletion of an underscore character from the second identifier name would result in an identifier name clash.

- There is a potential for identifier names to exceed the number of significant characters in the computer language specification or the implementation of the computer language compiler. Considering the example of an identifier name that is 255 characters in length, the addition of one more character would result in an identifier name that may be valid as stated by the Ada computer language specification (MIL-STD-1815, p2.2), however Ada compilers exist that do not support identifier names greater than 255 characters. Hence, such a modification could result in the source code no longer compiling as expected.

Any suggested identifier name is required to be checked against the identifier names in the same program scope and the suggestion presented only when an identifier name clash would not occur. Similarly, a modification to the source code should not result in the generation of an Ada text line containing an identifier that is longer than 255 characters in length. However, any identifier name approaching this length would not be considered to be an acceptable identifier name (see section 4.1.2). Once a suggested identifier name has been selected by the test subject, all occurrences of the original identifier name within the declaration scope can be similarly replaced by the suggested identifier name.

3.8.4 Design Considerations

As the source code editor must be able to recognize an identifier name at the point where the identifier is declared, it was necessary to limit the number of computer language syntaxes that are recognized by the source code editor. The set of supported computer languages was initially limited to the Ada and Java computer language syntaxes in order to support the computer language expertise of the programmers who would be introduced to the source code editor. However, this set was extended to include the Pascal computer language to accommodate the analysis of the source code supplied by the dated software survey.

Reps and Teitelbaum (1989) offer practical advice in the development of an editor, in that they suggest a subset of the computer language syntax be supported first. This advice, though initially useful, proved ultimately to be inadequate for handling the perverse syntactical arrangements used by the professional programmer. Hence the implementation of the full Ada 83, Ada 95, Java 2 and Borland Object Pascal syntax was found to be necessary in order, primarily to support the needs of the Identifier-Naming Style Flaw Analysis and Report Generator, which is described in the next section. As the source code editor was intended to be used for software development the source code editor was also required to be able to recover

from incomplete syntactical expressions without loss to the reporting of subsequent identifier-naming style flaws that may occur past the syntax anomaly.

Internally, the source code editor maintains a linked list of computer language tokens. As the programmer modifies text via the user interface the modified tokens are inserted into the existing linked list. Using the resultant linked list, a syntax tree is formed, where a node in the syntax tree corresponds to a single declaration or a single statement. The syntax tree can be rapidly traversed as the token corresponding to an identifier declaration has been previously discovered and is recorded within the node. The identifier-naming style guideline can be readily applied to the newly entered tokens that are either an identifier name or a numeric literal, and can also be applied to all other identifier declarations that could potentially be affected by the modification of the identifier declaration that has recently changed. Any new tokens are updated on the display and any tokens that show a change in identifier-naming style guideline activation are rendered on the display, as appropriate. This design appears adequate for editing small software units but results in a perceivable delay when a software unit is first opened from file, particularly for large files where increased processing is required to address the combinatorial checking and the required display rendering. However, it was subsequently discovered that by consigning display rendering to a background task, the slow user interface can be considerably quickened.

In addition to the capabilities previously mentioned, the source code editor also calculates the McCabe (1976) Cyclomatic Complexity, the Halstead (1977) Effort Software Science metric, the DeYoung and Kampen (1979) Readability Predictor and the Pearse and Oman (1995) Maintainability Index. The calculated values and their English meaning are displayed to the programmer on the source code editor's status area.

3.8.5 Session Log

The source code editor session log was implemented as a text file that logs programmer initiated editing events that could be expected to affect source code readability as a consequence of a change to an identifier name. The session log records each time stamped editing event on a separate line and reports the following events:

- The opening, creation or saving of a software unit
- Computer language selected

- Initial identifier-naming style guideline checking status for the selected computer language (i.e., whether the guideline has been activated or deactivated for checking purposes)
- Change in identifier-naming style guideline checking status
- Change in an identifier name (with the identifier name before and after reported)
- A request to replace an identifier name made by the programmer
- The current metric values calculated after a change to an identifier name or on file close
- Results of the Programmer Characteristics questionnaire responses

3.9 Identifier-Naming Style Flaw Analysis and Report Generator

The intent of the Identifier-Naming Style Flaw Analysis and Reporting (reporting capability) is to automate the source code analysis of large quantities of source code corresponding to an entire software project to identify the number of identifier-naming style flaws, the number of opportunities to generate these flaws and to calculate the software quality metrics previously identified. The reporting capability supports the ability to select the relevant computer language and select the indicative identifier naming convention. The reporting capability supports the ability to select from one of five identifier-naming conventions, which are described as "Java", "IdentifierName", "Identifier_Name", "identifier_name" and "IDENTIFIER_NAME". The "Java" identifier-naming convention is discussed in section 4.1.6 and the other identifier-naming conventions depict the actual representation of the identifier name. The reporting capability supports the ability to select from one of three computer languages i.e., Ada, Java and Pascal to identify the required computer language parsing.

The reporting capability generates a report identifying source code metrics for each software unit read. Which are tabulated or averaged, as appropriate, for the software project. The source code metrics consist of the identifier-naming style flaw data mentioned above for each of the identifier-naming style guidelines, the Software Quality metrics previously identified and the number of SLOC.

3.10 Textbook Survey

Identifier-naming standards have changed with the relaxation of compilers supporting only six significant characters in a Fortran identifier name, to 255 characters which has been typically imposed by the operating system or the computer language compiler, to potentially longer

identifier names. Similarly, compilers previously could only accommodate uppercase and numeric characters in an identifier name, whereas today (2006) compilers can typically accept lowercase, uppercase, numeric and some special characters in an identifier name. During this period of change, coding standards have also changed as our understanding of source code readability issues has similarly changed. All of which has potentially contributed to a modification in the identifier-naming style practices of the programmer working through this time period.

Identifying the changes in programming texts used during the transition from six significant characters in Fortran source code to today's relaxation on identifier naming possibilities, may offer insight into the programming practices of a programmer's source code collected over this time period. Experiential learning and external factors such as imposed project coding standards may also contribute to modifying a programmer's source code. Hence, a survey of dated programming textbooks remains important to identifying differences in individual identifier-naming style flaw instances, as being peer reviewed, the textbooks potentially reflect on programmer societal changes.

Indicative computer programming course textbooks, which were used for the education of programmers, were identified and the content scanned. Text or example source code which supports or counters the identifier-naming style guidelines was identified within each textbook. As the size of a textbook can vary considerably in the number of pages from one textbook to another, the first page number that addresses the identifier-naming style guideline either in support of or by detracting from the guideline, or the first page number that contains an example identifier declaration that is counter to the identifier-naming style guideline was recorded. The recording of examples where a specific identifier-naming style guideline is supported by the presence of an example identifier declaration was discounted. Such an approach would require that all example identifier declarations, within the textbook, similarly did not detract from the specific identifier-naming style guideline. Hence, the presence of possible support for an identifier-naming style guideline, where there was no example to the contrary, was recorded by the absence of a counter example page number, which corresponds to the recording of a null. As will be justified shortly within the method description for the maintenance and production experiment, the first page number that an ambiguous or meaningless identifier name was found in the relevant textbook was also recorded. Where the computer language, that is the subject of the textbook, does not allow an identifier-naming style flaw to occur without violating a computer language identifier syntax rule, a "N/A" (i.e., not applicable) was recorded.

3.11 Production Software Survey

The production of software post design will normally traverse through intermediate stages such as code and unit test, system test and acceptance test. During these stages, software defects may be encountered and may also be potentially removed. If during any of these stages soft defects such as identifier-naming style flaws are removed then the early reporting of these flaws during coding with consequent removal will free future resources to concentrate on other review activities. This action is desirable as the cost of modifying source code increases exponentially as the software lifecycle progresses. Hence any identifier-naming style flaws that survive to remain in production software is a candidate for dynamic reporting during the coding activity as removal at a latter stage, say during a future maintenance activity, will have greater impact on software cost than its removal during coding would have. This cost impact is manifest as a consequence of the re-work necessary to re-test the software and update documentation. However, before this determination can be made, a survey of production software source code is necessary to identify the pervasiveness of specific identifier-naming style flaws in production software. Two types of production software survey have been previously discussed i.e., a survey of dated software generated across the working life of a single individual and a survey of contemporary software which was typically generated as a product of a team based activity.

3.11.1 Contemporary Software Survey

The intent of the contemporary software survey is to establish which identifier-naming style flaws have survived to enter the maintenance phase of the software lifecycle. Due to the potential for various confounding variables to affect the relative frequencies of identifier-naming style flaws, source code was collected from different repositories to potentially limit the effects of any one confounding variable on the sample population. The relevant confounding variables in this case are identified as: (1) computer programming language and (2) program size. The computer programming languages chosen were limited to Ada and Java because of the similarity of the computer programming language syntax (as compared to the syntax of computer programming languages such as Cobol and Fortran) in that both computer programming languages are block structured and the declaration of local variables is common practice with the use of these computer programming languages. However, the computer programming languages remain sufficiently different to ensure the possibility that different identifier-naming practices will be used in the construction of a computer program. The program size has potential to skew the data collected. For instance, small software projects can be adequately developed with poor identifier-naming practices abundant in the source code, see

section 2.1.2. Hence, the identification of program size as a possible confounding variable.

Source code was collected from both Ada and Java applications; included a range of software project sizes. The source code collected for each application consisted of the entire source code necessary to build the computer program. Each computer program was analysed and a report generated for each computer program. Comparison between data collected against the computer programming language confounding variable was averaged separately for the Ada and the Java computer programming languages. Comparison between data collected against the program size confounding variable was averaged separately for the ranges of computer program size. The computer program size ranges corresponded to the same computer program size ranges used by question 4 of the Programmer Characteristics questionnaire (see Appendix B – Programmer Characteristics Questionnaire). The Spearman Rank Correlation Coefficient test was used to identify whether the data collected, which was distinguished by each of the confounding variables in turn, would belong to the same population or a different population.

3.11.2 Dated Software Survey

The intent of the dated software survey is to establish which identifier-naming style guidelines may be more usefully reported to a programmer at specific times in their professional development and to establish whether the predominance of identifier-naming style flaws changed with programmer experience. An individual programmer will not generally use the same identifier-naming style during her/his career as influences such as coding standards and experiential learning is expected to modify this style over time as more knowledge is amassed. The dated software survey concentrated on data collection over the period expected to correspond to the transition to expert programmer. Hence, software generated both before and after the ten-year anniversary of commencing work as a programmer engaged in the development of large mission critical systems was required to be collected.

3.12 Maintenance and Production Experiment

The intent of the maintenance and production experiment is to identify whether dynamic reporting of identifier-naming style flaws by a controlled experiment, can reduce the number of flaws relative to those generated by the control group during both a software maintenance exercise and a software production exercise. The maintenance and production experiment is composed of an introduction exercise, a maintenance exercise and a production exercise in that

order, and is supported by the Programmer Characteristics questionnaire. The Programmer Characteristics questionnaire was integrated into the source code editor and has been addressed previously. In addition, the test subjects were also given a set of instructions, see Appendix D – Test Subject Ethics Release and Instruction. The introduction exercise requires the test subject to replace poor identifier names within a software unit with more appropriate identifier names. As the test subject is given pre-existing source code to modify, and the required editing is extensively automated by the source code editor and fully described by the instructions, it was expected that the novelty of the source code editor interface would cause minimal distraction to the test subject in the completion of their task. The maintenance exercise requires the test subject to make minor changes to a software unit as directed. As the test subject is given pre-existing source code to modify, it was expected that the test subject could complete their task without further distraction in using the novel aspects of the user interface. The production exercise required the test subject to produce new source code as directed by a design statement. For the production exercise, the test subjects were given a skeleton software unit to edit, thus reducing the amount of time that they would need to spend before they were in a position to start entering identifier declarations.

3.12.1 Introduction Exercise

The introduction exercise was defined to be a very simple exercise to allow the test subjects to become familiar with the capability of the source code editor. The test subjects were requested to replace three identifier names and to declare a named constant within a 12 SLOC Java class software unit. See Appendix E – Experiment Files for a listing of the introduction exercise source code.

The file was 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, it was not anticipated that editing the file as requested by the test subject's instructions would present any difficulty for the test subjects other than the inherent difficulty that the test subjects would experience in devising suitable identifier names.

3.12.2 Maintenance Exercise

The maintenance exercise used contrived source code, however the identifier-naming style practices used in its construction were representative of production software. The identifier

names represented in the source code were mostly abbreviations of English words that corresponded to their actual function. However, in addition there were examples of meaningless identifier names and also declarations that did not deviate from the identifier-naming style guidelines as implemented by the source code editor. The test subjects were requested to make the following modifications to a 118 SLOC software unit: (1) insert a new colour i.e., violet into the current list of colour enumerates and update the Successor function as appropriate, (2) extend a selection statement to return the 4th prime number, (3) increase an array size, (4) change the secondary sort key, (5) correct a typographical error, (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. See Appendix E – Experiment Files for a listing of the maintenance exercise source code.

The test subject's instructions contained considerably greater detail that that presented above and as part of the instruction the test subject was given the identifier name that was the subject of the maintenance action, so that the test subject would have no difficulty in finding the exact line(s) requiring modification. The programming logic complexities of the maintenance exercise source code was not expected to result in any difficulty of understanding for the test subject as the algorithms implemented by the source code were simple and are typically introduced to programming students early during their undergraduate education.

3.12.3 Production Exercise

The production exercise required that the test subjects code the Exchange Sort algorithm as a Java method. The test subjects were given a software unit that contained at least one comment per required identifier declaration. The comment described the need for the identifier. In addition, there was one comment per main control statement that described the required function to be implemented. See Appendix E – Experiment Files for a listing of the production exercise source code. As the test subjects should be familiar with the Exchange Sort algorithm from their previous studies, particularly as they had just been asked to make a minor modification to an Exchange Sort method in the maintenance exercise and in addition the logic of the Exchange Sort method was contained in the file they were asked to edit, it was expected that the programming logic would not result in any difficulty of understanding for the test subjects.

3.12.4 Test Subject Selection

Weissman (1974) discovered that first year undergraduate students are not appropriate for programming experiments as they are prone to experiencing significant problems with programming constructs associated with the programming language before they can attend to the needs of the experiment. Subsequent experimentation by Weissman (1974) discovered that second year students were acceptable as test subjects for a programming experiment. To ensure that the students would not be unduly affected by programming language considerations, students who had completed sufficient programming course work to be familiar with the Java computer programming language syntax such that they could finish the editing tasks without the need to compile the resultant source code were identified as potential candidates to act as test subjects for the research.

Detienne (2001) raises concerns whether experimental results gathered using novice programmers as test subjects and small source code listing can be meaningfully extrapolated to real programming tasks and are relevant to software engineering theory in general. The choice of test subjects included novice programmers as the research is keen to identify whether novice programmers improve the readability of their source code when assisted by the dynamic reporting of identifier-naming style flaws. However, the research is also interested in the responses of professional programmers and so a representative sample from this group was also engaged as test subjects. The issue that Detienne (2001) raises, with regard to small program size, remains relevant. However, the research is concerned with the naming of identifiers, which are limited in their influence by their programming scope. The scope of identifier declarations have not otherwise been artificially constrained during the research investigation. Hence results obtained by the current research should be relevant to large software systems.

3.12.5 Experiment Delivery

The source code editor supports the reporting of identifier-naming style flaws which is, not normally supported by the function of an editor. Hence, an introduction to this capability was presented to all test subjects so that a novel user interface would not be disproportionately distracting to the experimental group. The introduction exercise was conducted with the full capability of the source code editor available to test subjects as this allows the test subjects to commence the exercises that followed with the same baseline experience pertinent to the source code editor. The control and the experimental groups were conducted separately so as not to

cause potential disruption if one test subject should notice that another test subject had access to a differing source code editor interface.

Weinberg (1998) 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 (Weinberg, 1998, p35). 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. Weissman (1974) 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 student's time management skills were not adequate to managing the timely completion of each task. However, it was not considered prudent to force the test subjects 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 would result in source code modification. 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 software units 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 software unit.

Field Marshal Helmuth von Moltke said that "no battle plan ever survives contact with the enemy." Similarly, no experiment can be expected to experience contact with programmers without uncovering unforseen usability problems. Most software useability problems can be uncovered by introducing a software product to a small number of people. Dumas (1995) cites Virzi (1992) who found that 80% of usability problems were discovered by five test subjects. Initial testing was conducted with a first year computer science student who had minimal programming experience; a Human Factors expert who had a computer science degree; and a mature age novice programmer who had recently completed their Computer Science degree. This particular mix of initial test subjects were chosen as it was expected that they would find most issues that would be encountered by the test subjects, who would encounter the experiment and that exposure to further test subjects would be unnecessary, primarily due to the expected simplicity of the task being requested of the test subjects. The initial test subjects were given a copy of the instructions for the maintenance and production experiment and were asked to follow the instructions. The initial test subjects were requested to identify any issues that they had with the source code editor, which was configured to supply dynamic reporting of identifier-naming style flaws, while they were following the instructions. Similarly, the initial test subjects were also asked to report any issues they had with the interpretation and execution of the instructions as they performed each instruction. After conducting the experiments with

the initial test subjects any issues uncovered were attended to before the experiment were conducted on a larger sample space. In addition, the initial test subjects were asked to identify any moral objections they would have to responding to the embedded Programmer Characteristics questionnaire and also to using such a software tool within industry. This activity was taken as a tool that raises moral objection from the users cannot be expected to be effectively used by those users.

Evidence of a predicted correlation between the reporting of an identifier-naming style flaw and the subsequent removal of this flaw by the test subject would require that the test subject's actions be recorded and analysed. The reporting of an identifier-naming style flaw and then the near immediate removal of the flaw would tend to indicate a strong causal relationship. An action by the test subject to remove further occurrences of this same flaw also tends to strengthen the causality of the relationship. The reporting of an identifier-naming style flaw and then failure of the test subject to remove the flaw would tend to indicate that no causal relationship exists for the particular flaw and the reporting function. However, it may be that the programmer is disinclined to accept the validity of a specific identifier-naming style flaw being reported but will act on the reporting of the same flaw on another instance. Removal of an identifier-naming style flaw without the explicit reporting of the particular flaw neither reduces nor strengthens the causality of the relationship as it is not possible, without direct enquiry, to identify whether the programmer was independently aware of the current identifier name being considered to be an identifier-naming style flaw at the time of the test subject's action. Direct enquiry of the test subject's motives was not undertaken as such an action could affect the test subjects' future actions as the intent of the experiment was to identify how dynamic identifier-naming style flaw reporting affects the source code and not the programmer actions.

3.12.6 Test Subject Instruction

Cohen, Swerdlik and Phillips (1996) cite studies indicating that test subjects generate different test scores depending on the attendance, familiarity, gender and general manner of the examiner. Given that these attributes affect test subject behaviour, instruction was given to the test subjects by the same person on all occasions. The student test subjects conducted the experiment in a tutorial room, in which they were buffered from external distractions, and were given instruction both verbally and in written form at the commencement of the experiment. The test subjects were made aware of the nature of the experiment, given clear instruction on what is required for participation, assured that data confidentiality will be maintained and subsequently

given access to the results. These requirements, on conducting experiments with test subjects, have been levied on the research by the University of Technology, Sydney's ethics policy. Cohen, Swerdlik and Phillips (1996) also support these requirements for the use of test subjects. The test subjects were also afforded the opportunity to ask questions in an open forum before they started work and could ask questions individually after they had commenced the experiment.

The professional programmer test subjects conducted the experiment at their own work desks and were given instruction in written form only. Verbal instruction was not appropriate for the professional programmer test subjects as this group were familiar with written work statements, corresponding to the written instructions, as this is part of their normal working practice.

The test subjects were requested to send the software units that they modified and the log files that were automatically generated to an email address at the completion of the experiment. The professional programmers consisted of programmers employed by academia and also programmers employed by the Defence industry. The academic programmers were offered two cinema tickets for their participation. The industrial programmers were not offered remunerations and they offered their participation freely, as they were not compelled to participate. In addition, both the academic programmers and the industrial programmers were given freedom to complete the experiment during their normal work hours.

3.12.7 Statistical Tests

The number of test subjects required to indicate a statistical correlation in the experiment results may need to be quite large. As Brooks (1980) observes, the variability in programmer performance necessitates hundreds of test subjects in order to obtain results that are significant. Otherwise there is a possibility that individual differences, amongst the test subjects will hide the experimental effects. As the availability of hundreds of programmers was not realised by the experiment, consideration of individual differences, as identified by researchers as affecting identifier naming was undertaken. Statistical analysis was used to show which experimental results are statistically significant by separating the data into groups corresponding to the individual differences as defined by the relevant confounding variables previously identified. As the data collected which corresponds to the percentage of meaningful identifier names as identified by an expert programmer, is represented by a continuous value range (i.e., 0 through 100%), any statistical significance tests would require the use of the Mann-Whitney

(directional) test for comparing two unpaired groups and the Kruskal-Wallis test for comparing three or more unmatched groups.

3.13 Case Studies

The case studies were conducted to identify the effects on the generation of and the subsequent removal of identifier-naming style flaws from source code written by programmers. The programmers were required to work to normal industry software development practices as specified by the corporate Software Engineering process. The corporate Software Engineering process consists of a collection of procedures which mandates the software development procedures necessary to progress each activity during software development. To result in the minimal disruption to the normal industry software development process, the programmers were required to code and unit test their software as per the corporate Software Engineering process and then to use the source code editor to improve the identifier naming used in their source code. This additional step was considered necessary, as the programmers insisted on using their preferred editors to develop source code and the Java Interactive Development Environment to debug source code. As the source code editor was developed as a research tool, the programmers did not trust the robustness of the tool and the source code editor was also perceived to run too slowly by the programmers to seriously consider its use for source code editing, given the size of the programming task, this concession was accepted.

3.13.1 Task Specification

Conducting a case study that would be responsible for generating a large software system in the order of 1M SLOC would be prohibitive as typically the Defence industry requires a decade to complete a mission critical software system of this size. In order to support a realistic software development activity, a software project with an estimated size close to 100k SLOC was considered adequate. In order to minimise the effects of confounding variables, the case study should ideally include a group of programmers with differing years of programming experience ranging from novice through to expert programmer, and also be conducted in a normal industry environment.

As the normal industry environment requires programmers to work from supporting documentation consisting of a Software Requirements Specification (SRS) and a Software Design Description (SDD), these documents were generated in support of the case study. The

SRS specified the software functionality, software constraints, design of the screen layouts and software test criteria necessary to satisfy the specified software requirements. The SDD defined the high-level (i.e., software architectural) design, the low-level (i.e., software unit) design and the software test criteria necessary for software unit testing. For each software unit, the design description contained within the SDD, identified the required object attributes (i.e., internal data structures), specified the method prototypes and described the method function for the constructor and all public methods required of the class. The description of the method functions used English statements with at least one sentence dedicated to describing the function of each internal control statement. Both documents were wholly generated by the author, with the SRS containing 108 pages and the SDD containing 185 pages.

As some identifiers are explicitly named by the standard industry SDD, the test subjects were instructed that they were at liberty to modify any corresponding identifier name within their source code and were also allowed to modify any design consideration, contingent on their maintaining the accuracy of the SDD and that the resultant design was also consistent with the SRS.

The programmers were informed of the nature of the experiment and were requested to generate highly readable source code as the source code may subsequently be read and modified by the corporate customer. The programmers were asked to make their identifier names meaningful so that any effect apparent from the use of the source code editor could potentially be due to identifier-naming practices superior to those that the programmers would individually bring to the coding activity.

The case study collected data from each software unit as it completed unit test, with the before and after software units collected for analysis. The programmers were requested to only apply the source code editor's dynamic reporting of identifier-naming style flaws after unit testing of a software unit had been completed. This was to allow the programmer to complete their normal development of a software unit, which included considering the source code readability necessary to support the system integration process and subsequently for system maintenance and also to allow normal team code reviews to occur. It was expected that allowing the programmers to only use the dynamic identifier-naming style flaw reporting after completion of unit test would highlight the benefits possible from the dynamic reporting of identifier-naming style flaws at a minimal level compared to what would be possible from earlier reporting and hence the results of the case studies would potentially undervalue the actual benefits possible.

The test programs generated during unit test were not used to supply data for the case study. The nature of a test program is to artificially exercise a software unit to identify unexpected results (i.e., software bugs). As such, the definition of identifier names and the typically linear sequence of the test code are sufficiently artificial to have questionable value to the case study data capture activity.

3.13.2 Task Staffing

As the case study was funded as a corporate research and development project, the allocation of staff was subject to corporate staff availability. Initially the project was staffed by a novice programmer. However, subsequently both internal and external factors caused the hibernation of the project. Subsequently, the project was restarted and a team of experienced programmers were allocated to the task. Some dynamic modification to the research method was required to support the diverse staffing changes apparent during the project. These modifications to the research method are described in the following subsections.

3.13.3 Novice Programmer Case Study Method

The novice programmer allocated to the case study was found to have a very rudimentary understanding of software object concepts and the SDD was written extensively as a functional design with the software units only loosely representing an object design. The novice programmer was not familiar with the corporate Software Engineering process and also demonstrated considerable difficulty in developing testable software. Hence, the corporate Software Engineering process was simplified by the removal of detail that was not relevant to the project and some explanatory material was made available to the novice programmer.

The novice programmer was requested to code and unit test each software unit by working to the following steps in the order given: (1) Generate a test program by directly entering source code into an editor, (2) Write the software unit method's source code using pen and paper only, (3) Using the test program listing and the paper copy of the software unit desk check the source code by execution against the test program, (4) Type the software unit prototypes and supply sufficient source code to support syntax error-free compilation into an editor, compile and build the test program, (5) Insert the source code into one method at a time and test that method until all methods have been tested for the software unit, (6) After successful completion of unit testing the software unit is to be read by the source code editor and the source code modified to

improve readability, (7) Re-test the resultant software unit using the test program. Steps (1) through (7) were required to be completed for a software unit before development of the next software unit was to start.

This method was chosen to bring the novice programmer 'up to speed' with the quality requirements considered necessary by industry and also to ensure that the novice programmer was very familiar with their source code before they were offered any direction by the source code editor. Step (1) required the novice programmer to consider the external interface to the class methods hence offering the novice programmer an opportunity to understand the processing required for each method. As the SDD described the minimum testing required for each method and specified the method prototype, it was not expected that this activity was premature. Step (2) required the novice programmer to concentrate on the class method's internal statement structure and to declare identifiers as they were required without necessarily being distracted by the need to supply meaningful identifier names or use correct syntactical constructs. As Detienne (2001) observes, novice programmers will prefer to use a bottom-up strategy that involves frequent review of prior efforts. This practice typically results in source code statements being written before proper consideration is given to naming the embedded identifier. This step was found to be necessary as the novice demonstrated considerable difficulty in following the design description. Step (3) required the novice programmer to review their source code for correctness and potentially for meaning as well. Step (4) was used to reduce potential confusion to the novice programmer which could result from issues relating to the generation of a syntactically correct program that would run (but not do anything of consequence) from any programming issues that the novice programmer may introduce at the next step. Step (5) required the novice programmer to revisit their source code and to once again consider the meaning of the source code as they typed the source code into an editor. While the novice programmer typed the methods that were purposefully designed to be small to trivial in size, it was expected that cognitive loading would be slight thus affording the novice programmer a full understanding of the method and that they would then be able to reflect on the meaning of each identifier name as they entered the source code. At step (6) and only after the test subject was supposedly very familiar with their own software and hence had been offered ample opportunity to improve the readability of the source code they were required to use the source code editor as a guide towards improving the readability of their source code further. Such steps were found necessary to progress the novice programmer through the software development activity.

A criticism could be made that the software development procedure imposed on the novice programmer does not reflect common programming practices and hence any results thus

obtained are of limited value as comparison with published studies will not be possible. Put more succinctly, the results are not repeatable. Such a criticism may be valid when the development of small programs is being reported. However, industry does impose similar software development requirements for the development of mission critical software systems. For instance, Sommerville (1995, p431-438) describes a process similar to the bench-checking activity specified previously for the novice programmer, as being typically mandated for the development of safety critical software units. The Defence customer as well as the Capability Maturity Model mandates the use of the review process and the use of test programs, to debug, software is mandated by the corporate Software Engineering process of many Defence contractors.

3.13.4 Programming Team Case Study Method

The experienced programmers took immediate ownership of the SDD and subsequently updated the document to represent a more pure Object Oriented design. Minor changes to the user interface were also suggested and adopted into the SRS. As the allocated programmers had considerable experience, the software development method required for the novice programmer was not applicable to the experienced programmers.

3.13.5 Data Collection and Analysis

The before and after source code was quarantined and the Identifier-Naming Style Flaw Analysis and Reporting computer program was used to collect raw data from the source code. The data was tabulated chronologically, relative to the order in which the software units were developed, individually for each programmer. The data recorded, against each identifier-naming style flaw was the percentage of flaws against the opportunity to raise the flaw in the software unit. Linear analysis of the data was conducted to identify any trends over time and these trends were compared against the expectation of improvement identified from the dated software survey. Data contained with the log files, generated by the source code editor, proved to be meaningless due to the way the source code editor was used by the programmers and hence the log files were ignored as a source of data.

The programmers were asked, at the end of the case study, to list their observations regarding the use of the source code editor. This was undertaken to identify whether the programmers found the tool to be useful either in its current form or in some other form.

3.14 Research Ethics

The ethical implication of the research on the participants and programmers in general, has been considered. The maintenance and production experiment design requires programmers to modify existing source code and to develop new source code from a design specification. Neither of these activities is outside of the normal day-to-day activities that a programmer would be expected to encounter during their training or in the work place. Programmers, almost certainly from their attempts to compile their very first programs, would be familiar with the concept of flaw reporting. Similarly, it was expected that programmers would be familiar with the concept of dynamic fault reporting and correction as used by the source code editor, from their successful attempts to enter spelling and grammatical errors into a Microsoft Word document. Whether the programmers would be damaged, in some way (eg. reduced morale concerning their programming abilities) by an automated tool informing them that they had entered an identifier name that deviates from the project identifier-naming style guidelines was unknown before the experiments were initiated. However, programmers are regularly subjected to a similar form of reporting during code reviews without lasting harm. It was expected that an automated tool would cause less harm, if any, as the social consequences, which are considered the most damaging (see section 2.2.4) are absent from the use of the tool.

During the maintenance and production experiment and the case studies, the participants were requested to complete the Programmer Characteristics questionnaire. The questions were concerned with identifying non-emotive characteristics of the test subject only. The initial test subjects who were employed to identify whether the source code editor and instructions were fit for purpose did not identify any moral issues with responding to the Programmer Characteristics questionnaire or to the dynamic reporting of identifier-naming style flaws. As the participants were not compelled to respond to the Programming Characteristics questionnaire and were assured of anonymity, in the case of the student test subject, and confidentiality, in the case of the professional programmers, it was not expected that any individual would feel anxiety in responding to the questionnaire.

The ethical implication of the research on the software engineering society has also been considered. The software engineering society has been categorised into two groups, for the purposes of this discussion i.e., those that create software (programmers) and those that manage the creation of software (managers). The managers will be able to use aspects of this research, in particular the Identifier-Naming Style Flaw Analysis and Reporting computer program, to see a quality aspect of software (i.e., adherence to the project identifier-naming style guidelines) more readily than by the more laborious task of reading the source code. The manager should

have no issue with accepting and using this capability as this information may enable them to identify and manage issues requiring staff development faster and easier than without this capability. However, the programmer may feel they are being judged for the purposes of performance appraisal. Where the programmer belongs to a large project team, the software engineering society has a right to police activities that can affect their society, even to the initial detriment of the individual. However, it is questionable whether the use of such a tool, applied to source code developed for small computer programs, where the follow on effects to the software engineering society is much reduced, would be acceptable to the individual programmer.

3.15 Chapter Summary

> *It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.*
> Franklin D. Rosevelt

In summary, the research method is characterised by the following:

- Statement of the foreground theory
- Statement and verification of the research hypothesis
- Identification and description of the variables that are required to support the research. The independent variable is varied during the maintenance and production experiment to control whether dynamic reporting of identifier-naming style flaws is either activated or deactivated. The dependent variable is measured and relates to whether an expert programmer perceives an identifier name to be meaningful or not.
- Various researchers have linked confounding variables to the production of readable source code. Hence the presence of specific values for the confounding variables could result in the skewing of results. The confounding variables are listed and are considered for their manifest effects on the readability of source code developed during the maintenance and production experiment.
- A Programmer Characteristics questionnaire is described, the intent of which is to collect programmer characteristics data from each of the test subjects who participated in the maintenance and production experiment and the case studies.
- An Identifier-Naming Style Guideline Programmer Acceptance questionnaire is described, the intent of which is to identify the attitudes of academics, programmers and software quality engineers to the identifier-naming style guidelines.

- Software quality metrics that could have the potential to measure source code readability are considered, but which are subsequently devalued by the necessity to use an expert programmer to adjudicate on source code readability. However their real worth will not become apparent until the results of the maintenance and production experiment, and case studies are evaluated.
- A source code editor that was used during the maintenance and production experiment and during the case studies is described. The source code editor was described in terms of its capabilities i.e., the user interface, in particular the method of dynamically reporting and subsequent correction of identifier-naming style flaws; the control afforded over the reporting of the identifier-naming style guidelines; a discussion regarding the design constraints appropriate to the source code editor; and a description of the session log.
- The Identifier-Naming Style Flaw Analysis and Reporting computer program necessary to support the survey of production software; and the analysis of data generated during the maintenance and production experiment and the case studies were discussed.
- The method of surveying programming texts was discussed.
- The method of surveying production software both contemporary and dated was discussed.
- The intent of the maintenance and production experiment was to conduct a controlled experiment and to look for any effect of the independent variable (i.e., whether dynamic reporting has been activated) on the dependent variable (i.e., whether the identifier name is meaningful). In addition any effect of the confounding variables is also to be considered during the analysis of the results.
- The intent of the case studies is to measure the effect of the dynamic reporting of identifier-naming style flaws over a longer period of time for both novice and experienced programmers. Task staffing, supporting documentation and the appropriate software development procedure were also discussed.
- The ethics of the research is considered in terms relevant to the participants and also to the software engineering society at large.

Taken collectively, these research methods will allow support for or rejection of the hypothesis stated at the beginning of this chapter. However, before the research method can be implemented the identifier-naming style guidelines require identification.

4 Identifier-Naming Style Guidelines Specification

This chapter shows the results of the literature review conducted in search of identifier-naming style guidelines. Having found an identifier name which is considered flawed by the application of an identifier-naming style guideline, in some cases, automatic correction of the identifier name can be applied. The relevant automatic correction to a flawed identifier name is similarly discussed in this chapter.

4.1 Literature Review Results

Fundamental to the research effort was the collection of identifier-naming style guidelines which could be used to evaluate identifier names against the software quality attribute of readability. As stated in section 3.4, these identifier-naming style guidelines would be found during the literature review. The following subsections present and discuss the relevant findings of the literature review. These findings have been grouped firstly by classes of identifier-naming style guidelines that are evaluated on a single identifier name, then by guidelines that are required to be evaluated against multiple identifier names and finally by the meaning implied by the natural language interpretation of words used to compose the identifier name.

4.1.1 Single Identifier – Character Relationships

Identifier-naming style guidelines which consider the relationship of characters that compose the identifier name are discussed below.

Weissman (1974) and Fang (2001) considers the use of 'magic numbers' (i.e., numbers where the value does not imply its meaning) should be declared as named constants to reduce the complexity of the source code and hence improve source code readability. The Software Product Evaluation – Quality Characteristics and Guidelines for their use, (ISO-9126:1991), in part, requires that all numeric constants except for the values: -1, 0 and 1, be identified by a named constant.

Guideline 01: Numeric literals should only be used in a constant declaration.

However, the literals -1, 0 and 1 may be regarded as exceptions to this guideline.

The use of a leading underscore character or the use of multiple contiguous underscore characters in an identifier name is discouraged as this arrangement has been reserved for standard C++ libraries and its use can result in confusion (Horton, 1998, p12). Similarly, Kernighan and Ritchie (1988, p35) recommend that a leading underscore character should not be used in the contraction of identifier names due to the potential for confusion (primarily since library routines often use this contract in the generation of global identifiers). There is some contention regarding the use of an underscore character in an identifier name. Wells (1986) states that one of his "pet peeves" is the inclusion of an underscore character in an identifier name. Conversely, Oualline (1992) states that "good variable names" are fabricated from words separated by the underscore character. However, neither Oualline nor Wells support their claims with a justification.

The research will show that the inconsistent use of a trailing underscore character in an identifier name can disrupt the successful completion of a maintenance exercise and also that an individual will show difficulty in distinguishing between identifiers composed of the same words in the same order but separated by differing numbers of underscore characters. Irrespective of the potential of the underscore character to separate words, the use of leading, trailing or multiple contiguous underscore characters will be flagged as an identifier-naming style flaw, as a programmer may have difficulty distinguishing between identifiers employing any one of these characteristics and hence this characteristic of an identifier name damages software readability. This issue is addressed further during the discussion that derives identifier-naming style guideline 17.

Guideline 02: Identifier names should not contain multiple contiguous underscore characters.

Guideline 03: Identifier names should not start or end with the underscore character.

McConnell (1993, p211) suggests that identifier names should not contain the numeric characters 0, 1, 2, 5 or 6 as the following pair groups: (0, O), (1, I), (1, l), (2, Z), (5, S) and (6, G) can be difficult to distinguish as they look similar when printed. More generally, McConnell (1993, p210) suggests that the use of numeric characters in an identifier name is inappropriate and damages software quality. Keller (1990) similarly suggests that identifier names should not contain digits. McConnel (1993, p210) suggests that if the intent is to define multiple identifiers, then an array data structure should be defined instead of the declaration of individual numerically enumerated identifiers

Guideline 04: Identifier names should not contain numeric characters.

4.1.2 Single Identifier – Character Count Relationships

Identifier-naming style guidelines which consider the number of characters that compose the identifier name are discussed below.

Mengel and Yerramilli (1999) found that identifier length is important in revealing source code understanding. DeYoung and Kampen (1979), Keller (1990) and Boundy (1991) suggest that in general, identifiers used over a short range within the source code should have short names and identifiers used over a greater range should have longer names but the researchers offer no indicative character length for identifier names used under these situations. Hence, implementation of a corresponding identifier-naming style guideline is not possible in this case.

Vivanco (2003) conducted research, using a generic algorithm, to find software quality characteristics which map closest to software maintainability as ranked by a system architect. The software quality characteristics, applicable to source code readability, identified by the best matching algorithm, was found to be the number of while-space lines in the project and the mean string length of method identifiers (Vivanco, 2003). Gorla, Benander and Benander

(1990) found that an average identifier name length of thirteen characters resulted in the lowest observed debug times for Cobol programmes. As understanding, which stems from an ability to read the source code, is required for the debugging process to progress it follows that the average number of characters in an identifier is important to source code readability. However, a requirement for an average identifier name length does not set bounds on the absolute length of an individual identifier name. McConnell (1993, p188) suggests that identifier names that are too short do not convey enough meaning and that identifier names that are too long are cumbersome to type and obscure the visual structure of the source code. McConnell (1993, pp81 & 188) cites research by Gorla, Benander and Benander (1990) suggesting that the average length of a variable identifier name should be 8 – 20 characters and additional research by Rees (1982) suggesting that the average length of a method identifier name should be 20 – 35 characters. Gorla, Benander and Benander (1990) discovered that an average variable identifier name of 10 – 16 characters was optimal for debugging Cobol programs by students and that a variable identifier name length of 8 – 20 characters was almost as effective for debugging purposes. Rees (1982) found that an identifier name length of 9 – 15 characters corresponded to a university examiner's concepts of good programming style. However, the work by Rees (1982) does not reference method identifier name length. The figure quoted by McConnell (1993, p81), for method identifier names, appears to be erroneous as it corresponds exactly to the value range reported by Rees (1982) for method SLOC length.

Rotenstreich (1988) suggests that the common practice of requiring student programmers to use words as loop variables (eg. Count, Index) should be discouraged in favour of single characters (i.e., i, j and k) particularly where the algorithm being implemented has a mathematical basis. Due to the prevalence of additional symbols such as l, m, n, t, x, y and z in mathematical and graphics algorithms, the list of single character names is extended to include these additional symbols. McConnell (1993, p190) qualifies further and suggests that simple Loop variable names, such as i, j and k, should only have scope within the Loop statement but accepts that this scope may be larger to accommodate variable initialisation. McConnell (1993, p190) asserts that if the Loop variable is used outside the Loop statement or the Loop statement stretches over many lines, a more meaningful name should be used to improve program readability. McConnel (1993, p190) does not state how many lines are appropriate before more meaningfully identifier names should be used for Loop counters. Hence, McConnell's assertion does not lend itself to implementation as an identifier-naming style guideline.

Guideline 05: Identifier names should be composed of no fewer than eight characters.

However, the algebraic variable identifier names (i.e., i, j, k, l, m, n, t, x, y and z) may be regarded as exceptions to this guideline.

Guideline 06: Identifier names should be composed of at most twenty characters.

4.1.3 Single Identifier – Word Count Relationships

Identifier-naming style guidelines which consider the number of words that compose the identifier name are discussed below.

Furnas et al. (1987) cite multiple empirical studies refuting the idea that there is an obvious and self-evident single word naming any function or object. These studies have found that we use a great variety of words to refer to the same thing and that any one individual's suggestion for a name will only rarely agree with the name chosen by another individual (Furnas et al., 1987). Hence, for an identifier name to be generally meaningful to other programmers, it should be composed of multiple words that combined are descriptive of the identifier's usage. Laitinen and Mukari (1992) suggest that the appropriate composition of an identifier, using their "natural naming" identifier-naming principals, is to use two to four words in the construction of the identifier name. However, this guideline would exclude the use of single character variable names and hence the guideline is qualified to accept the appropriate single character variable names identified in the previous section.

Guideline 07: Identifier names should be composed of two, three or four words.

However, the algebraic variable identifier names (i.e., i, j, k, l, m, n, t, x, y and z) may be regarded as exceptions to this guideline.

4.1.4 Single Identifier – Word Qualification Relationships

Identifier-naming style guidelines which consider word qualifications that are appended to form the identifier name are discussed below.

Hungarian identifier encoding, which is widely used by Microsoft Windows C computer language programmers, is useful for computer languages that are weakly-typed and in particular for the development of large computer systems (McConnell, 1993, pp205, 206 & 211). However, McConnell (1993, p206) also cites criticisms to the use of this notation which suggest that this practice should be avoided for the following reasons: (1) data type qualifiers are concatenated to the identifier name, which results in a name change when the base type changes; (2) the identifier names become uninformative as the name construction appears pre-occupied with the base type, one or more prefixes, a qualifier and finally the actual meaning of the data held by the identifier; and (3) the use of Hungarian notation generally encourages the declaration of uninformative identifier names. Full Hungarian notation qualification of an identifier name can result in a long identifier name, which contains a code identifying what the identifier is but still may not necessarily identify the intended meaning of the identifier. In addition, an emergent standard for Hungarian notation has not arisen, resulting in an inconsistent approach to using Hungarian notation to qualify identifiers. McConnel (1993, pp:203-205) cites twenty-nine Hungarian notation prefixes, where as, Hawkins (2003, p220) introduces three different Hungarian notation prefixes which differ from the previous list.

Guideline 08: Identifier names should not contain Hungarian notation identifier encoding.

Wu (1999, p43&44) describes the current Java identifier-naming convention. This identifier-naming convention singles out class identifier names for special treatment by the use of capitalisation rules so that they can be distinguished from all other identifiers. However, McConnell (1993, p198) and Vermeulen et al. (2000, p18) suggests that the practice of differentiating identifier names by capitalisation only, should be avoided, as the association of these identifier names to the appropriate meaning is arbitrary and confusing. Grogono (1979) and McConnell (1993, p198) observe that appending a type prefix (e.g., “type”) to a type identifier name, qualifies the identifier as a type and explicitly eliminates name confusion between type and variable identifiers. Ronsen (1995) also supports the qualification of special identifiers and proposes that the names of Ada 95 facet packages be appended with “Facet”.

Ronsen (1995) dissuades against appending "Class" to Ada 95 class packages as the source code becomes difficult to interpret as demonstrated by the example source code presented in Ronsen's argument.. McCall (2004) further asserts that type names should be unique within the source code.

Guideline 09:	**Class identifier names should be qualified by the word "class" concatenated to the end of the identifier name and type identifier names should be qualified by the word "type" concatenated to the end of the identifier name.**

Keller (1990) suggests that identifier qualifiers, such as "max" should be placed at the beginning of an identifier name. Conversely, McConnell (1993, p198) suggests that identifier qualifiers, such as "maximum" should be placed at the end of the identifier name and that this practice offers several advantages. Firstly, the most important part of the identifier name (i.e., the part that gives the most meaning) is read by the programmer first. Secondly, potential confusion caused by a random approach to appending a qualifier either to the front or to the end of the identifier name, is removed because a clear precedent has been defined. By appending a unique qualifier to the end of an identifier name, this condition can be supported. Researchers may not have published a list of these common identifier qualifiers, however a survey of production source code has identified the following words and their corresponding abbreviations as being common identifier qualifiers: "character", "index", "maximum", "number", "pointer", "record" and "string".

Guideline 10:	**Identifier names should have common qualifiers appended to the end of the identifier name.**

4.1.5 Single Identifier – Word Meaning Relationships

Identifier-naming style guidelines which consider the natural language meanings of words contained within an identifier name are discussed below.

McConnell (1993, p80) suggests that identifier names composed entirely of abstract or ambiguous words (e.g. Do_It, Calcuate_Value) do not constitute good identifier names as the name does not describe what the identifier is used for and only gives a vague idea of the intended use. The dictionary presents a list of abstract words which can be mined from its pages. As an exhaustive list of abstract words is not necessary to evaluate the effect on the source code of dynamically reporting every possible combination of abstract words forming identifier names, a modest set of abstract words have been identified and are listed in Appendix A – Identifier-Naming Style Guidelines. These abstract words were identified from the words used to form identifiers in production software.

Guideline 11: Identifier names should not be composed only of abstract words.

Laitinen and Mukari (1992) observe that identifier names have a tradition of being abbreviated and hence may be cryptic. This may be due to the unnatural shortening of a name imposed by early computer language compiler limitations (Schorsch, 1990, p16). This tradition remains popular today even though the limitations no longer exist. Keller (1990) suggests that the use of abbreviations in identifier names should be curtailed and that only common abbreviations such as "Id" should only be used. However, Laitinen and Mukari (1992) suggests that the ability of modern computer languages to allow long identifier names (i.e., 30 characters and more) has contributed to identifier names being more readable and understandable and that a natural approach to naming identifiers is now desirable. In support of this statement, Laitinen and Mukari (1992), and Laitinen (1996) found that the practice of using abbreviations in identifier names results in reduced software understandability and hence reduced maintainability. A common justification for the use of abbreviations in identifier names, made by programmers, is that the abbreviated name is easier to type. However, the savings in initial typing is small and will most likely be lost by the additional time required in deciding on the appropriate abbreviation but more importantly, time will be lost by future programmers who must interpret the meaning of the abbreviated identifier (Schorsch, 1990, p16).

McConnell (1993, p210) suggests that words in identifier names should not be miss-spelt as some people have difficulty in remembering the correct spelling of words and that expecting a programmer to remember an incorrect spelling adds unnecessary cognitive overhead. Laitinen and Mukari (1992) and Vermeulen et al. (2000, p17) also claim that the use of natural language words in identifier names results in increased software understandability. They base these

claims on results collected from experiments where subjects were required to respond verbally to questions designed to ascertain whether the subject understood the nature of the software, and also by asking subjects to demonstrate their ability to memorise and also to modify source code. Laitinen and Mukari (1992) found that usually, but not always with statistical significance, that the test subjects demonstrated greater understanding when the source code contained natural language identifier names than when abbreviations were used as identifier names. Caprile and Tonella (1999) further call for the standardisation of identifier names composed of words in the pursuit of source code understandability and maintainability.

The pursuit of readability in identifier names will require the use of acronyms. An application that controls a laser (Light Amplification by Stimulated Emission of Radiation) or an ATM (Automatic Teller Machine) would be cumbersome to read if the acronyms were expanded when generating the identifier names. Similarly, software projects can introduce acronyms unique to the specific project. Hence the failure to accept common or project specific acronyms would damage the readability of the source code by making identifier names cumbersome in length.

Guideline 12: Identifier names should not contain a word that cannot be found in the dictionary, or is not a common acronym or a project sanctioned acronym.

McConnell (1993, p196) suggests that constants should be named to identify the abstract entity and not the value that they represent. The identifier name "One_Hundred" is a poor name, regardless of whether the value 100 or some other value is assigned. A list of words that represent numeric values, the common numeric size multiplies (e.g., hundred, thousand) and the connecting word (i.e., and) are listed in Appendix A – Identifier-Naming Style Guidelines. In addition, the words representing numeric values include both forms of counting in the English language (i.e., one, two, three, … and fist, second, third, …) have also been included in the word list.

Guideline 13: Identifier names should not be composed only of words used to name numeric values.

Vermeulen et al. (2000, p21 & 26) suggest that classes that group related attributes, instances of these classes and array declarations should be pluralised. However, the complexity of this statement presents opportunity for incorrect and inconsistent application. Meyer (1988) similarly notes that there is also the potential for confusion where one identifier differs from another by only one character e.g. “count” and “counts”. Not all words have plurals that are spelt differently to the singular form of the word and some words do not have plurals. As such, the use of the plural form within an identifier name should be discouraged in preference to using one form over both forms. This practice will reduce confusion regarding the spelling of an identifier name. As there is no simple rule that allows an English word to be identified as the plural form of the word, a lookup table is required to identify plural words. Such a function can be supported by the use of comprehensive dictionaries, which allow for the identification of plural words.

Guideline 14: Identifier names should contain words only in the singular form.

4.1.6 Single Identifier – Naming Convention

Identifier-naming style guidelines which consider the identifier naming convention have are discussed below.

Meyer (1988) notes that there is potential for confusion by the programmer in the correct interpretation of an identifier name, particularly if the programmer is unfamiliar with the computer language identifier-naming syntax rules. McConnell (1993, p210) further suggests that identifier names should not be differentiated by capitalisation alone as identifier names will, only be unique for some computer languages (e.g. Java) and that this use of capitalisation can be confusing to programmers unfamiliar with the identifier-naming syntax rules. Spinellis (2003) asserts that the inconsistent capitalisation of acronyms and the inconsistent splitting of words by either capitalisation rules or the use of a separation character will make identifier names harder to remember. Two common techniques for increasing source code readability are the consistent use of capitalisation and the consistent use of spacing characters (eg. the underscore character) to separate words (McConnell, 1993, p199).

Booch (1994, p164) suggests that the naming convention is a matter of personal taste but a consistent naming convention should be used throughout the program source code to improve source code readability. Naming conventions have been established for common computer programming languages. For Ada a common naming convention is that the first letter of a word is capitalised and words are separated by the underscore character (Johnston, 1997, p9). For Java a common naming convention is that every letter in an identifier constant declaration is in uppercase and words are separated by the underscore character. Class names and constructors have the first letter of each word capitalised which are concatenated together without the use of the underscore character. All other identifier names have the first letter of composite words capitalised, except the first word in the identifier name and the words are concatenated together without the use of the underscore character (Wu, 1999, pp43 & 98). However, neither of these naming conventions properly considers the readability of the resultant identifier name.

Matis (1996) reviews the literature relevant to text readability and reports that lowercase word phrases are read 10% faster and that uppercase word phrases. Matis (1996) also reports that word phrases, where the first letter of each word is capitalised, are read 19% faster than uppercase word phrases. However, using uppercase word phrases does allow these word phrases to be located faster within the text of a document (Matis, 1996). This advantage is negated by the search capability common to text editors and any potential advantage is hence effectively eliminated. This finding implies that identifier names composed of words where the first letter only is capitalised and with the words separated by some separation character, will be read faster than identifier names containing either lowercase characters only or uppercase characters only. Hence, the current Java naming convention actually reduces readability of source code over the practice of defining identifier names written with the first letter of the word capitalised and with words separated by say an underscore character. However, as there are many computer language naming conventions used by programmers, who can be quite emotionally attached to specific naming conventions, a number of naming conventions have been supplied for selection by the programmer. The selections available are listed in Appendix A – Identifier-Naming Style Guideline.

Guideline 15: Identifier names should not deviate from the project selected identifier-naming convention.

4.1.7 Multiple Identifier Relationships

Identifier-naming style guidelines which consider the relationships between two identifiers, where one identifier is declared within the same program scope as the other are discussed below.

Weissman (1974) considers the practice of overloading identifier names to be confusing to the programmer. Further to this, Deimel and Naveda (1990) warn that this practice should be discouraged in student programmes, in the interests of source code readability but offer no justification for this statement. Chan and Yang (2002) note that the practice of substituting of meaningless identifier names into Java Byte code will complicate understanding of the decompiled source code but that this practice does not deter a determined individual from cracking the source code. Chan and Yang (2002) also discovered that the over-use of the same identifier name will often result in software cracking taking longer than the time initially required to develop the software, particularly when the identifiers are of different kinds (i.e., class, constant, variable). This result implies that readability is impaired with the declaration of multiple identifiers within the same program scope that have the same name.

Guideline 16: Identical identifier names that are different in kind should not be declared within the same program scope.

Deimel and Naveda (1990), McConnell (1993, p209) and McCall (2004) suggest that identifier names differing by two characters or less have the potential for confusion. Ledgard, Hueras and Nagin (1979, p125) support McConnell by listing a collection of identifier name pairs and labels the pairs where differ by one character only, as examples of poor identifier naming practice.

Guideline 17: Identifier names differing by one or two characters should not be declared within the same program scope.

Wells, Brad and Markosian (1995) found that source code that contained unused identifiers was usually of poor quality. The converse is not necessarily true but leaving unused identifiers in

the source code has the potential for confusion during maintenance and hence reduces source code readability as a consequence.

As an aside, if the programmer composed their source code sequentially, the dynamic reporting of identifier-naming style flaws would initially highlight the identifier name in the identifier declarations as being flawed, as the source code was entered. The highlighting would remain until the programmer progressed to the point of writing source code that used the identifier and at this point the highlighting of the identifier name would disappear, conditional that there were no other identifier-naming style flaws present in the identifier name. Such a situation is not considered relevant as typically the programmer would understand the reason for this initial reporting. Hence, the reporting of an unused identifier only becomes of relevance to the programmer when the source code, at one time, did use the identifier but that the programmer has now deleted all references to the identifier, save for the identifier declaration. Under such a situation, the reporting of an unused identifier would be relevant for the programmer.

Guideline 18: Identifiers that are declared but not used should have the declaration removed.

McConnell (1993, p209) suggests that the use of different identifiers that are composed of the same words but in a different order e.g. “File_Index” and “Index_File”, has the potential for confusion.

Guideline 19: Identifier names that are composed of the same words but occurring in a different order should not be declared within the same program scope.

4.1.8 Identifier Name Natural Language Meaning

Identifier-naming style guidelines which consider the natural language meaning of the words composing the identifier are discussed below. Laitinen and Mukari (1992) argue that software

maintenance is simplified when the words forming the identifier name corresponds exactly with the name used in the application domain and software documentation. For example, the concept "Book Library Number" would be represented by the identifier name: "Book_Library_Number". They observe that maintainers can use documentation to suggest the search string to be applied to the source code listing and the programmer can then be reasonably assured that an electronic search will find the appropriate place in the source code (Laitinen and Mukari, 1992). Sommerville (1995, p256) has surveyed the identifier-naming principals, suggested by a number of researchers and has catalogued them broadly as:

- Identification of objects (i.e., tangible things) from the application domain.
- Grammatical analysis applied to a natural language description of the computer system's functions to identify the objects.
- Scenario-based analysis used to identify the objects.
- Identification of the system behaviour and participants established.

These principles typically make reference to programming objects (procedures, variables etc.) and suggest which grammatical entities (nouns, verbs etc.) should be used to construct the programming object's identifier name (Sommerville, 1995, p256). Spinellis (2003) notes that the Windows Application Programming Interface defines function names that consist of a group name, a verb and an object component names but that these component names fall in no particular order. Spinellis (1998) laments that this practice makes identifier names harder to remember. Unfortunately, Spinellis (2003) does not suggest a standard ordering that would aid in source code readability. However, the automation of identifier-naming style guidelines based on these principles is problematic at best and so no identifier-naming style guidelines are suggested.

The automation of identifier-naming style guidelines based on the natural language meaning of words used to compose the identifier name is problematic. Particular issues associated with the evaluation of these identifier-naming style guidelines include the ambiguousness of words which requires access to context information which can be missing from the source code. As the research is concerned with the dynamic reporting of identifier-naming style flaws during editing of source code any actions necessary to be conducted external to the source code are deemed to be outside the scope of the current research.

4.2 Identifier-Naming Style Guideline Implementation

The identifier-naming style guidelines were implemented within the functionality of a source code editor. The intent of the source code editor was to facilitate the research, hence a high level of fidelity was not required to implement the identifier-naming style guidelines, as the experiment qualified the necessary fidelity of the implementation. The implementation limitations of the identifier-naming style guidelines are listed below:

Identifier-Encoding (08) – Hungarian notation is not specified by a defined and accepted standard. In addition to the Hungarian notation, there are other encoding philosophies used by programmers. Documented identifier encodings were captured from McConnell (1993, pp 203-205), however the 143 encodings found may not necessarily represent a complete set of identifier encodings used by programmers.

Constant/Variable Qualification (10) – Eight identifier qualifications that are commonly appended to identifier names and only the standard abbreviations of these qualifiers were implemented. This list is incomplete and the choice to only implement the standard abbreviations similarly limits the implementation.

Abstract Words (11) – 47 abstract words corresponding to those commonly found in production source code were implemented. This list is incomplete.

Numeric Name (13) – 68 words that can be used to construct a number where implemented. This list is potentially incomplete.

Plural Word (14) – The algorithm used to identify a word in the plural form marked any word ending with the character "s" as potentially in the plural form. A lookup table containing twenty-eight words, that are common exceptions, was used to correct the result for these word instances. This list of common exception words is incomplete.

4.3 Suggested Replacements

Those identifier-naming style flaws that can be automatically corrected are discussed below. Those flaws where no automatic correction is possible are summarily stated. The purpose of this discussion is to identify what suggested corrections to an identifier name can be automatically presented when an identifier-naming style flaw is detected.

Un-named Constant (01)

This identifier-naming style flaw is raised when the source code contains a numeric literal which is within an expression that is not part of an identifier declaration. By replacing the occurrence of the numeric literal with an identifier name and defining the

identifier as a named constant, the identifier naming style flaw can be removed from the source code. However, as the value of the numeric literal potentially holds no suggestion as to the intended meaning, no assistance can be given to the programmer by suggesting a possible replacement identifier name and the programmer is hence required to suggest a name for the numeric literal.

Once the programmer has selected an identifier name, an appropriate constant declaration can be automatically inserted into the source code and instances of the numeric literal can be replaced by the identifier name that was entered by the programmer. The constant declaration is required to be inserted before the first instance of the numeric literal. This positioning may be required to support the computer language compiler's requirements for identifier name visibility. However, the program scope of the named constant should be minimal due to its potential to hide another identifier declaration that could potentially use the same identifier name. Hence the automatically generated constant declaration should be positioned at the same program scope as the original numeric literal.

As the constant declaration program scope could conceivably encompass further instances of the same numeric literal, the replacement of these numeric literals with the new identifier name is also possible. However, the decision to effect this global replacement within the current program scope can only be an arbitrary decision as a second occurrence of the same numeric literal may not necessarily have the same program meaning as the first occurrence of the numeric literal. However, this action does not modify the execution of the source code and may potentially highlight, to the programmer, that further action on their part is required due to the inappropriate substitution.

Multiple Underscore (02)

This identifier-naming style flaw is raised when the subject identifier name, of an identifier declaration, contains at least two contiguous underscore characters. By removing the occurrence of the second and all subsequent adjacent underscore characters in the identifier name, a suggested replacement identifier name can be constructed. By applying the procedure recursively, all instances of multiple contiguous underscore characters can be removed.

Outside Underscore (03)

This identifier-naming style flaw is raised when the subject identifier name, of an identifier declaration, contains a leading, trailing or both a leading and a trailing underscore character. By removing the occurrence of the relevant underscore character(s) in the identifier name, a suggested replacement identifier name can be constructed. By applying the procedure recursively all instances of multiple leading and trailing underscore characters can be removed.

Numeric Digit(s) (04)

This identifier-naming style flaw is raised when the subject identifier name, of an identifier declaration, contains numeric characters. By removing the occurrence of all numeric characters i.e., "0" through "9" within the identifier name, a suggested replacement identifier name can be constructed. In addition, the Multiple Underscore (02) procedure should then be called to remove any contiguous multiple underscore characters resultant from the removal of a sequence of numeric characters that were separated by the underscore character.

Short Name (05)

No specific action is possible, other than to allow the programmer to enter a replacement identifier name.

Long Name (06)

No specific action is possible, other than to allow the programmer to enter a replacement identifier name.

Word Count (07)

No specific action is possible, other than to allow the programmer to enter a replacement identifier name.

Identifier Encoding (08)

This identifier-naming style flaw is raised when the subject identifier name, of an identifier declaration, is qualified by encoding information. This identifier-naming style flaw can be removed by removing the occurrence of all encoding characters. Appendix A – Identifier-Naming Style Guidelines contains a list of the encoding qualifications used to support this procedure.

Class/Type Qualification (09)

This guideline is raised as an identifier-naming style flaw when the subject identifier name, of an identifier declaration, is not qualified as appropriate for a class name or for a type name. The identifier-naming style flaw can be removed by concatenating the appropriate qualification character string to the end of the identifier name. The appropriate qualification character string depends on the identifier kind, i.e., "class" is concatenated to the end of Class names, and "type" is concatenated to the end of Type and Subtype names.

Constant/Variable Qualification (10)

This identifier-naming style flaw is raised when the subject identifier name, of an identifier declaration, is not qualified as appropriate for a constant name or a variable name. The identifier-naming style flaw can be removed by shifting the qualification word from the current location within the identifier name to the right of the identifier name. In addition, qualification word abbreviations that are detected in the identifier name can be replaced by the corresponding English word and similarly moved. Appendix A – Identifier-Naming Style Guidelines contains a list of the qualification words and their common abbreviations that the source code editor is sensitive to.

As the definition of an identifier name containing multiple occurrences of the same qualification is possible, the duplication of the qualifications at the right of the identifier name may not offer any additional understanding. Hence, any duplication of the same qualifications will be removed in the suggested identifier name.

Abstract Words (11)

No specific action is possible, other than to allow the programmer to enter a replacement identifier name.

English Words (12)

No specific action is possible, other than to allow the programmer to enter a replacement identifier name.

Numeric Name (13)

No specific action is possible, other than to allow the programmer to enter a replacement identifier name.

Plural Word (14)

This identifier-naming style flaw is intended to be raised when the subject identifier-name, of an identifier declaration, contains a word in the plural form of the word that is not also the singular form of the same word. The identifier-naming style flaw can be removed by replacing the plural form of the word with the corresponding singular form of the word within the identifier name. Appendix A – Identifier-Naming Style Guidelines describes the characteristics of identifier names that result in their being identified as containing plural words. The procedure used to support the generation of a suggested identifier name is to remove the last character from each word forming the identifier name, where that character corresponded to the character "s" or "S" and where the resultant word continues to exist as an English word.

Acknowledgement is made that this procedure is incomplete and that a complete implementation of the required functionality was not undertaken for the generation of the research tool. Hence, the results from the maintenance and production experiment were checked to identify any impact that the incomplete implementation of the algorithm may have had on the test subject. No impact was found as the exercise was extensively constrained to remove the possibility of an issue.

Naming Convention (15)

This identifier-naming style flaw is raised when the subject identifier name, of an identifier declaration, contains word(s) that deviate from the selected project identifier-naming convention. The identifier name can be decomposed into a list of words using clues such as the presence of an underscore character and the change in character case from lowercase which is proceeded by an uppercase character. Once the word list is found, the identifier name can be formatted as required for the selected project identifier-naming convention.

Acknowledgement is made that this procedure is incomplete and that a complete implementation of the required functionality was not undertaken for the generation of the research tool. Hence, the results from the maintenance and production experiment were checked to identify any impact that the incomplete implementation of the algorithm may have had on the test subject. No impact was found as the exercise was extensively constrained to remove the possibility of an issue.

Duplicate Names (16)

No specific action is possible, other than to allow the programmer to enter a replacement identifier name.

Similar Names (17)

No specific action is possible, other than to allow the programmer to enter a replacement identifier name.

Unused Identifier (18)

This identifier-naming style flaw is raised when the subject identifier name, of an identifier declaration, is not used within the program scope of the identifier declaration. The identifier-naming style flaw can be removed by deletion of the identifier declaration. Programmer confirmation of this activity is requested before the automatic deletion of the identifier declaration is undertaken.

Same Words (19)

No specific action is possible, other than to allow the programmer to enter a replacement identifier name.

On construction of the new identifier name using any of the procedures discussed above, the project identifier style is applied to generate the suggested identifier name.

The application of a procedure for one class of identifier-naming style flaw could unintentially introduce a different class of flaw into the identifier name. For instance, the suggested replacement for an identifier name that raised the Multiple Underscore (02) identifier-naming style flaw could reduce the length of the identifier name by the removal of specific instances of the underscore character such that the Short Identifier (05) identifier-naming style flaw is now raised where it was not previously raised on the original identifier name. These procedures do not purport to be complete and the assistance of a programmer may be required to further edit the suggested identifier name in order to generate a meaningful identifier name. Where no specific action is possible, replacement of the identifier name by a more meaningful identifier name would require an understanding of the relevant application. Such an understanding is outside the scope of this dissertation and is left to the programmer to make a determination for the appropriate replacement of the identifier name.

4.4 Chapter Summary

> *Names are an important key to what a society values. Anthropologists recognize naming as 'one of the chief methods for imposing order on perception.'*
> David S. Slawson

In summary, the identifier-naming style guidelines specification is characterised by the following:

- Identifier-naming style guidelines that could be automated and only requiring source code as input have been found within the literature and have been specified in the form of a guideline.
- Of the nineteen identifier-naming style guidelines found, only four are supported by empirical data, with the remainder supported only by researcher's assertions that the guideline detects identifier-naming style flaws that affect source code readability.
- Of the nineteen identifier-naming style guidelines, ten procedures have been described which when employed will result in the corresponding flaw being removed from the identifier name.

5 Investigation & Results

In a previous chapter the research method, appropriate for this thesis, was defined. In this chapter the results from applying the research method are presented. In particular the results of a questionnaire, used with the intent of capturing programmer attitudes specific to the identifier-naming style guidelines, are reported. In addition, dated programming texts are analysed for instances of identifier-naming style flaws. The research problem investigation continues with the collection of contemporary source code from multiple sources and dated source code generated by the same individual over a protracted period of time. The source code collected is analysed for instances of identifier-naming style flaws. As the source code analysis survey found identifier-naming style flaws that survived into production, a maintenance and production experiment is conducted to identify the distribution of identifier-naming style flaws that persist through to completion of common software units by groups of test subjects. However these results could not be considered indicative of an extended exposure of the programmer to the reporting of identifier-naming style flaws and so case studies are conducted to identify the effects on the source code resulting from a longer exposure to the reporting of identifier-naming style flaws. A small case study is conducted using a novice programmer and a larger case study conducted using a number of experienced programmers. The results collected during these activities are described in the following sub-sections.

5.1 Identifier-Naming Style Guideline Programmer Acceptance Questionnaire

The purpose of the Identifier-Naming Style Guideline Programmer Acceptance questionnaire is to facilitate the collection of software professionals' attitudes that are relevant to their acceptance of the identifier-naming style guidelines. The Identifier-Naming Style Guideline Programmer Acceptance questionnaire was administered, with 34 individuals responding, consisting of seven novice, twelve intermediate, seven expert programmers, two academics and five software quality assurance engineers.

5.1.1 Attitude Statement Acceptance

An attitude questionnaire records attitudes of the respondents but as discussed in the Research Method chapter, if any attitude statement attracts a Pearson correlation value of less than 0.85

that attitude statement should be discarded from further consideration as the particular attitude statement is not considered relevant by the population sample.

Table 5-1 reports the Pearson correlation values for each of the identifier-naming style guidelines. The Pearson correlation values have been calculated for the questionnaire responses received from the novice programmers, intermediate programmers and the expert programmers. A value appearing on a shaded background (red) indicates that the corresponding identifier-naming style guidelines are not accepted as necessarily being relevant to source code readability.

Table 5-1 - Attitude Statement Pearson Correlations

Id	Identifier-Naming Style Guideline Name	Pearson Correlation		
		Novice	Intermediate	Expert
01	Un-named Constant	0.81	0.94	0.94
02	Multiple Underscore	0.94	0.95	0.96
03	Outside Underscore	0.93	0.91	0.88
04	Numeric Digit(s)	0.89	0.88	0.88
05	Short Name	0.96	0.94	0.92
06	Long Name	0.96	0.94	0.89
07	Word Count	0.85	0.81	0.91
08	Identifier Encoding	0.89	0.91	0.96
09	Class/Type Qualification	0.96	0.94	0.97
10	Constant/Variable Qualification	0.85	0.85	0.94
11	Abstract Words	0.98	0.91	0.93
12	English Words	0.88	0.92	0.95
13	Numeric Name	0.84	0.93	0.94
14	Plural Word	0.88	0.97	0.96
15	Naming Convention	0.92	0.94	0.91
16	Duplicate Names	0.92	0.85	0.97
17	Similar Names	0.83	0.92	0.94
18	Unused Identifier	0.90	0.93	0.98
19	Same Words	0.85	0.87	0.86

Table 5-1 indicates that the expert programmers believe, for the example identifier names used in the Identifier-Naming Style Guideline Programmer Acceptance questionnaire, that the representative identifier-naming style guidelines relate to source code readability, either for or

against, in all instances. The intermediate programmers believe that the Word Count (07) identifier-naming style guideline may not necessarily relate to source code readability. Where as, the novice programmers believe that the Un-named Constant (01), Numeric Name (13) and the Similar Names (17) identifier-naming style guidelines may not necessarily relate to source code readability. A progression in beliefs would be expected if the transition from novice through intermediate to expert programmer occurs with the assimilation of more knowledge regarding, amongst other things, the importance of identifier-naming style and in particular the readability of identifier names. However, the transition is not a simple adoption of attitudes that mimic those shared by the expert programmer. This aspect will be returned to in the next section and again when the confounding variables are addressed during the analysis of the maintenance and production experiment data.

The above result identifies a difference in attitudes between novice, intermediate and expert programmers, which would be expected, but does not indicate whether the questionnaire respondents agree that the identifier-naming style guidelines direct towards improved source code readability or away from improved source code readability.

5.1.2 Attitude Statement Agreement

The Identifier-Naming Style Guideline Programmer Acceptance questionnaire coding mean was calculated for each identifier-naming style guideline. The questionnaire coding used a five-point Likert scale: with a value of one corresponding to the highest acceptance of the particular identifier-naming style guideline.

Table 5-2 reports for each of the identifier-naming style guidelines, the sample population mean questionnaire coding and also separately the novice, intermediate and expert programmers mean questionnaire coding. The statistical significance of the expert programmer's mean score is represented as a probability value, calculated by a Student's t test over the sample population. A value appearing on a shaded background (red) corresponds to a mean value for which the questionnaire respondents did not generally accept the identifier-naming style guideline as directing towards improved source code readability for the example identifier names presented in the questionnaire statements.

Table 5-2 - Identifier-Naming Style Guideline Programmer Acceptance Questionnaire Coding Means

Id	Identifier-Naming Style Guideline Name	Questionnaire Score Means				Statistical Significance
		Sample	Novice	Inter-mediate	Expert	
01	Un-named Constant	2.1	1.9	1.8	2.4	0.102
02	Multiple Underscore	2.0	1.4	2.3	2.1	0.171
03	Outside Underscore	2.9	3.3	2.7	2.4	0.051
04	Numeric Digit(s)	2.5	3.0	2.7	1.5	<0.001
05	Short Name	3.1	3.3	3.3	2.9	0.023
06	Long Name	3.2	3.4	3.8	2.6	0.102
07	Word Count	1.9	1.6	2.0	1.7	0.182
08	Identifier Encoding	2.4	2.0	2.4	2.3	0.039
09	Class/Type Qualification	3.4	3.9	3.3	3.7	0.001
10	Constant/Variable Qualification	1.8	1.9	2.3	1.3	<0.001
11	Abstract Words	2.4	2.5	2.4	2.1	0.039
12	English Words	1.8	1.8	2.2	1.6	<0.001
13	Numeric Name	1.7	2.1	1.6	1.3	<0.001
14	Plural Word	2.0	1.6	2.3	2.0	0.182
15	Naming Convention	2.5	2.1	2.8	2.0	0.178
16	Duplicate Names	2.1	2.3	2.2	1.7	<0.001
17	Similar Names	1.6	2.0	1.8	1.4	<0.001
18	Unused Identifier	2.8	2.8	3.2	2.7	0.013
19	Same Words	2.2	2.3	2.7	1.7	0.178

Table 5-2 indicates that the expert programmers generally accept that the identifier-naming style guidelines are relevant to directing towards improved source code readability for the example identifier names presented in the Identifier-Naming Style Guideline Programmer Acceptance questionnaire for all but the Class/Type Qualification (09) guideline. However, the previous statement is true only for the Java computer programming language as none of the expert programmers chose to respond to the Ada computer programming language translation of the questionnaire statements.

One other point of interest to note is that there would be an expectation that intermediate programmers would transition between he attitudes of the novice programmers and the expert

programmers as they represent a transition phase in experience between the novice and expert programmers. This statement appears to hold for seven of the nineteen identifier-naming style guidelines. However, there remain ten identifier-naming style guidelines where the intermediate programmers showed a lesser acceptance of the guideline than both the novice and expert programmers, and potentially indicate that intermediate programmers adopt poor programming habits that they eventually lose on their path to becoming an expert programmer. This observation is addressed further during the discussion of the maintenance and production experiment results.

Table 5-3 reports the identifier-naming style guidelines that are not generally accepted by the software professional groups that participated in the Identifier-Naming Style Guideline Programmer Acceptance questionnaire. For each of the software professional groups, the identifier-naming style guideline Id is tabled where the guideline is not generally accepted by the corresponding software professional groups (i.e., where a mean questionnaire coding value of greater than 3 is reported against the guideline).

Table 5-3 - Non-Accepted Identifier-Naming Style Guideline Id

Software Professional Groups	Non-Accepted Identifier-Naming Style Guideline Id				
Novice	03	05	06	09	
Intermediate		05	06	09	18
Expert				09	
Academic				09	

As expected, with an increase in programming experience, more of the identifier-naming style guidelines are accepted by the programmer. The results suggest that attention to different identifier-naming style guidelines would be important, depending on the development stage of the programmer. In particular, the reporting of Outside Underscore (03), Short Name (05), and Long Name (06) may be particularly more important to novice programmers. It is important to note that the Class/Type Qualification identifier-naming style guideline should not be reported as the guideline has been rejected by the expert programmer as not necessarily directing towards improved source code readability. However, merely because a programmer acknowledges that an identifier-naming style guideline directs towards improved source code readability, does not necessarily mandate that their actions will result in the production of source code that is extensively free of the corresponding identifier-naming style flaws. The identifier-naming style flaws actually produced by novice programmers are further investigated with the investigation of the maintenance and production experiment and with the investigation of the case studies.

5.1.3 Novice/Expert Comparisons

With the exception of the identifier-naming style guidelines: Un-named Constant (01), Multiple Underscore (02), Word Count (07), Identifier Encoding (08) and Plural Word (14), the expert programmer shows a stronger or equal general acceptance of the guideline compared to that of the novice. As the expectation was that the expert, by virtue of greater experience, should have demonstrated stronger or at least equal acceptance of all the identifier-naming style guidelines past that of the novice, either these particular guidelines are not supportive of improved source code readability or there was a flaw in the questionnaire. Hence further investigation was conducted to ascertain the nature of the discrepancy against the expected result.

The Identifier-Naming Style Guideline Program Acceptance questionnaire results for the Un-named Constant (01), Word Count (07), Identifier Encoding (08) and Plural Word (14) identifier-naming style guidelines were highly polarised by the novice programmer's tendency to support the questionnaire statement by selecting on a response that scored for the value 1. Conversely, the expert programmer's preference was to support the questionnaire statement by selecting on a response that scored for the value 2. However, this polarisation, in the support of the identifier-naming style guidelines, was responsible for the mean scores calculated and does not represent instances of disagreement. By reducing the questionnaire coding from five values to three values (i.e., non-acceptance, neutral and acceptance) the results of the novice and expert programmers align for the identifier-naming style guidelines.

All of the questionnaire respondents who responded with acceptance of the Multiple Underscore (02) identifier-naming style guideline were contacted and a direct verbal enquiry requesting clarification of their response to the corresponding questionnaire statement was made. In all instances an emphatic response was given by the questionnaire respondents that they did not accept the use of multiple contiguous underscores in the construction of identifier names. When the questionnaire respondents were shown the example method skeletons, from the relevant questionnaire statement, the questionnaire respondent's unanimous response was to state that they did not notice the double underscore in the identifier name until its existence was brought to their attention. One questionnaire respondent commented that the question's presentation actually demonstrates the difficulty in distinguishing between identifier names that differ in this way and that this further demonstrates the detrimental readability issue with the practice of using multiple contiguous underscores in an identifier name. Hence, in this instance the questionnaire was flawed in that the method skeletons did not allow the questionnaire respondents, in all cases, to readily distinguish between the two identifier name declarations. However, the questionnaire did indirectly succeed in establishing the reading difficulty that

software professionals have in correctly distinguishing identifier names containing multiple contiguous underscore characters used as part of the identifier name.

5.2 Textbook Survey

While the dated software was being developed well before this research was being conducted (see next section) the author was exposed to various programming textbooks which possibly influenced the development of the source code. These programming textbooks had all been used in Australian universities or had been recommended by corporate training companies affiliated with a university. Table 5-4 identifies these textbooks by listing the year of publication, the relevant computer programming language and the author's name(s). This information is presented as it is sufficient to identify the specific textbook from the bibliography.

Table 5-4 - Computer Programming Language Course Textbooks

Year	Computer Programming Language	Author
1972	Fortran	McCracken
1978	Pascal	Jensen & Wirth
1982	Modular-2	Wirth
1988	C	Kernigham & Ritchie
1989	Ada	Barnes
1998	C++	Horton
2000	Java 2	Roberts, Heller & Ernest

A scan of these textbooks has identified occurrences where the identifier-naming style guideline is explicitly supported by a statement of good programming practice in support of the guideline; where the text passively supports the guideline in that no deviation from the guideline is present in the text; and where the text passively detracts from a guideline by including source code listings that contain examples of identifier-naming style flaws. None of the texts explicitly contradicted any of the identifier-naming style guidelines.

Table 5-5 reports the results of the textbook survey. The numbers in the table correspond to the first page number where support for or against the identifier-naming style guideline appears. Explicit support for an identifier-naming style guideline is indicated by a page number appearing on a shaded background (green), passive support for a guideline is indicated by a

hyphen and passive detraction from a guideline is indicated by a page number on an unshaded background. The appearance of "N/A" in the table is an indication that the relevant identifier-naming style guideline is not applicable for the relevant computer programming language, as the identifier naming syntax does not allow for the generation of a valid identifier that would result in a corresponding identifier-naming style flaw being raised (eg. Fortran IV did not support the occurrence of the underscore character in an identifier name). In addition, the first row of the table reports the first page number where an ambiguous or meaningless identifier name was used in an identifier declaration within the text.

Table 5-5 - Course Textbook Page Numbers

Id	Identifier-Naming Guideline Name	Fortran 1972	Pascal 1978	Modular-2 1982	C 1988	Ada 1980	C++ 1998	Java 2 2000
-	Ambiguous or Meaningless	24	3	8	9	11	45	5
01	Un-named Constant	23	3	8	9	84	59	9
02	Multiple Underscore	N/A	N/A	N/A	-	-	-	-
03	Outside Underscore	N/A	N/A	N/A	35	-	44	6
04	Numeric Digit(s)	53	3	58	128	88	61	78
05	Short Name	23	3	8	6	13	44	12
06	Long Name	-	10	-	-	20	-	383
07	Word Count	37	10	38	29	13	44	15
08	Identifier Encoding	-	-	-	-	-	-	-
09	Class/Type Qualification	N/A	34	62	146	45	98	21
10	Constant/Variable Qualification	-	37	41	38	137	194	-
11	Abstract Words	31	34	90	-	-	50	21
12	English Words	23	3	23	9	13	44	12
13	Numeric Name	-	-	-	-	151	-	-
14	Plural Word	53	34	-	39	155	184	21
15	Naming Convention	N/A	34	19	-	-	44	53
16	Duplicate Names	-	68	47	-	21	323	85
17	Similar Names	86	68	41	39	89	168	90
18	Unused Identifier	152	82	26	84	17	107	79
19	Same Words	-	-	-	-	-	350	-

As can be readily seen from the table, only one textbook provides explicit support for one of the identifier-naming style guidelines (i.e., Outside Underscore (03)). With the exception of the Multiple Underscore (02) and Identifier Encoding (08) identifier-naming style guidelines, at least one of the textbooks presents an example identifier declaration that detracts from each of the remaining identifier-naming style guidelines. On average one need only to turn through fourteen pages of one of the textbooks surveyed by the research, before the reader encounters identifier declarations that deviate from at least one of the identifier-naming style guidelines or alternatively turn through fifteen pages on average before the reader encounters an ambiguous or meaningless identifier name. If this practice is endemic in programming textbooks, it would appear that the student programmer cannot appeal to their programming textbooks to learn good identifier-naming practices and that they must discover good identifier-naming practices either from their teachers, colleagues and/or from their programming experience.

Hence, due to the proliferation of poor identifier-naming style practices encountered in this otherwise unremarkable set of historic and contemporary textbooks commonly used by university software engineering courses, it is unlikely that programmers learn or are encouraged to learn good identifier-naming style practices from their course textbooks. As the dated software was developed under suffrage of textbooks that indirectly advocate poor identifier-naming practices, extensively through poor example, any reduction in identifier-naming style flaws would be expected to have occurred due more to experiential learning than to instruction born out of programming textbooks. This issue is addressed further in section 5.4.

5.3 Contemporary Software Survey

Having established the attitudes of professional programmers to the identifier-naming style guidelines, by the use of the Identifier-Naming Style Guideline Programmer Acceptance questionnaire, attention was then focussed on the extent to which these guidelines were being used by production programmers. Contemporary production software was collected from open source software projects via an Internet download. Source code was collected from both closed source and open source, sources. Raghunathan et al. (2005) observes that there are many critics who state that open source software quality is inferior to that of closed source software and conversely that there are proponents who argue the opposite. Raghunathan et al. (2005) found that there were no substantial quality characteristics that were not shared by both closed source and open source software. Hence, offering the potential for the meaningful comparison of software quality characteristics of closed source and open source software.

The software projects collected were diverse in their application and included Interactive Development Environments, mathematical algorithms, data analysis applications and source code analysis applications. In total 359k SLOC of Ada, composed of 37 software projects ranging in size from 81 SLOC to 71k SLOC, and 354k SLOC of Java, composed of 12 software projects ranging in size from 1.3k SLOC to 96k SLOC was collected. In addition, two software projects, composed of 5k SLOC and 38k SLOC of Ada source code were sourced from a Defence contractor.

Comparison of the Naming Convention (15) identifier-naming style guideline between the Ada and Java software projects was not found to be meaningful due to the range in cognitive effort required of a programmer to support the various and sometimes unorthodox identifier-naming conventions encountered in the Ada software projects compared to those encountered in the Java software projects. Some Ada software project identifier-naming conventions used a different character case rule for single words than for multiple word identifier names but specific words (eg. false, true) were common exceptions to this rule. Different capitalisation rules were also employed for method parameters than for declarations internal to the method. In addition, different capitalisation rules were apparent depending on the declaration kind (eg. constant, method, package, type, variable). Due to the difference in level of complexity that a programmer would experience between the standard Java project naming conventions, and the relatively complex and varied project naming conventions used for the Ada projects, a comparison between the number of Naming Convention (15) identifier-naming style flaws would not result in a comparison of the same programmer effort. Hence, the Naming Convention (15) identifier-naming style guideline has been removed from further consideration within this section. Consequently any tabulated data that follows is presented on a grey background for the Naming Convention (15) identifier-naming style guideline within the subsections to this section and is not otherwise used during the following statistical treatments of the data.

Note that the values recorded against the Unused Identifier (18), identifier-naming style guideline records the instances where the declared identifier is not used within the class/package of its declaration and does not purport to suggest that the identifier may not necessarily be used elsewhere within the project source code.

5.3.1 Ada and Java Software Comparison

The source code from each software project was independently analysed, the identifier-naming style flaw data collected and programming language data amalgamated. A comparison of the identifier-naming style counts between the Ada and Java source code is not meaningful due to the differences in the number of SLOC collected for the two computer programming languages, and also due to the differences in the numbers and kinds of identifier declarations found in the different software projects. However, the count of specific identifier-naming style flaws is useful to identifying the pervasiveness of the flaws in contemporary production software. A comparison between the Ada and Java software for the mean percentage value of identifier-naming style flaws is meaningful as this value corresponds to the ratio of the flaws to the number of opportunities to raise this flaw which is not dependent on the absolute number of flaws found within the software project.

Table 5-6 reports the identifier-naming style guideline for the corresponding count of the particular identifier-naming style flaws and the mean number of the same flaw as a percentage against the opportunity to raise this flaw for both the Ada and the Java software projects.

Table 5-6 - Identifier-Naming Style Flaws in Contemporary Software

Id	Identifier-Naming Style Guideline Name	Ada Source Code		Java Source Code	
		Counts	Percentage	Counts	Percentage
01	Un-named Constant	14,437	30%	14,939	42%
02	Multiple Underscore	0	0%	2	0%
03	Outside Underscore	0	0%	3,652	1%
04	Numeric Digit(s)	4,979	3%	5,299	4%
05	Short Name	46,398	54%	52,944	44%
06	Long Name	10,022	4%	7,103	5%
07	Word Count	48,588	54%	54,065	42%
08	Identifier Encoding	3,271	3%	2,387	5%
09	Class/Type Qualification	4,489	83%	5,015	99%
10	Constant/Variable Qualification	2,200	3%	931	1%
11	Abstract Words	2,044	4%	1,952	1%
12	English Words	29,778	19%	25,763	19%
13	Numeric Name	75	0%	59	0%
14	Plural Word	4,414	4%	10,531	10%

15	Naming Convention	47,401	52%	16,351	20%
16	Duplicate Names	6,181	6%	6,246	4%
17	Similar Names	9,443	3%	5,178	4%
18	Unused Identifier	27,705	20%	9,637	12%
19	Same Words	262	0%	103	0%

Note: The instances of the Outside Underscore (03) identifier-naming style flaw being reported for the Java source code is not necessarily relevant due to the convention of prefixing variables, which are to be used for operating system and package calls, with an underscore character. This has occurred consistently for this purpose in four of the Java software projects surveyed.

The percentage of identifier-naming style flaws against the opportunity to generate the corresponding flaw was calculated for the Ada and Java data using the Spearman Rank Correlation Coefficient to derive an approximate value for r_s. The value calculated for r_s was 0.9173, with df = 16 (i.e., excluding the Naming Convention (15) identifier-naming style flaw data), giving $p < 0.000001$. The result provides strong evidence that the identifier naming practices encountered in Ada and Java software projects are similar as it would be statistically highly unlikely that the data collected would have come from different populations purely by chance and hence the choice of computer programming language was not significant in modifying the relative percentages of specific identifier-naming style flaws found in the production software.

5.3.2 Computer Program Size

Having previously independently analysed the identifier-naming style flaw data from each software project the data from computer program sizes was amalgamated. The computer program size graduations used by the Programmer Characteristics questionnaire (see Appendix B – Programmer Characteristics Questionnaire) were used to bracket the computer program sizes of the software projects.

Table 5-7 reports mean identifier-naming style flaw as a percentage against the opportunity to raise the corresponding flaw for each of the computer program size ranges. The table also contains a column of the mean percentage value over the entire sample population.

Table 5-7 - Identifier-Naming Style Flaws by Computer Program Size Range

Id	Identifier-Naming Style Guideline Name	0 – 1k	1 – 2k	2 – 5k	5 – 10k	10-100k	Sample Pop.
01	Un-named Constant	17%	36%	32%	46%	38%	33%
02	Multiple Underscore	0%	0%	0%	0%	0%	0%
03	Outside Underscore	0%	0%	0%	0%	1%	0%
04	Numeric Digit(s)	2%	2%	1%	6%	5%	3%
05	Short Name	59%	56%	45%	55%	47%	52%
06	Long Name	2%	1%	6%	3%	7%	4%
07	Word Count	59%	54%	44%	52%	48%	51%
08	Identifier Encoding	4%	3%	5%	3%	3%	3%
09	Class/Type Qualification	86%	97%	91%	63%	90%	87%
10	Constant/Variable Qualification	2%	2%	2%	3%	2%	2%
11	Abstract Words	6%	3%	2%	3%	2%	3%
12	English Words	10%	12%	25%	18%	26%	19%
13	Numeric Name	0%	0%	0%	0%	0%	0%
14	Plural Word	3%	4%	8%	4%	6%	5%
15	Naming Convention	57%	29%	47%	54%	39%	45%
16	Duplicate Names	5%	4%	4%	7%	6%	5%
17	Similar Names	1%	2%	3%	3%	6%	3%
18	Unused Identifier	11%	9%	25%	20%	22%	18%
19	Same Words	0%	0%	0%	0%	0%	0%
	r_s	0.8830	0.9028	0.9630	0.9384		

The percentage identifier-naming style flaws against the opportunity to raise the corresponding flaw was correlated for the smaller program sizes in turn against the 10k+ program size data using the Spearman Rank Correlation Coefficient to calculate an approximate value for r_s. The value calculated for r_s is reported in the last line of the table and was calculated with df = 16 (i.e., excluding the Naming Convention (15) identifier-naming style flaw data), giving a directional value for $p <= 0.000001$ in all four cases. The result indicates that the identifier naming practices encountered in each of the computer program size range groups are similar as it would be highly unlikely that the data collected would have come from different populations purely by chance and hence the computer program sizes of less than 100k SLOC was not significant in modifying the relative percentages of specific identifier-naming style flaws.

5.4 Dated Software - Survey

Having established that programmers in different stages of their transition to expert status hold different attitudes to the identifier-naming style guidelines by the use of the Identifier-Naming Style Guideline Programmer Acceptance questionnaire, investigation of production software developed by an individual over an extended period was conducted. The collection of data for diverse sources proved to be problematic and resulted in the potential substantial devaluing of this component of the research as the author's own software was found to be the only software readily available.

Data collected has been indexed by the year that the software was completed, with year 4 corresponding to the first year of employment as a professional programmer. Production source code was collected from a twenty-eight year period, during which the author was employed in a maintenance and production environment and was engaged with the development of large and complex mission critical software systems. After four years in this environment, the author was promoted to team leader and subsequently to technical manager. For reasons relating to commercial-in-confidence and corporate security, it was not possible to collect source code associated with some early maintenance tasks during year numbers 4 through 7 or to access source code from two large team based projects during year numbers 13 through 15. The source code generated for year 14 was part of a corporate project for which the author specified the coding standard for the project. However, the author had no control over the coding standard for the source code generated for year 21. The remaining source code, that was available for analysis, was generated without the benefit of a code review and was developed in the absence of mandated coding standards. Hence all identifier naming practices, with the exception of that source code generated in year 21, is solely due to the author at the then current appreciation of identifier naming, as modified by experience. All source code collected was written in the Pascal computer programming language, with the exception of the source code generated in year 21, which was written in the Visual Basic computer programming language. The Visual Basic source code, which corresponded to a single software project, was manually translated into Pascal source code so that analysis could be conducted on the identifier declarations using the same source code parser for all of the source code analysed. In total 37k SLOC of Pascal, composed of 13 software projects ranging in size from 158 SLOC to 21k SLOC was collected. The software projects implemented a health insurance system, cruise liner booking system, military audio database and control system, mortgage workflow system, satellite ground station control system; submarine system emulator; and source code editor.

A criticism can be levied that I would have been expected to have had or at least developed an interest in identifier names during the development of the source code used in support of the dated software survey. However, I was not consciously aware of the identifier-naming style guidelines presented in Appendix A – Identifier-Naming Style Guidelines for all but the last software project analysed. Hence, the results should be useful if only to show the chronological progression in identifier-naming style for one individual as an indication of what changes in identifier-naming style is possible over time as a consequence of increased experience. As this activity corresponds to a self assessment, any results collected should not necessarily be considered to be indicative of the general programmer. In other words, this research does not claim that the specific awareness of identifier-naming issues is typical of programmers. Rather, the results are being used to illustrate a potential evolution of personal style which is possible.

5.4.1 Temporal Comparison

Source code collected from the last three software projects being analysed was generated using the Borland Delphi Object Pascal dialect. This product automates the generation of a Graphical User Interface (GUI) and as a consequence also generates a proliferation of identifier declarations necessary to support the GUI. However, the user has inadequate control over the form these identifier declarations take and typically these identifier declarations result in identifier-naming style flaws being reported. When the main bulk of the automatically generated identifier declarations are removed, the mean percentage of identifier-naming style flaws reported reduces from 13% to 9%, 12% to 9% and 9% to 6% for the three software projects concerned. However, it was not possible to comment out the procedure identifier names generated by the GUI tool as commenting-out the entire procedure would remove, from consideration, any of the internal identifier declarations inserted by the programmer. Similarly, renaming of the procedure was considered inappropriate as the GUI tool's naming requirements had originally constrained the procedure name which could affect any attempt to devise a replacement name. Hence the identifier-naming style flaw percentage values for the last three projects would be slightly higher than could be expected without the assistance of the GUI tool.

Table 5-8 reports the mean identifier-naming style flaw as a percentage against the opportunity to raise the flaw for each of the years when a software project was completed. The results reported in Table 5-8 are with the automatically generated identifier declarations removed. The data is tabulated with a row for each of the identifier-naming style flaws, identified by the identifier-naming style guideline Id and with a column for each of the software project completion years. Years 4 through 15 used the Turbo Pascal dialect, years 21 used Visual Basic

and years 22 through 28 used the Delphi Object Pascal dialect. The last column of the table indicates the slope of the mean percentage of identifier-naming style flaws over time but excludes the data for year 28. The result obtained for year 28 cannot be considered significant as the corresponding software project was constructed with full knowledge of the identifier-naming style guidelines used by this research, whereas conscious knowledge of the guidelines was not available for the construction of the prior software projects. The slope was calculated using the least mean squares method. The last row of the table is used to report the mean percentage value over all of the identifier-naming style flaw data but as before excluding the Naming Convention (15) identifier-naming style flaw data. A value appearing on a shaded background (red) corresponds to a slope that is definitively positive.

Table 5-8 - Identifier-Naming Style Flaws in Dated Software

Id	Completion Year										Slope
	4	5	9	12	14	15	21	22	23	28	
01	48%	50%	58%	55%	50%	51%	39%	19%	29%	20%	-1.32
02	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.00
03	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.00
04	8%	3%	8%	7%	5%	9%	23%	10%	10%	1%	0.47
05	62%	52%	21%	26%	26%	27%	10%	13%	13%	4%	-2.22
06	0%	1%	3%	14%	11%	11%	30%	33%	26%	29%	1.69
07	76%	64%	20%	31%	29%	24%	29%	24%	28%	10%	-1.94
08	5%	1%	5%	5%	3%	6%	0%	0%	0%	0%	-0.20
09	98%	54%	5%	0%	0%	0%	0%	4%	5%	10%	-3.45
10	9%	9%	23%	6%	4%	1%	0%	0%	0%	0%	-0.72
11	0%	2%	2%	2%	3%	3%	0%	2%	2%	0%	-0.02
12	43%	31%	31%	12%	11%	1%	5%	15%	16%	2%	-1.41
13	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0.02
14	4%	7%	6%	2%	3%	1%	1%	1%	3%	0%	-0.22
15	60%	35%	0%	63%	1%	53%	52%	27%	40%	10%	-0.01
16	3%	9%	14%	12%	6%	14%	1%	21%	11%	13%	0.23
17	1%	4%	3%	5%	3%	4%	15%	15%	14%	3%	0.72
18	27%	14%	11%	8%	7%	3%	0%	5%	4%	10%	-0.94
19	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	0.02
Mean	21%	17%	12%	10%	9%	9%	9%	9%	9%	6%	

The slope data is further discussed in section 5.4.2.

Figure 5-1 depicts the mean percent of identifier-naming style flaws graphed over the time period corresponding to the number of years programming experience current for the programmer at the time that the software project was completed. The expected exponential decline in the number of identifier-naming style flaws can be seen represented in the graph (recall that the last three values are higher than would be achieved without the GUI capability of Delphi Object Pascal inflating the number of identifier-naming style flaws).

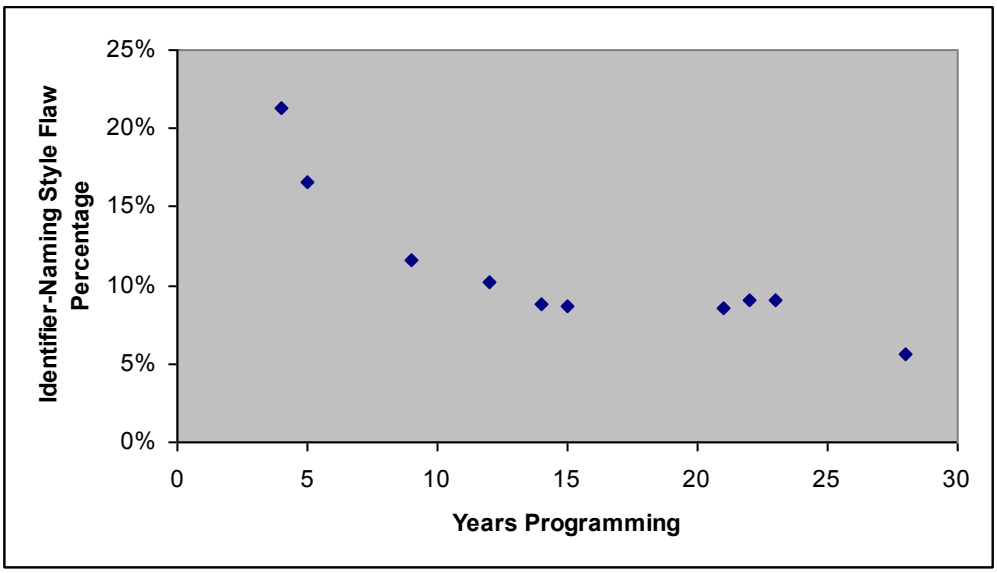

Figure 5-1 - Identifier-Naming Style Flaw Reduction by Time

5.4.2 Identifier-Naming Style Flaw Trends

There was an expectation that as the programmer gains experience, the number of identifier-naming style flaws would decrease. This would require that the percentage of identifier-naming style flaws against the opportunity to raise the corresponding flaw should also decrease. Hence, the expectation was that the slope of a line, generated by the least mean squares method for the percentage of identifier-naming style flaws graphed over time, should be negative. Figure 5-2 depicts the slope these lines, represented as a bar graph, for the source code generated inclusive of years 4 through 23.

As can be seen from the figure, in many cases the line slopes are negative. The instances where the slope is near zero indicates either that the programmer made no changes to their behaviour for the corresponding identifier-naming style flaw or that no corresponding flaw was raised within the source code. However, the positive slope, for the identifier-naming style guidelines

Numeric Digit(s) (04), Long Name (06), Duplicate Names (16) and Similar Names (17) could indicate that as the programmer gains experience that their acceptance in practice of these guidelines decreases. These particular identifier-naming style guidelines will be further addressed during the case studies.

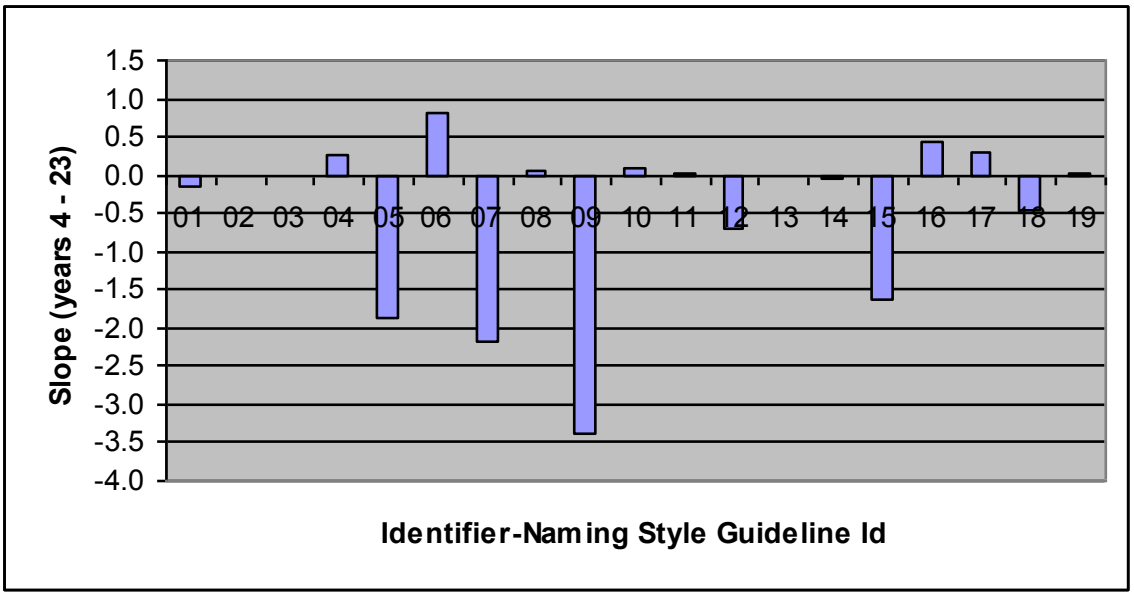

Figure 5-2 - Identifier-Naming Style Flaw Trends

5.4.3 Single Programmer Sample Concerns

The value of data collected from a single individual is questionable, as a sample of one may not necessarily be indicative of the general software engineering population. In order to identify the likelihood that the percentage of identifier-naming style flaws in the Pascal software projects belong to the same population as for the software engineering population, comparison against the percentage of flaws for the combined Ada and Java source code was calculated for each of the years when the Pascal software project(s) completed. A Spearman Rank Order Correlation Coefficient test was used to indicate the strength of the correlation between the combined Ada and Java software projects and the Pascal software projects. The results of a non-directional test with df = 16 (i.e., excluding the Naming Convention (15) identifier-naming style guideline) is reported in Table 5-9.

Table 5-9 - Group Ada/Java and Individual Pascal Software Project Correlations

Completion Year	r_s	p
4	0.8274	0.000012
5	0.8728	0.000002
9	0.6682	0.001225
12	0.6456	0.001909
14	0.6776	0.000992
15	0.5848	0.005442
21	0.5737	0.006421
22	0.6970	0.000651
23	0.7861	0.000055
28	0.7734	0.000084

The results show a very strong correlation to the Ada/Java software projects. There was a basis for concern that the dated software results may not be useful due to the potential for the author to act as a confounding influence as the author obviously has an interest in identifier-naming practices which may not necessarily be shared by the common programmer (and indeed this interest may have increased over time). There was also a secondary concern that temporal effects could manifest due to the relative age of the very early software, making a comparison with contemporary software meaningless. Hence, the above result indicates that the identifier naming practices encountered in the dated source code are similar to those encountered in the contemporary Ada and Java source code, as the likelihood that the data collected coming from a different population is remote.

5.5 Identifier-Naming Style Guideline Validity

Before addressing the results for the maintenance and production experiment and the results of the case studies, the results collected so far will be summarised as an aid to the reading of this thesis.

Table 5-10 addresses each of the identifier-naming style guidelines considered by this research and presents the findings relevant to the individual guideline. A tick indicates that the identifier-naming style guideline may be useful to improve source code readability and a cross indicates the converse. The table lists the identifier-naming style guideline and a discussion

which summarises the findings of the research investigation. Within the discussion the statistical significance is classified as at best *poor* when the significance level is above 0.1, *weak* when the significance level is 0.1, *moderate* when the significance level is 0.01 and *strong* when the significance level is 0.001, after Shneiderman (1980).

Table 5-10 - Identifier-Naming Style Guideline Summary

Id	Identifier-Naming Style Guideline Name	Discussion
01 ☑	Un-named Constant	This guideline was accepted by all software engineer classifications and by expert programmers in particular with weak statistical significance. The guideline shows stronger acceptance in practice as programming experience is gained. Approximately one third of the identifier names declared in production software raise the corresponding flaw, hence the reporting of this flaw would be useful to programmers.
02 ☒	Multiple Underscore	This guideline was accepted by all software engineer classifications and by expert programmers in particular with weak statistical significance. The guideline is also passively supported by the results of the textbook survey. However, the usefulness of this guideline is questionable as instances of the corresponding flaw rarely occur in production software, hence reducing the potential value of this guideline for the programmer.
03 ☒	Outside Underscore	This guideline was accepted by the intermediate programmers and by expert programmers in particular with strong statistical significance. However, the usefulness of this guideline is questionable as instances of the corresponding flaw occur rarely in production software, hence reducing the potential value of this guideline for the programmer.
04 ☑	Numeric Digit(s)	This guideline was accepted by all software engineer classifications and by expert programmers in particular with strong statistical significance. Less than 5% of identifier declarations were found to raise the corresponding flaw, however the guideline remains useful to the programmer due to the

Table 5-10 - Identifier-Naming Style Guideline Summary

Id	Identifier-Naming Style Guideline Name	Discussion
		relatively low frequency with which the programmer could be expected to have previously encountered this flaw.
05 ☑	Short Name	This guideline was accepted by expert programmers with moderate statistical significance. The guideline showed a stronger acceptance in practice as the programmer gained experience. Approximately half of the identifier names declared in production software raise the corresponding flaw; hence the guideline remains useful to the programmers, particularly as generally both the novice programmer and the intermediate programmer do not perceive the guideline as directing towards improved source code readability.
06 ☒	Long Name	This guideline was accepted by expert programmers with moderate statistical significance. Out of all the guidelines considered this guideline showed the strongest decrease of acceptance in practice as the programmer gained experience. The guideline is questionable due to the increasing tendency of expert programmers to ignore the guideline in specific cases, which were relevant to the programmer.
07 ☑	Word Count	This guideline was accepted by all software engineer classifications and by expert programmers in particular with weak statistical significance. The guideline shows stronger acceptance in practice as programmer experience is gained. Approximately half of the identifier names declared in production software raise the corresponding flaw; hence the reporting of this flaw would be useful to programmers.
08 ☑	Identifier Encoding	This guideline was accepted by all software engineer classifications and by expert programmers in particular with moderate statistical significance. This guideline is also passively supported by the textbook survey, as none of the textbooks

Table 5-10 - Identifier-Naming Style Guideline Summary

Id	Identifier-Naming Style Guideline Name	Discussion
		surveyed used Hungarian notation in the construction of identifier names that were used within the example source code. Less than 5% of identifier declarations were found to raise the corresponding flaw, however the guideline remains useful due to the relatively low frequency with which the novice programmer could be expected to have previously encountered this flaw.
09 ☒	Class/Type Qualification	As the responses by the questionnaire respondents were strongly biased towards responding to the Java source code skeleton methods and all test subjects used the Java computer programming language in response to the maintenance and production experiment, no observations are possible for the use of a Type qualification to an identifier name as this is specific to Ada source code. This guideline was rejected by all software engineer classifications and by expert programmes in particular with strong statistical significance. The value of this guideline, to the programmer, is suspect as 99% of the identifier declarations were found to raise the corresponding flaw, hence reducing the potential value of this guideline for the programmer.
10 ☑	Constant/Variable Qualification	This guideline was accepted by all software engineer classifications and by expert programmers in particular with strong statistical significance. Less than 5% of identifier declarations were found to raise the flaw corresponding to this guideline, however the guideline remains useful due to the relatively low frequency with which the novice programmer could be expected to have previously encountered this flaw.
11 ☑	Abstract Words	This guideline was accepted by all software engineer classifications and by expert programmers in particular with moderate statistical significance. Less than 5% of identifier declarations were found to raise the corresponding flaw, however the guideline remains useful due to the relatively low frequency

Table 5-10 - Identifier-Naming Style Guideline Summary

Id	Identifier-Naming Style Guideline Name	Discussion
		with which the novice programmer could be expected to have previously encountered this flaw.
12 ☑	English Words	This guideline was accepted by all software engineer classifications and by expert programmers in particular with strong statistical significance. The guideline shows stronger acceptance in practice as programmer experience is gained. Approximately 20% of the identifier names declared in production software raise the corresponding flaw; hence the reporting of this flaw would be useful to programmers.
13 ☒	Numeric Name	This guideline was accepted by all software engineer classifications and by expert programmers in particular with strong statistical significance. However, the usefulness of this guideline is questionable as instances of the corresponding flaw rarely occur in production software, hence reducing the potential value of this guideline for the programmer.
14 ☑	Plural Word	This guideline was accepted by all software engineer classifications and by expert programmers in particular with weak statistical significance. Less than 5% of identifier declarations were found to raise the corresponding flaw, however the guideline remains useful due to the relatively low frequency with which the novice programmer could be expected to have previously encountered this flaw.
15 ☑	Naming Convention	This guideline was accepted by all software engineer classifications and by expert programmers in particular with weak statistical significance. The presence of this flaw varies depending on the particular software project, however in cases with software written in the Java computer programming language, 20% of the identifier names were found to raise the corresponding flaw and hence the reporting of this flaw would be

Table 5-10 - Identifier-Naming Style Guideline Summary

Id	Identifier-Naming Style Guideline Name	Discussion
		useful to programmers.
16 ☑	Duplicate Names	This guideline was accepted by all software engineer classifications and by expert programmers in particular with strong statistical significance. Less than 5% of identifier declarations were found to raise the corresponding flaw, however the guideline remains useful due to the relatively low frequency with which the novice programmer could be expected to have previously encountered this flaw.
17 ☑	Similar Names	This guideline was accepted by all software engineer classifications and by expert programmers in particular with strong statistical significance. Less than 5% of identifier declarations were found to raise the corresponding flaw, however the guideline remains useful due to the relatively low frequency with which the novice programmer could be expected to have previously encountered this flaw.
18 ☑	Unused Identifier	This guideline was accepted by the novice programmers and the expert programmers in particular with strong statistical significance. The guideline shows stronger acceptance in practice as programmer experience is gained. As approximately 15% of the identifier names declared in production software raise the corresponding flaw, the reporting of this flaw would be useful to programmers.
19 ☒	Same Words	This guideline was accepted by all software engineer classifications and by expert programmers in particular with weak statistical significance. However, the usefulness of the guideline is questionable as instances of the corresponding flaw occur rarely in production software, hence reducing the potential value of this guideline for the programmer.

Almost all research on understanding software maintenance discusses and investigates methods to measure or improve subsequent maintenance actions. However, the work conducted by Chan and Yang (2002) is different in that it reports a simple identifier-naming style treatment that effectively disrupts the ability of a programmer to maintain software to such an extent that it is found to be cheaper to entirely rework the software solution than to attempt maintenance of the treated source code. This simple identifier-naming style treatment is to overuse the same name, which has been allocated to identifiers of different kinds, trusting on the compiler to distinguish the unique identifiers by kind. This practice burdens the programmer with a large cognitive overhead to understanding the source code, that even small computer programs can more easily be rewritten than maintained (Chan and Yang, 2002).

A survey of the programming textbooks, used by the author, was found to be dismissive of the identifier-naming style guidelines. While these textbooks were being actively used by the author, the author generated source code which, over time, progressively contained a reducing percentage of identifier-naming style flaws contained within the source code. Further to the analysis of the dated Pascal source code developed by the author, analysis of contemporary source code identified that it would be highly unlikely that the distribution of identifier-naming style flaws, found in the Ada, Java and Pascal source code, would be representative of different population spaces. Similarly, it would also be highly unlikely that the identifier-naming style flaws found in source code, belonging to different sized software projects up to 100k SLOC, would also be representative of different population spaces.

5.6 Source Code Editor

Having ascertained the attitudes of programmers to the identifier-naming style guidelines and having analysed both contemporary and dated source code for the presence of identifier-naming style flaws, consideration was then directed towards identifying the kinds of flaws that are inserted and subsequently removed in source code. This consideration necessitated the development of a source code editor that would support the dynamic reporting of identifier-naming style flaws to the programmer. Once the source code editor found and reported an identifier-naming style flaw, the source code editor's user interface could, in some cases, suggest a possible correction and could automatically replace all relevant identifier names in scope of the identifier declaration. This capability was implemented to minimise the editing effort required of the test subject, as the aspect relevant to the research was the choice of replacement identifier name and not the editing abilities of the programmer.

As the source code editor would initially be employed to support the maintenance and production experiment, trials were run incorporating the experiment's instructions as a basis for evaluating the fitness of the source code editor and the understandability of the experiment instructions. This action was taken to primarily confirm the suitability of the source code editor for the maintenance and production experiment, and also to confirm the suitability of the source code editor as a research tool to support the case studies.

Trials were conducted on: (1) a professional programmer who had in excess of ten years programming experience, (2) a novice programmer who had only recently graduated from a Computing Science Bachelors degree but who had a considerable number of years working in industry, (3) a Human Factors expert who, in addition to their psychology masters, also had a Computer Science Bachelors degree and (4) a first year computing science student who had less than six months exposure to programming.

The professional programmer demonstrated no issues with the use of the source code editor but lamented the absence of a menu driven find/replace capability. This feature was not supplied as this functionality was not initially considered to be relevant to the operation of the research tool. This decision appeared to be moot as the test subjects who would use the source code editor, particularly the programmers who were engaged in the case studies, would edit the source code in their preferred editors and only use the source code editor to report the occurrence of any identifier-naming style flaws. The professional programmer also complained about the tool's speed on large files, which has been previously addressed in section 3.8.4.

The novice programmer discovered a number of what were considered to be trivial bugs with the source code editor's user interface (eg. the automatic generation of a suggested replacement identifier name that shifted a plural word, that formed part of the identifier name, into the singular form but resulted in the singular word having an incorrect Java naming convention identifier name representation for the resultant identifier name). This and other similarly minor 'features' of the user interface had a marked effect on the novice programmer's ability to tolerate the use of the source code editor. Hence effort was expended to identify and remove these minor bugs should their presence also adversely affect the collection of data during the maintenance and production experiment or the case study.

The Human Factors expert showed little difficulty in interpreting the instructions or using the source code editor. However, their comments on the instructions resulted in a change to the presentation style to increase the prescriptive nature of the instructions and also to simplify the wording used in the instruction text. In addition to these changes, important concepts (eg. menu

selections, file names and relevant identifier names) were bolded in the instructions to allow the test subject to easily scan the instructions for important information. The instructions were also reformatted to highlight the boundary between one part of the experiment and the next.

As expected the first year university student required extensive assistance with the construction of syntactically correct Java source code. However, they did demonstrate a correct understanding of the maintenance and production experiment instructions as their actions and their requests for assistance was in concert with the instructions. As a consequence of observing the first year university student, the maintenance and production experiment instructions were further simplified to point explicitly to the parts of the source code that required modification so as to reduce the time required to interpret and follow the instructions.

5.7 Programmer Characteristics Questionnaire

Table 5-11 lists the percentage of meaningful identifier names (and not the percentage of identifier-naming style flaws) generated by the 34 groups of test subjects who participated in the maintenance and production experiment and who also responded to the Programmer Characteristics questionnaire (see Appendix B – Programmer Characteristics Questionnaire). The table contains entries against each of the programmer characteristics recorded in the Programmer Characteristics questionnaire. Each entry is comprised of an Id to facilitate ease of referencing during the discussion that follows; the relevant confounding variable's name and list of options available for the test subject to select from in their response to the Programmer Characteristics questionnaire; the percentage of meaningful identifier names combined over both the control groups and the experimental groups, the control groups only; the experimental groups only; and the statistical significance of the data collected between the control groups and experimental groups. Where data is not available for the percentage of meaningful identifier names, a dash has been inserted into the table. Where "N/A" appears in the table, there is no statistical significance present between the relevant data.

Table 5-11 - Programmer Characteristics

Id	Confounding Variable	Meaningful Identifier Names			
		Combined	Control	Exp.	Statistical Significance
1	Professional Programmer				
	No:	27%	15%	29%	N/A
	Yes:	37%	36%	37%	N/A
2	Years Programming				
	1:	39%	35%	41%	N/A
	2:	34%	22%	43%	N/A
	3:	23%	12%	28%	N/A
	4 - 9:	35%	35%	36%	N/A
	10+:	57%	53%	60%	N/A
3	Program Size (k SLOC)				
	0 - 1:	32%	0%	36%	0.05
	1 - 2:	28%	26%	31%	N/A
	2 - 5:	25%	27%	23%	N/A
	5 - 10:	44%	-	44%	N/A
	10+:	43%	41%	47%	N/A
4	Subjected to Source Code Review				
	No:	33%	17%	38%	0.025
	Yes:	36%	37%	36%	N/A
5	Source Code Reviewer				
	No:	34%	24%	38%	N/A
	Yes:	35%	34%	37%	N/A
6	Touch-Type				
	No:	28%	24%	32%	N/A
	Yes:	40%	38%	40%	N/A
7	Identifier-Naming Effort				
	Much More:	52%	54%	51%	N/A
	More:	33%	18%	36%	N/A
	Average:	35%	32%	36%	N/A
	Less:	27%	23%	30%	N/A
	Much Less:	23%	23%	-	N/A

Table 5-11 - Programmer Characteristics

Id	Confounding Variable	Meaningful Identifier Names			
		Combined	Control	Exp.	Statistical Significance
8	Flexible Work Practice				
	Much More:	51%	-	51%	N/A
	More:	37%	34%	38%	N/A
	Average:	31%	30%	33%	N/A
	Less:	29%	23%	32%	N/A
	Much Less:	-	-	-	N/A

Considering the "Combined" column, as expected, the professional programmers generated a larger percentage of meaningful identifier names than the student programmers (Id 1); programmers who have had their source code reviewed at some time in the past generated a larger percentage of meaningful identifier names than programmers who have never had their source code reviewed (Id 4); programmers who have reviewed someone else's source code generated a larger percentage of meaningful identifier names than programmers who haven't reviewed someone else's source code (Id 5); and programmers who can touch-type generated a larger percentage of meaningful identifier names than a programmer who cannot touch-type (Id 6). As expected, the programmer's self reporting of their typical identifier-naming effort expended was reflected by a reduction in the percentage of meaningful identifier names as the reported level of effort reduced (Id 7). Also, as expected, the percentage of meaningful identifier names reduced with the reduction in work practice flexibility as reported by the programmer (Id 8). The results for the programmer characteristics, identified by Id 1, 4, 5, 6, 7 and 8, are consistent with the result of other researchers.

Again, considering the "Combined" column, for the Program Size (Id 3) programmer characteristics, the results show that programmers in their first year of study, who would have been typically expected to only have generated computer programs of less than 1k SLOC, produced a higher percentage of meaningful identifier names than programmers who had produced only slightly larger computer programs of 1 – 5k SLOC in size. The experience with slightly larger computer programs would imply greater experience and hence additional years on that of the programmers who had only produced small computer programs. Similarly, those programmers who have had experience with computer programs of more than 5k SLOC would be expected to have more experience than those programmers who only had generated computer

programs of 1 – 5k SLOC in size. However, the results show a continuing decline that requires a number of years experience to recover to the same level as a first year student which is evident in the results for the Years Programming (Id 2). This trend is further addressed in the following section.

5.8 Maintenance and Production Experiment

The test subjects chosen to participate in the maintenance and production experiment were Software Engineering students enrolled at the University of Technology, Sydney (UTS) in the undergraduate *48475: Software Systems Analysis* and *48485: Software Systems Design* courses and the postgraduate *49263: Software Analysis & Design* course. 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 exercises completed in that order and in a single sitting within the environment of a computer lab. The Software Systems Analysis and the Software System Design students were allocated to the experimental group effectively replicating this group and the Software Analysis & Design students were allocated to the control group. In total 69 students participated forming 35 groups that completed all the exercises. A greater number of groups than 35 participated, but only 35 groups completed all the exercises.

In addition ten professional programmers, with two drawn from academia and eight drawn from industry, also participated in the maintenance and 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.

5.8.1 Expert Programmer Identifier Name Adjudication

An expert programmer, with over twenty-five years experience developing and managing the development of large mission critical software systems was requested to adjudicate on the meaningfulness of identifier names generated by the test subjects engaged by the maintenance and production experiment. The expert programmer was given the skeleton source code listings reported in Appendix E – Experiment Files and a table of the identifier names generated by the test subjects. The test subject's identifier names were separated by file, corresponding to the three exercises forming the maintenance and production experiment, and by identifier type within each file Duplicate identifier names devised by the test subjects had the duplicates

removed from the table and the identifier name lists were sorted alphabetically for each of the identifier types appearing in the experiment files.

The intent was to use the test subject's identifier names, chosen by the expert programmer as being meaningful for the given context, to select the identifier names that satisfied the criteria for readability. However, the variety of identifier names chosen by programmers is large, as will be seen in the following sections. Hence, three additional expert programmers, all with over fifteen years of experience with large mission critical software systems, were also asked to repeat the adjudication process for the specific identifiers relevant to the introduction exercise. This was done to ensure against using bias identifier name adjudication from the initial expert programmer. These additional expert programmers were not similarly asked to adjudicate on the test subject's identifier names relevant to the maintenance exercise or the production exercise due to the effort that this would impose on the additional expert programmers. The lists of identifier names selected by the four expert programmers, as being meaningful, were not identical, which was expected. Hence linear correlation was used to compare the results of the additional expert programmers against the results of the initial expert programmer.

The percentage of meaningful identifier names was calculated individually for each of the 44 test subject groups that completed the introduction exercise, separately against each of the four expert programmer's identifier name lists. Three pairings were compared for their linear correlation, i.e., each of the additional expert programmer's values in turn against those values derived from the initial expert programmer. Table 5-12 reports the mean and standard deviation percentage of meaningful identifier names as adjudicated by the four expert programmers. In addition, the three pairings of the additional expert programmers results (i.e., Expert 1, Expert 2 and Expert 3) against the results from the initial expert programmer were compared using linear correlation, the non-directional Student t-test score and the corresponding probability value for the sample distribution is reported in the table.

Table 5-12 - Introduction Exercise Correlation between Expert Programmer Identifier Name Adjudication

	Expert 1	Expert 2	Expert 3	Initial Expert
Mean	19.11%	19.50%	20.45%	42.98%
Std. Dev.	26.07%	20.87%	21.75%	31.54%
t	5.022	4.476	4.805	
p	<0.0001	0.0001	<0.0001	

The results show that despite the different identifier name lists, that the expert programmers selected, the selections made by the expert programmers were consistent in their identification of identifier-naming style practices with very high statistical significance.

5.8.2 Control/Experimental Group Differences

As noted previously, 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. This allocation was arbitrarily made and was based solely on the class sizes in an attempt to balance the size of the control and experimental groups. However, it transpired that the Software Analysis & Design course was populated by postgraduates and the two other courses were populated by 3rd year undergraduates. Hence, comparison of results between the control groups and the experimental groups may not necessarily be meaningful due to the difference in education between the two population samples. A Mann-Whitney test, which is a non-parametric test sensitive to the differences in distributions of two independent samples, returns a probability of 0.6241 for a non-directional test that the two samples differ due to chance. This probability value is sufficiently large as to make comparison of data between the control groups and the experimental groups highly questionable.

Comparing the two population samples may not be meaningful over the individual tutorial groups but it can be argued that comparing the results for individuals with a similar number of programming years experience would be possible. Figure 5-3 depicts the average percentage of meaningful identifier names graphed against the number of years programming experience for both the control groups and the experimental groups. The figure shows that for the coding groups, from the relevant question in the Programmer Characteristics questionnaire, that the average percentage of meaningful identifier names is greater for the experimental groups than for the control groups. The percentage of identifier-naming style flaws also showed a decrease when the test subject was benefited by dynamic reporting of flaws, with the control groups generating at least one flaw in 57% of the identifiers that they declared compared to 44% for the experimental groups.

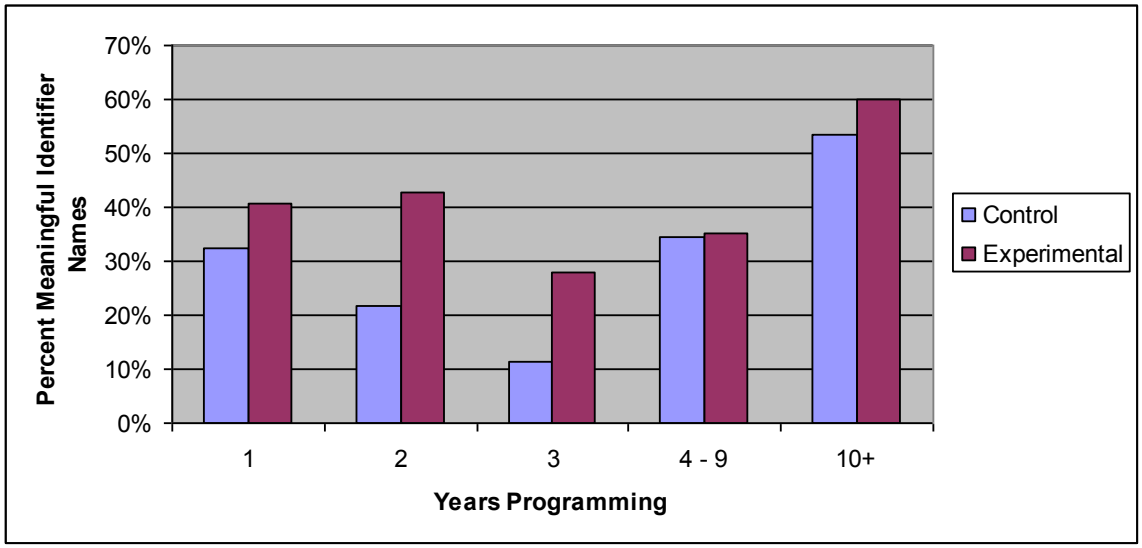

Figure 5-3 - Average Percentage of Meaningful Identifier Names

5.8.3 Readability Metrics

The research was hoping to use the predicted readability metric (DeYoung and Kampen, 1979) and the maintainability index metric (Pearse and Oman, 1995) as metric values to assess the readability of the source codes developed during the case studies. However, before use of the metric values can be made, effective demonstration that the metric values are responsive to the changes occurring in the source code generated as a consequence of using identifier-naming style flaw reporting is required. The introduction exercise was ideal for this evaluation, as the source code structure was the same, with the only difference being the choice of identifier name proposed by the test subjects. As both the experimental and control groups had access to the dynamic reporting of identifier-naming style flaws and as the same expert programmer's list of meaningful identifier names was used for each source code file, all external variables except for the test subjects themselves, were the same.

Table 5-13 lists the Pearson correlation value which was calculated against the percentage of meaningful identifier names in the source code files against the predicted readability metric and also against the maintainability index value. In addition, the Pearson correlation value was calculated against the predicted readability metric and the maintainability index value. For all Pearson correlation values listed, the corresponding probability of a direct correlation is also listed. The Pearson correlation values were calculated with a df = 42.

Table 5-13 - Readability Metric Correlations

	r_s	Probability
Predicted readability metric vs. percentage of meaningful identifier names	0.2342	> 0.1
Maintainability index metric vs. percentage of meaningful identifier names	0.0114	> 0.1
Predicted readability metric vs. maintainability index metric	0.0503	> 0.1

The table shows that there were no correlations evident and that the predicted readability metric and the maintainability index metric can not be used as indicative measures of source code readability, when considering identifier naming. This result was expected, see sections 3.7.3 and 3.7.4.

5.8.4 Introduction Exercise

The test subjects were requested to replace the class name "zero", the method name "fred" and the variable name "A_global" with more meaningful identifier names and to qualify the numeric literal "10" with a named constant. All test subjects irrespective of whether they formed part of the control or the experimental groups were supported by the dynamic reporting of identifier-naming style flaws during the introduction exercise.

Using the source code editor to edit a file did not appear to present any difficulty to the test subjects 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 identifier names before choosing to ignore the dynamic reporting of the identifier-naming style flaw. Other test subjects similarly exhibited difficulty in devising identifier names but required fewer iterations before their final selection.

Table 5-14 reports for each identifier kind: the total number of test subject groups that supplied data for analysis; the number of unique identifier names devised by the test subjects; the percentage 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 percentage of identifier names that were considered meaningful to the intent of the identifier by an expert programmer.

Table 5-14 - Introduction Exercise Results

Identifier Kind	Test Subject Groups	Unique Id Names	Accepted Id Names	Meaningful Id Names
Class	43	27	58%	56%
Constant	44	33	50%	20%
Method	43	31	70%	42%
Variable	44	27	70%	50%

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 25% (44/174) 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 guidelines implemented by the source code editor and were also subject to a high level of suggestion from the embedded 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 "Time Index" and even with this level of prompting only 30% (13/44) of the test subjects choose to label the corresponding identifier with the name "timeIndex".

The most common identifier name chosen by the test subjects, for the class name, was "TimeIndexClass", with nine instances declared. The most common constant name was "TIME_INDEX_MAXIMUM" with seven instances declared. The most common method name was "incrementTimeIndex" with eight instances declared and the most common variable name was "timeIndex" with thirteen instances declared. Some of the more meaningful identifier names suggested by the test subjects were composed of the following word combinations: count, counter, current time, time, time count, time counter, time index value and time value. In addition, the use of abbreviations in the identifier names was high with the presence of common abbreviations such as "idx", "inc", "int", "Max", "Min" and "Num". There was also a number of less common abbreviations that generally required the context of the identifier's use in order to establish the identifier's meaning, such as the abbreviations "cls", "EC", "incr", "Mod" and "ti". Such abbreviations may be meaningful to the originator of the source code but are

generally not read with meaning by a second programmer, without investing some additional effort to understand the contextual use of the identifier.

5.8.5 Maintenance Exercise

The test subjects were requested to conduct a number, of what were believed to be simple maintenance actions, based on the performance of the programmers employed during the source code editor trials. The maintenance actions required the test subjects to: (1) insert a new colour enumerate; (2) extend a Selection statement; (3) increase an array size; (4) change a secondary sort key; (5) correct a typographical error; (6) remove the declaration of an unused identifier; and (7) modify a function result.

Table 5-15 reports, for each identifier kind: the total number of test subject groups that supplied data for analysis; the number of unique identifier names devised by the test subjects; the percentage 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 percentage of identifier names that were considered meaningful to the intent of the identifier by an expert programmer. The data for the accepted identifier names and the meaningful identifier names has been separated for the control groups and the experimental groups, with the control group data presented first.

Table 5-15 - Maintenance Exercise Results

Identifier Kind	Test Subject Groups	Unique Id Names	Accepted Id Names		Meaningful Id Names	
			Control	Exp.	Control	Exp.
Colour constant	20	3	100%	78%	100%	100%
Flag	32	32	48%	82%	10%	9%
Numeric constant	30	3	5%	0%	5%	0%
Value store	4	4	100%	33%	100%	0%

The Maintenance exercise demonstrated but did not attempt to quantify the level of difficulty that programmers show when modifying source code written with poor identifier names. Quantification of the level of difficulty or completeness of understanding 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. Conversely, 95% of the test subject groups were successful in correctly completing a similar maintenance activity that did not mix named constants with numeric literals.

Table 5-16 reports the percentage of test subject groups, separated into the student groups and the professional programmers, that failed to respond correctly to the maintenance exercise activities.

Table 5-16 - Failed Maintenance Actions

Maintenance Activity	Student Group	Professional Programmer
(1)	86%	30%
(2)	6%	0%
(3)	6%	10%
(4)	51%	20%
(5)	17%	0%
(6)	57%	20%
(7)	43%	30%

The result indicates some considerable difficulty that the test subjects experienced, particularly the student group, with the correct execution of the experiment instructions. In particular, the test subjects demonstrated difficulty with maintenance activities (1), (4), (6) and (7). This outcome is further discussed below. As the test subjects were allowed to complete the maintenance exercise at their own rate, it was not expected that insufficient time could be attributed as a cause.

Maintenance activity (1) had a high overall failure rate of 71% with the test subjects appearing not to understand the function, as the relevant line of code inserted by the test subjects typically returned the same colour enumerate value as that passed by parameter and not the next colour enumerate value as required by the function and as described by the function header. The confusion resulted from the mixing of numeric literals with named constants. The high success rate for maintenance activity (2) demonstrated that the test subjects could correctly insert source code as required by a similar maintenance activity and hence the purposeful inconsistent mixing of numeric literals with named constants very effectively disrupted the maintenance action. This result was unexpected but has little impact on the experiment results as the apparent test

subject confusion was most likely centred on algorithmic understanding and not on the declaration of a new identifier.

Maintenance activity (4) required the test subjects to modify a sort key. The source code was purposefully written to use inconsistent identifier naming, in that the initial sort key required the use of the variable "P_" and the replacement identifier name, required to be selected by the test subjects, was the variable "F". Both identifiers were declared together in a simple record structure containing three declarations. Of the 42 groups that supplied data for the maintenance exercise 13 groups renamed the relevant identifier names using the capability of the source code editor, one would assume in an attempt to better understand the source code. This assumption is based on the replacement of the identifiers "P_" and "F" with identifier names that better named these identifiers with their actual function (i.e., prime number and Fibonacci number respectively). Of the 13 groups that did modify the relevant identifier names, 54% correctly modified the source code and of the 29 groups that did not modify the relevant identifier names, 55% correctly modified the source code. Hence the poor performance of the test subjects cannot be attributed to poor identifier naming in this instance. The experiment design is not sensitive to being able to identify the cause of the poor performance and further observation on this issue is not meaningful.

Maintenance activity (6) required the test subjects to delete the declaration of an unused identifier. As there was no statistical difference in the performance of the control group over that of the experiment group, and as only 47% of the experimental test subjects removed the duplicated declaration, the source code editor's reporting of unused identifiers was ineffective in this instance. Further investigation discovered that the small size of the identifier name, which consisted of a single character, effectively allowed the identifier declaration to be hidden in the source code and hence the reporting, that the source code editor provided, was effectively hidden from the test subject's view. This result potentially highlights a defect in the experiment and hence data that would have identified whether the dynamic reporting offered, by the source code editor, to highlight unused identifiers should not be relied on to be meaningful in this case. However, the flaw in the experiment also highlights that the use of short identifier names, i.e., the identifier "i" in this instance, can result in the associated identifier declaration being missed by the maintenance programmer. Even with the explicit instruction to remove the duplicate identifier names, more than half of the text subjects failed to complete this action.

Maintenance activity (7) required the test subjects to modify a function which was expected to result in the declaration of a Boolean flag, a named constant and possibly a temporary store, depending on the efficiency of the source code constructed by the test subjects. Of the 42

groups that completed the maintenance exercise, 31 unique Boolean flag identifier names were devised, four temporary stores were declared and only 3 named constants were declared. This last result was disappointing as the introduction exercise had previously demonstrated that the source code editor attends to the construction and insertion of the named constant's syntax while automatically replacing any occurrences of the associated numeric literal once the test subject had selected an identifier name.

The remaining maintenance actions typically appeared not to create issue for the test subjects. 86% of the groups were successful in finding the commented-out assignment in the "Swaps" method and either inserted the missing word into the identifier name or corrected the source code by renaming the identifier to a much shorter name. As the test subjects were told which line required modification and what word was missing from the identifier name, the scope for potential confusion was consequently minimal. 95% of the groups were successful in inserting source code in the Switch statement to return the 4th prime number. It would appear that the existing source code, which was written in terms of numeric literals only, was sufficiently transparent as to present minimal maintenance challenge to the test subjects. Also, 95% of the groups were successful in updating the named constant. As the test subjects were directed to a specific line and requested to increase the size of an array from "10" to "20", the scope for potential confusion appears to have been minimal.

5.8.6 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 their data was removed from further consideration as the identifier name declarations were not of their own invention.

Table 5-17 reports, for each identifier kind: the total number of groups that supplied data for analysis; the number of unique identifier names devised by the test subjects; the percentage 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 percentage of identifier names that were considered meaningful to the intent of the identifier by an expert programmer. The data for the accepted identifier names and the

meaningful identifier names has been separated for the control groups and the experimental groups, with the control groups presented first.

Table 5-17 - Production Exercise Results

Identifier Kind	Test Subject Group Number		Unique Id Names	Accepted Id Names		Meaningful Id Names	
	Control	Exp.		Control	Exp	Control	Exp.
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 considerable individuality, particularly considering that the sort algorithm was given to the test subjects as embedded structured English within the source code file that they were required to modify. Those test subjects that deviated from the given algorithm used their own sort methods, not all of which would correctly sort the array. Hence not all identifiers that were expected to be declared were actually declared. However, irrespective of the sort method used by the test subjects, the declaration of identifiers occurred, which could then be used for the purposes of the experiment.

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 devised by the test subjects with two instances each being declared for “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 in the choice of identifier name.

5.8.7 Software Bugs

The test subjects were requested to code the Exchange Sort algorithm as the main activity of the production exercise from the maintenance and production experiment. This was achieved to some level of success in many cases but some test subjects generated source code that deviated from the Exchange Sort algorithm and in some cases the source code contained software bugs. This result was not anticipated, as there was an expectation that the test subjects would have been able to successfully generate source code that would compile to execute the Exchange Sort algorithm. In particular, as a structured English description of the Exchange Sort algorithm was imbedded in the source code file that the test subjects were requested to edit and also because the algorithm is both uncomplicated and previously known to the test subjects. The opportunity to analyse the test subject's source code and to correlate the types of software bugs to the percentage of meaningful identifier names being presented, was hence capitalised on.

The source code, developed as part of the production exercise, was categorised as: (a) different from the specified algorithm; (b) containing major software bugs; (c) containing minor software bugs; and (d) containing no software bugs. A software unit that would compile and faithfully execute the Exchange Sort algorithm was ladled as containing no software bugs. A software unit that contained only trivial compilation errors (eg. the absence of a required semicolon character) and/or inefficient source code (eg. a redundant Else statement, the declaration of an array to hold a single temporary value) but that faithfully defines the Exchange Sort algorithm was ladled as containing minor software bugs. A software unit that contained a logic flaws (eg. an infinite loop, fall-through always after a single iteration) was ladled as containing a major software bug. The software units that implemented a different sort algorithm (eg. the Binary Sort), called a library function to sort the data or where the test subject directly copied the Exchange Sort source code from the maintenance exercise, were not considered further for the purpose of this analysis.

Table 5-18 reports the mean and standard deviation for the number of meaningful identifier names per software unit.

A Student's t-test rejected the hypothesis that the data collected from source code that contained no software bugs and the data collected from source code that contained major software bugs, had come from the same population with a statistical significance of 0.0349. This result demonstrates a statistically significant correlation between the use of meaningful identifier names and the absence of software bugs but it does not imply that there is a causal relationship.

Table 5-18 - Software Bugs and Meaningful Identifier Names

Software Bugs	Meaningful Identifier Names	Standard Deviation
Major	24%	19%
Minor	32%	21%
None	46%	14%

5.8.8 Acceptance of Suggested Replacement Identifier Names

The source code editor was capable of suggesting a replacement identifier name for specific identifier-naming style flaws. The introduction exercise was designed for the investigation into whether or not the test subject would use this capability or not. Of the 41 groups who made some attempt to generate meaningful identifier names, 22% (i.e., 9/41) of the groups did not accept any of the identifier-naming style recommendations and devised replacement identifier names, for each of the identifiers which were defined within the introduction exercise without assistance from the source code editor.

In 82% (i.e., 56/65) of the relevant instances where the test subject used a suggested identifier name as a replacement, one or two iterations were required before the final identifier name was selected by the test subjects. However, some test subject required up to seven iterations, involving both suggestions from the source code editor and using direct entry by the test subject, before the final identifier name was selected by the test subject. In total 57% (i.e., 36/65) of the final identifier names selected by the test subjects that were assisted by the source code editor's suggestion for an identifier name were considered meaningful by an expert programmer. Consider that only 25% of the identifier names selected by the test subjects during the introduction exercise were considered to be meaningful by an expert programmer (see section 5.8.4), an almost two-fold increase in the number of meaningful identifier names was apparent when the test subjects were influenced by the suggestions offered by the source code editor.

This result is consistent with the test subjects self reporting of the effort that they place into choosing identifier names, as reported by the results of the Programmer Characteristics questionnaire. The correlation between the test subjects's self reporting of identifier-naming effort and their willingness to consider the suggested identifier names offered by the source code editor during the selection of their final identifier name, was calculated giving a Pearson's correlation value r_s of -0.2926 with df = 32 which is significant at the 10% level. Hence, the result may not necessarily be useful as those programmers who identify themselves as putting

more effort than the average programmer into choosing meaningful identifier names were also the most willing to accept suggestion from the source code editor, in the choice of their identifier names. This result also means that those programmers that are not inclined to invest effort into their identifier naming will not necessarily use a suggested replacement identifier name, even if it is presented to them.

5.8.9 Maintenance and Production Experiment Completion Time

The use of a source code editor to dynamically report identifier-naming style flaws has the potential to consume more time responding to the reporting of flaws during coding than would be consumed by a code review and the consequent effort required to respond to the reported flaws. Investigation of the time spent editing the test files, used in the maintenance and production experiment, was undertaken in order to establish the time required for a test subject to complete the experiment.

Table 5-19 reports the mean number of minutes taken by the test subjects to complete the introduction, maintenance and production experiments. Comparison of results between test subject groups only allowed for a comparison where the test subject reported 4 – 9 years programming experience. This was due to lack of data available for comparison for the other year ranges. The table separates the results for the control and the experimental groups, and amalgamates the student and professional programmer test subjects to report a mean time in minutes. In addition, the statistical significance of the result has been calculated using a non-directional Student t-test, which reports the probability that the individual times belong to a different population distribution.

Table 5-19 - Maintenance and Production Experiment Times

	Control (Minutes)	Exp. (Minutes)	Statistical Significance
Introduction exercise	8.6	8.1	0.88
Maintenance exercise	36.6	30.2	0.57
Production exercise	24.2	21.1	0.74

5.8.10 Experiment Overview

The test subjects were expected to create three identifiers during the maintenance exercise and seven identifier declarations during the production exercise. 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 least variability in choosing an identifier name occurred for another numeric literal where 13% of the identifier names devised in the creation of the relevant named constant were unique. The difference in the range of diversity for the two named constants was due to the instructions requiring that the test subjects replace the first numeric literal with a named constant but the choice was left to the test subjects in the second case, resulting in few identifier names actually being created for the second case.

Comparison between only 22 groups (i.e., 11 control and 11 experimental) was possible as not all test subjects supplied data for both the maintenance and the production exercise, or there was evidence that an exercise was incomplete, 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 relevant maintenance exercise and the relevant production exercise was 29% for the control group and 44% for the experimental group, with a statistical significance of 0.0488 using a Mann-Whitney non-directional test. However, this result although statistically significant may not necessarily be meaningful, as noted previously due to the possibility that the data for the control groups and the experimental groups have a different sample population distribution, and alone it does not respond to whether the result is a manifestation brought on by some external effect. Hence, some longer term investigation is required and this concern is addressed by the case studies.

5.8.11 Confounding Variables

The literature survey found a number of programmer attributes that researchers have claimed would affect the readability of source code and that these programmer attributes could confound the interpretation of results if they were not measured and addressed when analysing the data. These variables are discussed further with reference to the research and are presented in Table 5-20. The table reports the confounding variable's title; qualifies the observations made by the relevant researchers; and reports whether the research supports the researcher's observations.

Table 5-20 - Confounding Variable Support Findings

Id	Confounding Variable Title	Observations	Support
1	Professional Programmer	Detienne (2001) states that novice programmers do not give proper consideration to naming identifiers, which often results in poor identifier naming. The research has found a 10% difference in the presence of meaningful identifier names devised by experienced professional programmers than those identifier names devised by student programmers.	Yes
2	Years Programming	Li and Prasad (2005) infer that software quality practices that are introduced to student programmers may be flawed. The research has found that the percentage of meaningful identifier names generated by student programmers decreases steadily during their instruction by as much as 20% and approximately a decade of experience is required to recover to their initial level.	Yes
3	Program Size	Ramanujan, Scamel and Shah (2000) found that the absence of meaningful identifier names did not affect the maintainability of small computer programs but the absence of meaningful identifier names did affect the maintainability of large software systems. The research found that as a programmer's experience increased, so too did the size of the largest computer program generated by them to date. Initially, the percentage of meaningful identifier names decreased but this may have been due more to the relative level of programming experience than due to the actual size of the largest computer program generated at that stage of their development. As expected, as further experience	Yes

Table 5-20 - Confounding Variable Support Findings

Id	Confounding Variable Title	Observations	Support
		was obtained by the programmer the size of their largest computer program increased and so too did the percentage of meaningful identifier names in their source code. A more proper evaluation of this confounding variable awaits the collection of data from larger computer programs than have been generated by programmers who have closer to a decade's programming experience.	
4	Subjected to Source Code Review	Rees (1982) found that students who were subjected to having their source code reviewed acted to improve their identifier naming practices. The research found a 3% difference in the percentage of meaningful identifier names in the source code that had been written by a programmer who had had their source code reviewed over those programmers who had never had their source code reviewed. The research supports this confounding variable with low statistical significance.	Yes
5	Source Code Reviewer	Land, Sauer and Jeffery (1997) conducted empirical research which indicated that experienced programmers do not perform substantially better than student programmers at detecting software defects. The research found a 1% difference in the percentage of meaningful identifier names in the source code that had been written by programmers who had reviewed source code previously over those programmers who had never reviewed another's source code. As the result was not statistically significant, support for this confounding	Indeterminate

Table 5-20 - Confounding Variable Support Findings

Id	Confounding Variable Title	Observations	Support
		variable is not provided by this research.	
6	Touch Type	Alsio and Goldstein (2000) have shown that skilled touch typists use negligible cognitive resources compared to a programmer who must use hunt-and-peck to use a keyboard. Hence, it would be expected that the cognitive resources consumed by the programmer who must use hunt-and-peck to use a keyboard would be available to a programmer who was a skilled typist, possibly for the generation of meaningful identifier names. The research found a 12% difference in the percentage of meaningful identifier names in the source code that had been written by programmers who identified themselves as touch typists.	Yes
7	Identifier-Naming Effort	Neumann (2002) observes that programmers do not care whether they produce a quality product. The research tested this statement and found that some programmers do care about the quality of their identifier naming, where as others appear not to care as judged by the names chosen for their identifiers during the maintenance and production experiment. The research found a 29% increase in the percentage of meaningful identifier names in the source code that had been written by programmers who identified that they place “much more” effort into identifier-naming than those programmers who identified that they place “much less” effort into identifier-naming.	No

Table 5-20 - Confounding Variable Support Findings

Id	Confounding Variable Title	Observations	Support
8	Flexible Work Practice	Boehm and Basili (2001) observes that programmers behave as though the only thing of matter is the initial generation of source code and Glass (2001) concludes that in our current culture, programmers will ignore quality in the pursuit of a working product. The research tested these statements by requesting the test subjects to potentially modify their normal work practices and found that some programmers are flexible in their work practices in that they responded favourably to the dynamic reporting of identifier-naming style flaws. Conversely other programmers partially ignored the dynamic reporting of flaws or in some cases totally ignored the dynamic reporting and generated source code containing meaningless identifier names. The research found a 22% increase in the percentage of meaningful identifier names in the source code that had been written by programmers who identified that they are "much more" flexible in their work practices than those programmers who identified that they were "much less" flexible in their work practices.	No

5.9 Novice Programmer Case Study

The software development project that was the subject of the Novice Programmer case study was funded as a corporate research and development activity within a Defence organisation. The research and development activity was ultimately responsible for the development of the Computer Architecture Topography Simulator (CATS). The CATS employs a number of data entry and dynamic graphical display windows, maintains internal state transitions, supports multiple internal message queues, supports database access and contains a number of complex scheduling algorithms. The nature of the CATS programming task, although at best medium sized by industry standards, required the programming considerations typical of large industrial sized computer programs. The Novice Programmer case study was supported by a 108 page

Software Requirements Specification (SRS) and a 196 page Software Design Description (SDD) both of which were written by the author. A newly graduated software engineer, who had not previously written software as a professional, was chosen as the test subject for the Novice Programmer case study. The Novice Programmer case study resulted in the production of 767 SLOC over four software units (i.e., files), before corporate funding was suspended in order to support higher priority tasks.

5.9.1 Novice Programmer Comments

The novice programmer was positive in their attitude towards the reporting of identifier-naming style flaws and noted that they found the highlighting of poorly named identifiers to be useful to their software development activity. The novice programmer did state that they did not perceive the reporting generated by the source code editor to be threatening or unsettling. The novice programmer was cognisant of their limited experience and reported that they would consider the use the identifier-naming style flaw reporting capability, offered by the source code editor, during their future software development activities as the capability found issues that they had not been able to identify by themselves. In particular, the novice programmer stated that they did not always accept the reporting of the Un-named Constant (01), Short Name (05), Word Count (07), and that they refused to accept the reporting of the Class/Type Qualification (09) identifier-naming style guidelines. This result is consistent with the attitude of novice programmers as reported by the Identifier-Naming Style Guideline Programmer Acceptance questionnaire for the Short Name (05), and Class/Type Qualification (09) identifier-naming style guidelines, and the actual practice of novice programmers, particularly for the Un-named Constant (01) and Word Count (07) identifier-naming style guidelines. The novice programmer may accept that the Un-named Constant (01) identifier-naming style guideline directs towards improved source code readability but as seen during the maintenance and production experiment, the novice programmer was reluctant to define a named constant, unless explicitly directed to do so. In addition, novice programmers, on occasion tend to be terse in the naming of identifiers, resulting in the generation of single word identifier names. Hence, the novice programmer's reluctance to necessarily accept the Word Count (07) identifier-naming style guideline as directing towards improved source code readability.

The novice programmer further noted that the source code editor ran slowly. The execution speed of the source code editor did appear to be an issue for the novice programmer as the novice programmer preferred to use the Java IDE to modify identifier names in preference to allowing the source code editor to make the appropriate identifier name replacements. Attempts

were made to speed the response performance of the source code editor. However, the source code editor continued to run slowly, due to the combinational checking against identifier names, and the Java IDE remained the editor of choice for the novice programmer.

5.9.2 Source Code Inspection

As the source code generated during the Novice Programmer case study was to be part of a much larger corporate project the source code underwent source code inspection. The source code inspection activity occurred after the novice programmer had used the source code editor to report any identifier-naming style flaws and had corrected these flaws, as appropriate. Table 5-21 summarises the source code inspection review comments for each of the software units coded.

Table 5-21 - Novice Programmer Source Code Inspection Review Comments Summary

Software Unit Id	Comments
1	One comment unrelated to identifier naming
2	Eleven comments unrelated to identifier naming
3	Eleven comments unrelated to identifier naming Four comments related to identifier naming
4	No review comments were generated

The review comments regarding identifier naming all related to the choice of identifier name not being indicative of the actual function that the identifier was used for, which is outside of the scope of the identifier-naming style guidelines used by this research. By way of an example, the novice programmer defined a character constant identifier, to be used in the discovery of a delimiting character during a text parsing operation, with the name "COMMA_CHARACTER". The novice programmer had considerable difficult in understanding that this was a poor name, akin to naming an integer constant, that holds the value "10", with the name "TEN" (i.e., that is what it is but not what it means). The source code was not further analysed after the novice programmer had responded to the source code inspection comments as any changes to an identifier name would most likely be due to the review comments and not necessarily to the reporting of an identifier-naming style flaw.

5.9.3 Source Code Effects

The effect of reporting identifier-naming style flaws to the novice programmer was to reduce the total number of flaws in each of the four software units. The instances where the effect of identifier-naming style reporting resulted in an increase in the number of specific identifier-naming style flaws after editing was for the Long Name (06) flaw with seven instances distributed over two software units, English Words (12) flaw with one instance and Same Words (19) flaw with one instance.

The increase in Long Name (06) identifier-naming style flaws being reported was due to the novice programmer choosing to identify the relationship between a number of related constant identifier declarations by qualifying the applicable identifier names with a standard word phrase to show this common relationship. As a consequence, the identifier names were longer but they were also more meaningful to the actual use of the identifier.

The instance of the English Word (12) identifier-naming style flaw was due to the definition of an identifier with the name "CHARACTER_TYPE_NEWLINE". The character string "NEWLINE" is not an English word and the identifier name was reported as an identifier-naming style flaw. However, the novice programmer failed to act on the reporting as the novice programmer assumed that the reporting was due to the length of the identifier name which was responsible for the reporting of an identifier-naming style flaw and not some other issue. In this case the reporting of the Long Name (06) identifier-naming style flaw resulted in an additional flaw being overlooked by the novice programmer.

The instance of the Same Words (19) identifier-naming style flaw was due to a failure of the novice programmer to devise a meaningful name either for the method that counts data fields, which was named "dataFieldCount" and the variable that was used to count the data fields, which was named "countDataField". Such similarity of naming between the method name and the variable name would cause issue to a maintenance programmer, particularly as the variable was declared global to the method. The novice programmer, when questioned, stated that they could not think of a better name for the identifiers.

In all other cases the novice programmer edited the source code to reduce the number of incidents where identifier-naming style flaws were reported. This was achieved by the replacement of generic or ambiguous identifier names by more specific identifier names (e.g. "fr" was replaced with "fileReadData"). In particular, the following reductions in identifier-naming style flaws were recorded: Un-named Constant (01) with 15 occurrences reducing to 4,

Short Name (05) with 32 occurrences reducing to 10, Word Count (07) with 27 occurrences reducing to 17 and Plural Word (14) with 2 occurrences reducing to 0. In all cases reporting of identifier-naming style flaws was instrumental in reducing, if not eliminating, a number of flaws from the source code being edited and particularly in increasing the number of meaningful identifier names.

The novice programmer had not memorised the identifier-naming style guidelines or was unable to apply them from memory, as the novice programmer recreated the same identifier-naming style flaws that they had previously made and subsequently modified the relevant identifier name to remove the flaw under the suggestion offered by the source code editor. This occurred on multiple occasions within a software unit. In the case of the Word Count (07) identifier-naming style flaws, the flaws were removed in all four software units and in the case of the English Words (12) and Duplicate Names (16) flaws, the corresponding flaws were removed in three of the four software units.

5.10 Programming Team Case Study

The Programming Team case study capitalised on the re-start of the CATS project which was the subject of the Novice Programmer case study. The Programming Team case study was supported by eight programmers with differing duration of industry experience. Over a four month period, the design documented within the SDD was validated against the SRS, Unified Modelling Language (UML) diagrams inserted and functionality added (e.g. an object messaging facility and a code execution logging facility), which resulting in the SDD increased to 271 pages. The code and unit test, software integration and software system test project activities ran for ten months, and resulted in the development of 15k SLOC distributed over 125 software units.

The project was initially staffed by two programmers. One programmer had ten years industry programming experience and the other had fifteen years industry programming experience. Three months into the project a third programmer, with twenty-four years programming experience, was assigned to work on the simulation management component of the project. One month later, the two original programmers left the company and were consequently replaced by a further two programmers, one with four years industry programming experience and the other had eight years industry programming experience. Subsequently, three ‘free’ resources were made available to support the project. Two of these ‘free’ resources had four years industry programming experience and the other had thirteen years of industry programming experience.

With the addition of these extra staff the most experienced programmer was tasked to act as team leader and hence was not expected to directly contribute to any further software development.

5.10.1 Source Code Editor Usage

The programmers made few comments regarding the source code editor; except that the first two programmers said that they ignored the reporting of the identifier-naming style flaws. Hence no data was available against these two programmers. The programmers were unanimous in their disinterest to use the source code editor to actually edit their source code, preferring to use the editing environment that they were most familiar with. However, the second set of programmers did paste their source code into a source code editor edit window to gain access to the identifier-naming style flaw reporting capability and then they would use their preferred editing environment to effect change to the source code.

As the programmers had previously developed an understanding for the importance of meaningful identifier names from working in the Defence industry, their general acceptance of the identifier-naming style flaw reporting and subsequent correction of identifier names presented few disputes against the identifier-naming style guidelines generally accepted by expert programmers. One of the programmers stated that they had learnt to predict the issues that the identifier-naming style guidelines, implemented by the source code editor, would report and had modified their identifier naming practices as appropriate to minimise the reporting of a corresponding flaw.

5.10.2 Source Code Inspection

As with the Novice Programmer case study, source code inspection occurred after the programmers had used the source code editor to report any identifier-naming style flaws. Table 5-22 reports the total number of review comments generated for each of the programmers who employed the reporting of identifier-naming style flaws to improve the readability of their source code. The programmers are identified by an Id to protect their privacy. For each programmer, their Id, the number of years programming experience in a mission critical software development facility, the number of SLOC that they were responsible for generating, the number of review comments relating to identifier names and the total number of review comments generated has been reported.

Table 5-22 - Programming Team Source Code Inspection Review Comments

Programmer Id	Years Programming	SLOC	Review Comments	
			Naming	Total
A	4	2,667	18	35
B	4	388	0	5
C	4	7,329	15	54
D	8	545	11	31
E	13	2,506	7	31
F	24	1,710	12	76

The small number of review comments per SLOC was due to the software design limiting the size of the constituent methods to understandable sizes, and was also due to the software testing, mandated by the SRS, which effectively eliminated most coding issues. The review comments that were generated, that addressed identifier naming, included comment on the inappropriate use of abbreviations, inconsistent naming of related identifiers and the general meaningfulness of identifier names.

5.10.3 Reporting Effects on Source Code

Six programmers used identifier-naming style flaw reporting during the Programming Team case study to assist them in the improvement of their source code's readability. Figure 5-4 depicts the percentage reduction in identifier-naming style flaws reported between the programmer's penultimate and their last edit for a software unit that they checked into the configuration management repository used to save their work. The last software unit, checked into the configuration management repository, was generated by editing the penultimate software unit as a consequence of the reporting and addressing of identifier-naming style flaws. The figure individually graphs data from the six programmers and labels this data with the number of years that the programmer has worked as a professional programmer. The value graphed corresponds to the delta reduction of identifier-naming style flaws from the penultimate to the last edit of the software unit cumulatively totalled over all software units generated thus far as of the relevant software unit number.

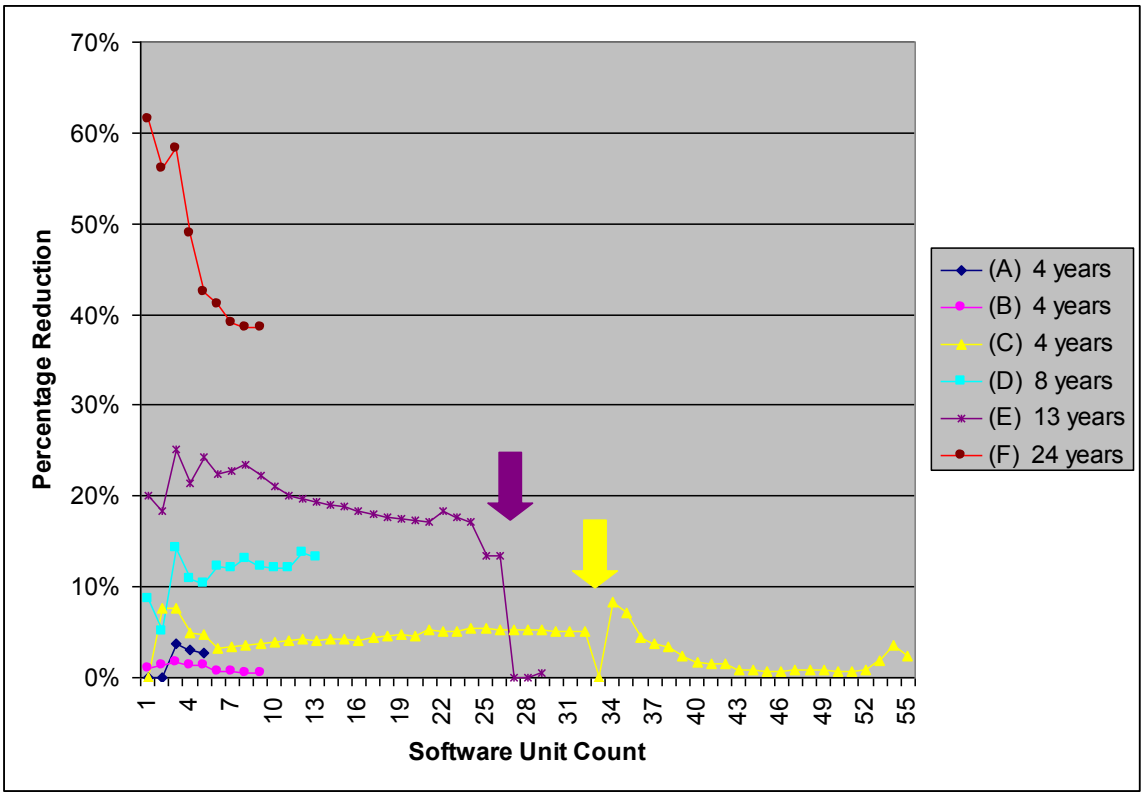

Figure 5-4 - Identifier-Naming Style Flaw Reduction Comparison between Programmers

As can be seen from the graphical representation of the data, the delta reduction in the percentage of identifier-naming style flaws remains as a positive percentage value against all instances where reporting of flaws has resulted in a modified software unit i.e., there are no instances where an increase of flaws occurs as a consequence of the final edit. What was also expected was that the number of identifier-naming style flaws being removed would reduce over time. This trend is apparent in the figure with the steady decline apparent in most instances.

Referring to the legend in Figure 5-4, programmers (C), (D) and (E) first tested source code that had been previously written by the original two programmers assigned to the CATS project. Programmers (C) and (E) then wrote and tested their own source code, with the transition point identified by the step in the lines corresponding to software units 33 and 27 respectively, as indicated by the arrows on the figure. Such a step would be expected as two programmers seldom employ the same naming rules for identifiers and their personal identifier-naming practices would be expected to generate a different distribution of identifier-naming style flaws. For all instances, where the programmer used dynamic reporting of identifier-naming style flaws to correct their own source code, the slope of the line graphing the percentage reduction of identifier-naming style flaws tends to decrease with the next software unit written. The slope of

the lines represented in the figure correspond to -0.51 for programmer (A), -0.12 for programmer (B), -0.18 for programmer (C), -0.63 for programmer (E) and -3.18 for programmer (F). This decrease in line slope would be expected if the programmers were learning from exposure to the reporting of identifier-naming style flaws on their source code before they wrote the next software unit. However, such a decrease would also be seen if the programmers became more complacent to the reporting of identifier-naming style flaws and began to ignore the reporting of flaws.

On direct questioning the programmer's responses to the source code review comments indicated that they were learning to anticipate the identifier-naming style flaw reporting that would result from their choice of identifier name. Programmer (D) responded with the following observations to one of the source code review comments made against their work:

> … "Yes, it's only taken me 4 or 5 files and I'm already trying to do things to appease the almight[y] editor with it's terrifying underlines …"

Accepting that the value of the identifier-naming style flaw reporting was diminishing slightly with each software unit completed, the low mean percentage reductions of identifier-naming style flaws apparent for the three programmers who had four years professional programming experience was initially unexpected, in that it was expected that the reporting of identifier-naming style flaws would have offered a greater reduction in the number of flaws than that seen. However, on conversation with these programmers, reasons for the low mean percentage reductions of identifier-naming style flaws were uncovered. Programmers (A) and (B) have been classified by their team leader and also by their software functional manager as very much in the realm of the novice programmer. Detienne (2001, p35) observes that novice programmers favour a retrospective bottom-up approach to software development. This method can result in the novice programmer iterating through a number of code/test cycles before releasing the source code. Programmers (A) and (B) both confirmed this general approach and also confirmed that they used the identifier-naming style flaw reporting during this cycle to improve the readability of their source code and hence data was only captured from their very last cycle, which artificially reduced the apparent percentage reductions of identifier-naming style flaws recorded.

Programmer (C) developed their own source code from and including software unit number 33. Programmer (C) was very confident in their coding and generated fewer identifier-naming style flaws than would be expected of an expert programmer, hence the low mean percentage reductions of identifier-naming style flaws of 2.2% for the source code that they were responsible in generating. The source code that they inherited from a previous programmer,

with the exception of the first five software units, contained few instances of identifier-naming style flaws and the reporting of these flaws was not generally useful to programmer (C) as the source code was considered adequate as is by programmer (C) and hence little change was required.

Programmer (D) did not write any of their own source code and so the mean percentage reductions of identifier-naming style flaws is not meaningful in this situation as programmer (D) had no opportunity to apply learning to the initial allocation of the identifier names in the source code that they tested.

Programmer (E) developed their own source code from and including software unit number 27. Programmer (E) was an expert programmer who was careful in their derivation of identifier names. Hence the low mean percentage reductions of identifier-naming style flaws of 0.2% for the source code that they were responsible for generating. However, the source code that they inherited from a previous programmer contained numerous instances of identifier-naming style flaws and the reporting of these flaws was self-reported as useful to programmer (E) in identifying these instances for subsequent remedial action.

Programmer (F) was an expert programmer who also inherited the team leader responsibility when the original two programmers left the CATS project and was required to establish the new software development team. Programmer (F) also appeared to have some difficulty in expressing themselves in written English, predominantly as English is not their first Natural language. The reporting of identifier-naming style flaws was useful to programmer (F) to identify the existence of Abstract Words (11) identifier-naming style flaws which occurred in four of their nine software units and the existence of identifier names that were considered either too short or too long which occurred in eight of their nine software units.

5.11 Identifier-Naming Style Guideline Support

As a consequence of conducting the research the background theory addressing the identifier-naming style guidelines is discussed and appropriate modifications to the background theory are presented in Table 5-23. The table lists the relevant identifier-naming style guideline Id for which the background theory has been modified; presents the relevant background theory; presents a discussion relevant to the identification of support or detraction from the background theory as a consequence of the research; identifies the previous justification for the background theory; and identifies whether there is support for the background theory that is claimed by the

research. The discussion presented in the table identifies the specific researchers whose work was used as a basis for the identifier-naming style guideline. The discussion then concludes with a summary of the results obtained by the research.

Table 5-23 - Background Theory Support Findings

Id	Discussion	Background Theory	Support
01	Weissman (1974) and Fang (2001) suggest that 'magic numbers' should be declared as named constants.	Opinion	Yes
	The research has found that the practice of expert programmers was to accept this guideline with weak statistical significance and that this trend increases as the programmer gains experience.		
02	Wells (1986) and Oualline (1992) differ in their opinion as to whether identifier names should contain the underscore character or not, and justify their prejudice by stating that their stance is based on their personal taste.	Opinion	Indeterminate
	The research method was not defined with the intent to discover whether the use of an underscore character in separating words effects source code readability and so no determination on the background theory is possible using the results of the research. However Matis' (1996) survey of the literature found that reading speed is significantly reduced by not using a separation character (i.e., the blank or space character).		
03	Horton (1998, p12) dissuades against the use of leading underscore characters.	Opinion	No
	The research has found that the use of a trailing underscore character was erroneously ignored by novice test subjects during the maintenance and production experiment, making the value of a leading or trailing underscore character questionable in support of improved source code readability. However, the research has found that the practice is sufficiently infrequent as to severally limit the value of the corresponding guideline in the practical sense.		
04	McConnel (1993, p210) suggests that numeric characters in an identifier name damages software quality.	Opinion	Yes

Table 5-23 - Background Theory Support Findings

Id	Discussion	Background Theory	Support
	The research has found that the practice of expert programmers was to accept this statement with strong statistical significance.		
05	Rotenstreich (1988) suggests that loop variables should be single character identifier names.	Opinion	No
	Gorla, Benander and Benander (1990) found that identifier name length should be at least 8 characters long.	Empirical	Yes
	The research has found that the practice of expert programmers was to accept this guideline with moderate statistical significance and that this trend increases as the programmer gains experience. The research also found that a single character identifier name can result in the declaration being hidden from the programmer and that software integration testing can be hampered when single character identifier names are used to index multiple dimensioned arrays. Hence, the guideline requires modification to remove the acceptance of single character identifier names as exceptions that may be applied to the guideline.		
06	Gorla, Benander and Benander (1990) found that identifier name length should be at most 20 characters long.	Empirical	No
	The research has found that the practise of expert programmers was to accept this guideline with weak statistical significance but that this trend decreases as the programmer gains experience.		
07	Laitinen and Mukari (1992) suggest that identifier names should be composed of from two to four words.	Opinion	Yes
	The research has found that the practice of expert programmers was to accept this guideline with poor statistical significance and that acceptance increases as the programmer gains experience.		

Table 5-23 - Background Theory Support Findings

Id	Discussion	Background Theory	Support
08	Mc Connell (1993, p206) suggests that Hungarian notation should be avoided.	Opinion	Yes
	The research has found that the practice of expert programmers was to accept this guideline with moderate statistical significance.		
09	McConnel (1993, p198) and Vermeulen et al. (2000, p18) suggest that the practice of differentiating identifier names that have the same name by capitalisation only, should be avoided.	Opinion	No
	The research has found that the practice of capitalisation of class names and the creation of objects with the same name and capitalisation but with the first letter in lowercase is stronger than the programmer's willingness to use a different identifier-naming convention. The research has found that the practice of expert programmers was to reject this guideline with strong statistical significance.		
10	McConnel (1993, p198) suggests that identifier qualifiers should be placed at the end of the identifier name.	Opinion	Yes
	The research has found that the practice of expert programmers was to accept this guideline with strong statistical significance and that this trend increases as the programmer gains experience.		
11	McConnell (1993, p80) suggests identifiers should not be composed entirely of abstract words.	Opinion	Yes
	The research has found that the practice of expert programmers was to accept this guideline with moderate statistical significance.		
12	Laitinen and Mukari (1992) suggest that the use of abbreviations should be avoided.	Opinion	Yes
	McConnel (1993, p210) suggests that words in identifier names should not be miss-spelt.	Opinion	Yes

Table 5-23 - Background Theory Support Findings

Id	Discussion	Background Theory	Support
	Laitinen and Mukari (1992), Caprile and Tonella (2000), and Vermeulen et al. (2000, p17) suggest that the use of natural language words should be used in the construction of identifier names.	Opinion	Yes
	The research has found that the practice of expert programmers was to accept this guideline with strong statistical significance and that this trend increases as the programmer gains experience.		
13	McConnel (1993, p196) suggests that identifier names should not be composed only of words or digits that are used to represent numeric values.	Opinion	No
	The research has found that the practice is sufficiently infrequent as to severally limit the value of the corresponding guideline in the practical sense.		
14	Meyer (1988) notes that there is potential for confusion when one identifier differs from another by one character. Expecting a programmer to recall whether an identifier has been declared in the plural or singular form adds cognitive overhead, which can be alleviated by declaring all identifiers in the singular.	Opinion	Yes
	The research has found that the practice of expert programmers was to accept this statement with poor statistical significance.		
15	Booch (1994, p164) suggests that an identifier naming convention should be consistently applied.	Opinion	Yes
	The research has found that the practice of expert programmers was to accept this guideline with poor statistical significance.		
16	Weissman (1974), and Deimel and Naveda (1990) suggest that identifier names should not be overloaded.	Opinion	Yes
	Chan and Yang (2002) found that the overloading of	Empirical	Yes

Table 5-23 - Background Theory Support Findings

Id	Discussion	Background Theory	Support
	identifier names frustrates the programmer in their ability to readily understand the source code.		
	The research has found that the practice of expert programmers was to accept this guideline with strong statistical significance, except as previously reported for Id 09, where an instance of an object is created from the class.		
17	Deimel and Naveda (1990), McConnell (1993, p209) and McCall (2004) suggest that identifier names should not differ by two characters or less.	Opinion	Yes
	The research has found that the practice of expert programmers was to accept this statement with strong statistical significance.		
18	Wells, Brand and Markosian (1995) suggest that unused identifiers should not be left in source code.	Opinion	Yes
	The research has found that the practice of expert programmers was to accept this statement with moderate statistical significance.		
19	McConnell (1993, p209) suggests that different identifiers that are composed of the same words but in a different order should be avoided.	Opinion	No
	The research has found that the practice is sufficiently infrequent as to severely limit the value of the corresponding guideline in the practical sense.		

In addition to researcher's theories that directly relate to the identifier-naming style guidelines, the following researcher's work has been supported by the research reported in this thesis:

- Furnas (1987) cites multiple empirical studies that refute the idea that there is an obvious and a self-evident single word naming any function or object. The research found a multitude of identifier names used to name the variable that was used to hold a time index, which was part of the introduction exercise of the maintenance and production experiment. What was considered to be an obvious identifier name by the researcher was not obvious or self-evident to most of the test subjects. This and other results, which address the choice of identifier names devised during the maintenance

and production experiment, support the conclusion held by the researchers cited by Furnas (1987).

In addition to researcher's theories that directly relate to the identifier-naming style guidelines, the following researcher's work has not been supported by the research reported in this thesis:

- DeYoung and Kampen (1979) claimed that their predicted readability metric could compare an examiner's solution to the student's work and present a relative score of source code readability. The research found that the predicted readability metric had at best poor correlation against the percentage of meaningful identifier names in the test subject's work as adjudicated by an expert programmer. The comparison may suffer from the use of a trivially small software unit, which was the subject of the introduction exercise in the maintenance and production experiment. However trivially small software units are found in mission critical software and an expectation exists that the metric would be useful in this situation too. Hence, the work undertaken by DeYoung and Kampen (1979) is not supported by the research.
- Similarly, the work undertaken by Pearse and Oman (1995) in the definition of their maintainability index metric shows poor correlation at best against the percentage of meaningful identifier names in the test subject's work as adjudicated by an expert programmer. Hence, the work undertaken by Pearse and Oman (1995) is not supported by the research.

5.12 Chapter Summary

Reality is that which, when you stop believing in it, doesn't go away.
Phillip K. Dick

In summary, the investigation and results are characterised by the following:

- The results from the Identifier-Naming Style Guideline Programmer Acceptance questionnaire were investigated. The analysis support the following observations:
    - Novice programmers were not sure whether the Un-named (01), Numeric Name (13) or Similar Names (17) identifier-naming style guidelines were relevant, either for or against, to improved source code readability; intermediate programmers were not sure whether the Word Count (07) guideline was relevant; and expert programmers were sure that all the guidelines were relevant one way or the other.

    - Novice programmers did not accept that the Outside Underscore (03), Short Name (05), Long Name (06) or Class/Type Qualification (09) identifier-naming style guidelines actually direct towards improved source code readability; intermediate programmers did not accept that the Short Name (05), Long Name (06), Class/Type Qualification (09) or Unused Identifier (18) identifier-naming style guidelines actually direct towards improved source code readability; and expert programmers did not accept that the Class/Type Qualification (09) identifier-naming style guidelines actually direct towards improved source code readability.
- A small sample of dated textbooks was surveyed and in all cases, for all identifier-naming style flaws, with the exception of one instance in one of the textbooks, there was no support for the identifier-naming style guidelines.
- Dated software from a single individual was surveyed and the percentage of identifier-naming style flaws was generally found to decrease or remain the same as experience was gained by the programmer. However, the Numeric Digit(s) (04), Long Name (06), Duplicate Names (16) and Similar Names (17) identifier-naming style flaws were found to be increasingly rejected as the programmer gained experience.
- Contemporary software was surveyed and few instances of the Multiple Underscore (02), Outside Underscore (03), Numeric Name (13) and Same Words (19) identifier-naming style flaw were found. In addition, the Naming Convention (15) identifier-naming style guideline was found not to be useful for Ada source code due to the large range of complexity in the naming conventions used within the Ada source code. About half of the identifier names raised the Short Name (05) identifier-naming style flaw, hence making the reporting of this flaw potentially the most useful to programmers in general.
- The source code editor capability was introduced and in addition, the effort undertaken to ensure the maintenance and production experiment could be worked by the test subjects was discussed.
- The Programmer Characteristics questionnaire, in conjunction with the results from the maintenance and production experiment, showed agreement with other researcher's results relative to the percentage of meaningful identifier names present in the source code. What the questionnaire did show was that a programmer develops poor identifier naming practices early on which then requires approximately ten years of experience to recover to the same level of meaningful identifier name usage as that generated by a first year student programmer. However, programmers can improve past that of the ability of the first year student as they gain further experience.

- The maintenance and production experiment was investigated:
    - Four expert programmers were used to adjudicate on a list of identifier names generated by the test subjects during the introduction exercise, to identify which names were meaningful for the purpose of the identifier. The resultant name list was different in each case but the use of any list was found to consistently identify poor identifier naming practice.
    - Test subjects were extremely diverse in their choice of names for the same identifiers.
    - Novice programmers showed considerable difficulty in correctly maintaining even simple source code.
- The use of an automated tool to report identifier-naming style flaws has the potential to save expert programmer's time during a code review.
- The identifier-naming style guidelines are capable of aiding the novice programmer and the experienced programmer in the development of meaningful identifier names.
- The background theory has been modified as a consequence of conducting this research and a discussion describing this modification to the background theory was presented.

6 Conclusions

This chapter presents a summary of findings, identifies the limitations in the current research, brings the research results together and suggests what future work is now appropriate. The research limitations are discussed with reference to the specific identifier-naming style guidelines used by this research and are also discussed in terms of the limitations of the research tools (i.e., questionnaires, surveys, source code editor, experiment and case studies). The research contribution is discussed in terms of the support that it offers to the foreground theory. The effects of the confounding variables to the research results are discussed with reference to the background theory. In addition, the research hypothesis is decomposed and discussed in terms of the research results. Finally, work that is now relevant to furthering the research is presented for consideration.

6.1 Summary of Findings

The research discussed in the literature review found that source code readability is dependent on many attributes of the source code. One attribute, that the programmer has extensive control over, that affects source code readability is the naming of an identifier. A programmer may have control over the naming of an identifier but this does not mean that they are effective in their control. Programmers find the simultaneous activity of coding and the devising of meaningful identifier names that are readable by other programmers to be a difficult task. This difficulty is due in part to current educational practices and the programming cultural norms, and is compounded by the cognitive limitations of the programmers in their attempts to devise meaningful identifier names. The research has found evidence in support of identifier-naming importance to source code maintenance and has found support for some but not all of the identifier-naming style guidelines investigated.

6.1.1 Importance of Identifier Naming

Identifier naming is relatively unimportant for the development of small software systems, as identifiers can be treated, by the programmers who use them, simply as place holders to store data. However, there are instances where identifier-naming is important for small computer programs as well as for large software systems. Single character identifier names were shown to be an issue for the test subjects during the maintenance exercise of the maintenance and

production experiment, where the identifier declaration was effectively camouflaged from the test subject's view and the test subject was not able to 'see' the scope of the identifier declaration. Hence an unused identifier was left in the source code, contrary to the maintenance activity instructions to the test subjects that this declaration should be removed. The use of single character identifier names was also an issue for the novice programmer during the Novice Programmer case study, where the novice programmer was confused by the incorrect order of indexing within a multi-dimensional array and as a consequence their testing of the software unit was protracted. It was only when the novice programmer replaced their identifiers "i" and "j" with meaningful identifier names could they identify how the source code was incorrect.

The maintenance exercise of the maintenance and production experiment demonstrated a high failure rate of 86% within the student group and 30% within the professional programmers group for one of the maintenance activities. This result occurred due to the inconsistent use of named constants and numeric literals. The same experiment found that if only numeric literals were used, the more acceptable failure rates of 6% within the student group and 0% within the professional programmers were seen. The maintenance difficulties demonstrated by the test subjects were unexpected, for the first result as the maintenance activities were trivial in comparison to real-world maintenance activities. The design of the maintenance and production experiment did not extend to considering the use of named constants only, as the a high failure rate in the first result was unexpected. Hence there is no data for an equivalent maintenance action where only identifier names were used as the subjects of the selection statement case labels and also the subject of the associated assignment statement for each case label. Hence, the maintenance and production experiment design did not define a third maintenance activity allowing for comparison between the three maintenance actions. Even though this comparison is not possible, the result from the first maintenance activity demonstrates the importance of the consistent use of identifier names to the maintenance activity.

6.1.2 Identifier-Naming Style Guideline Summary

Discovering the characteristics of a meaningful identifier name proved to be problematic in that the absence of significant research in this area meant few identifier-naming style guidelines were able to be found. Those identifier-naming style guidelines that were found during the literature review are listed in Appendix A – Identifier-Naming Style Guidelines. However, not all of the identifier-naming style guidelines proved to be useful to software engineering practice. Undertaking the literature survey found an incomplete set of identifier-naming style guidelines that had been identified as being relevant to source code readability. At best there is empirical

research supporting some but not all aspects of source code readability that the identifier-naming style guidelines offer suggestion for. However, typically there is only the assertion, by researchers, that the identifier-naming style guidelines direct towards improved source code readability.

This research has found that the identifier-naming style guidelines are generally regarded by expert programmers as being relevant to source code readability and that eighteen of the guidelines were accepted by expert programmers as actually directing towards improved source code readability at least for the examples presented. The Class/Type Qualification (09) identifier-naming style guideline was rejected by the expert programmers as not necessarily directing towards improving source code readability. The Long Name (06) identifier-naming style guideline was accepted by the expert programmers as directing towards improved source code readability but as the programmers gained experience, the tendency to ignore the guideline increased in frequency. The identifier-naming style guidelines considered relevant to source code readability by expert programmers were surveyed within production software and the Multiple Underscore (02), Outside Underscore (03), Numeric Name (13) and Same Words (19) identifier-naming style flaws were found to rarely survive into production software and hence are of questionable value to software engineering practice.

In summary, the following identifier-naming style flaws, that are considered most appropriate to report to programmers in order to improve source code readability, are listed below:

Un-named Constant (01) – Programmers should be dissuaded from writing source code with embedded literals, with the possible exceptions of the literals minus one, zero and one.

Numeric Digit(s) (04) – Programmers should be dissuaded from declaring identifiers with names that contain numeric digits.

Short Name (05) – Programmers should be dissuaded from declaring identifiers with names composed of less than eight characters. Hence, single character generic identifier names are explicitly excluded from the set of acceptable identifier names.

Word Count (07) – Programmers should be dissuaded from declaring identifiers which are composed of a single acronym/word or are composed of a combined total of more than four acronyms/words.

Identifier Encoding (08) – Programmers should be dissuaded from declaring identifiers that contain type information.

Constant/Variable Qualification (10) – Programmers should be encouraged to define constant and variable identifiers with an identifier qualifications concatenated to the end of the identifier name.

Abstract Words (11) – Programmers should be dissuaded from declaring identifiers that are composed entirely of abstract words.

English Words (12) – Programmers should be dissuaded from declaring identifiers which contain acronym(s) that are not project sanctioned or do not belong to a relevant natural language. Hence, abbreviations are specifically excluded from the set of acceptable identifier names.

Plural Words (14) – Programmers should be dissuaded from declaring identifiers that contain word(s) that are in the plural when a singular form of the word is available.

Naming Convention (15) – Multiple empirical studies have found that 'news paper headlines' formatting can be read faster than other formatting. Hence, programmers should be dissuaded from declaring identifier names that do not separate the component acronym(s) and word(s) by the use of a separation character and have the first letter only, of all component word(s) capitalised. However, despite empirical research supporting this naming convention for improved readability, the expert programmers showed strong support for the Java naming convention, which deviates from the suggested naming convention.

Duplicate Names (16) – Programmers should be dissuaded from declaring identifiers that are duplicates of another identifier that are in the same programming scope.

Similar Names (17) – Programmers should be dissuaded from declaring identifiers that differ by one or two characters from another identifier that is in the same programming scope.

Unused Identifier (18) – Programmers should be dissuaded from writing source code with identifiers that are declared but not used. If the identifiers are not used the identifier declaration should be deleted from the source code.

6.1.3 Temporal Effects on Identifier-Naming Style

Programmers arguably require time to amass experience before they can recognise quality attributes in their software. As expected, both the novice and the intermediate programmers accepted fewer of the identifier-naming style guidelines than the expert programmers, corresponding to 80%, 80% and 95% respectively.

A survey of programming textbooks was undertaken and there appears to be almost no support for the identifier-naming style guidelines considered by this research within the programming textbooks used by programmers. Accepted that only seven programming textbooks published over a 30 years period were surveyed in detail but there is nothing in the text to suggest that

these textbooks are different from the normal population of programming textbooks. Hence, it was not surprising to see a gradual drop in the percentage of meaningful identifier names used during the programmer's first three years of software engineering study, with the percentages of meaningful identifier names dropping from 30% to 20% to 10% at the end of their first, second and third years of study respectively. Approximately ten years of programming experience is required in recovering to the same level of meaningful identifier names where the programmers were when they first started to learn to program. Fortunately, after this period the percentage of meaningful identifier names did improve further.

A related trend was also seen by the survey of dated software produced by a single individual over a 25 years period, with the greatest decrease in the percentage of identifier-naming style flaws occurring during the first ten years of commencing to program. These results can be applied to the general population of programmers, as the distribution of identifier-naming style flaws correlated with very strong statistical significance to the source code collected from multiple sources collected during the contemporary software survey.

6.1.4 Computer Programming Language Effects on Identifier-Naming Style

No effect on the percentage of identifier-naming style flaws was detected that could be contributed to a specific computer programming language. Very strong statistical significance in the population sample of identifier-naming style flaws was found for the correlation of the Ada and Java source code surveyed. However, there is potential to make meaningless any comparison between the two computer programming language source code samples due to the differences during when source code was developed and the differences in computer program sizes used for the comparisons. Hence, comparisons between similar size applications and computer programs completed at similar times, was undertaken. Correlations against the identifier-naming style flaws showed very strong statistical significance over both the computer program's completion date and the application size, for a range of computer programs up to and larger than 10k SLOC, and also for computer programs developed over a range of years going back 25 years. Hence, the previously identified result for comparison between Ada and Java source code remain relevant. The identifier-naming style guideline component of the research may be relevant to most if not all computer programming languages, as Ada and Java use different methods of declaring identifiers which are also used by other common computer programming languages.

6.1.5 Effort Required to Chose Meaningful Identifier Names

Unexpectedly, dynamic reporting of identifier-naming style flaws resulted in a statistically significant lesser editing time for the experimental test subject groups than for the control test subject groups. Recall that the primary instruction to the test subjects was that they should devise meaningful identifier names. Hence it can be argued that the dynamic reporting of identifier-naming style flaws allowed the test subjects to reach consensus on an identifier name faster than would occur without the benefit of the dynamic reporting of identifier-naming style flaws. Coupled with the time savings due to an expert programmer no longer being required to identify and suggest remedial action to correct an identifier-naming style flaw during a code review, the use of dynamic reporting of flaws has the potential to save a considerable amount of time from the code review activity and from the development activity.

6.2 Limitations in Material

The research literature necessarily suffers from limitations, otherwise the existing literature would have already been complete and there would be no need to further the research. Similarly, the research is also limited, as it was not possible to predict all possible outcomes and to plan against the full set of possible eventualities nor was it possible to address every avenue of opportunity. The limitations in material affecting and arising out of the research are identified and discussed in the following subsections.

6.2.1 Identifier-Naming Style Guideline Limitations

The literature reviews found nineteen identifier-naming style guidelines that were unambiguous, could be readily automated using only the source code as input and were supported by researcher's assertions or have empirical evidence supporting that the guideline directs towards improved source code readability. There may be additional identifier-naming style guidelines waiting to be discovered in the literature but they are not extensively referenced by other researchers and hence it is unlikely that they have contributed significantly to software engineering practice to date. The researchers who assert that a specific identifier-naming style guideline will direct towards improved source code readability have offered minimal justification and their justifications appear to be often based on the researcher's opinions or experience, rather than solid empirical evidence. Of the nineteen identifier-naming style guidelines only four guidelines were supported by empirical data. However, this empirical

research, relating an identifier-naming style guideline to improved source code readability, is similarly infrequently referenced within the literature and hence it is likely that this empirical research has not been duplicated nor independently verified.

Given that identifiers are used extensively throughout source code and that identifiers have the potential to offer a strong clue as to the function of the source code, it is surprising how few empirical studies addressing the construction of identifier names for readability have been published. Our understanding of the relationship between identifier-naming style practices and source code readability is currently at a primitive level.

6.2.2 Identifier-Naming Style Guideline Implementation Limitations

The implementation of the identifier-naming style guidelines suffered from an incomplete rendering of the guideline in a number of instances. See section 4.2 for a description of the implementation limitations of the identifier-naming style guidelines. This action was taken as a full implementation of the guidelines would have been prohibitively expensive in resources. This decision to incompletely implement the identifier-naming style guidelines was taken as the research was interested to discover whether the dynamic reporting of flaws would result in a consequent improvement in source code readability. The research was not overly concerned with a potential failure to mark an identifier name as being flawed.

The contemporary and dated software surveys used the same incomplete implementation of the identifier-naming style guidelines; hence a consistent result was possible across the software surveys. As the maintenance and production experiment was designed not to be affected by the incomplete implementation of the identifier-naming style guidelines and the actuality of the identifier names chosen by the test subjects did not result in anomalous reporting of flaws, the incomplete implementation of the identifier-naming style guidelines was considered acceptable.

6.2.3 Identifier-Naming Style Guideline Programmer Acceptance Questionnaire Limitations

The Identifier-Naming Style Guideline Programmer Acceptance questionnaire was limited by the relatively small number of questionnaire respondents. The number of questionnaire respondents was 34 in total. The small number of responses from academics made comparison with this group against other groups of little value. Although statistically significant results could be obtained for the expert programmer group, the small sample size of this group

increases the possibility that the sample may not be representative of the expert programmer population in general.

The Identifier-Naming Style Guideline Programmer Acceptance questionnaire was further limited as identified below:

- The questionnaire is limited in that it cannot guarantee to measure the questionnaire respondent's agreement or non-agreement against one instance of an identifier declaration that would be supported by the identifier-naming style guideline of interest. This is because the questionnaire respondent could potentially be responding to some other aspect of the identifier name that is external to the guideline of interest. Consequently, the research used the assumption that all manifestations of an identifier-naming style flaw, that modified the base identifier name when compared to the original base identifier name, would result in the questionnaire respondent choosing the base identifier name over that of the derived identifier name with the inherent identifier-naming style flaw. In order to increase the likelihood that the questionnaire respondent was actually responding to the relevant identifier-naming style guideline of interest, multiple related questionnaire statements would need to be generated per guideline. This was not undertaken as the interest in the programmer's attitude was considered secondary to the actual practice during source code editing shown by the programmers.
- The questionnaire is potentially flawed in that a different choice of identifier name pair could conceivably reverse the selection made by an arbitrary questionnaire respondent. Whether this would actually occur is unknown as only one pair of identifier names was presented to the questionnaire respondent for each of the identifier-naming style guidelines. Again, as the actual practice of the programmer was considered more important to the research than the attitudes of the programmer, the questionnaire statements were not expanded to consider multiple identifier declarations designed to test a single identifier-naming style guideline.
- The choice to collapse the qualification of class and also type identifiers into the Class/Type Qualification (09) identifier-naming style guideline on retrospect was potentially invalid. It is reasonable to expect that the same programmer, when writing Ada source code would accept the concatenation of "_Type" to the end of a type identifier name but would also reject the concatenation of "Class" to the end of a class identifier name, when writing Java source code. Hence, the acceptance of the Class/Type Qualification (09) identifier-naming style guideline may depend on the computer programming language of choice by the questionnaire respondent.

A detailed analysis would no doubt find other issues with the representation of the identifier-naming style flaws within the questionnaire statements but an extensive treatment of this issue is not valuable to the understanding of the results obtained and no further observations on this subject are offered.

6.2.4 Textbook Survey Limitations

The textbook survey is limited in that only one textbook per programming language was considered and the number of textbooks surveyed is small compared to the number published over the prior three decades. In parallel with the use of their textbooks, the student programmer would typically see example source code and hence identifier declarations from other sources (e.g. their lecturers and tutors). Hence, a survey of the example source code presented to the programmer during their tuition should also accompany the survey of the textbooks. However, this information was not available, being potentially lost in antiquity. Hence, the textbook survey cannot necessarily be considered as representative of the source code encountered by an arbitrary programmer over the period that the textbooks were published. However, as before, the actual practice of the programmer during source code editing is considered to be more important to the research than an external stimulus that may have affected the programmer.

6.2.5 Dated Software Survey Limitations

As noted previously (see section 5.4), the value of the dated software survey was potentially substantially devalued by the use of a single programmer's source code (i.e., the author) and that considerations, other than the programmer's experience, such as the programming culture at the time could have affected the source code surveyed.

Attempts to collect source code from other programmers were unrealised. The requirement for numerous computer programs for which the development periods were scattered over decades and where the computer programs had been developed by a single individual, proved to be difficult to satisfy. The research would have benefited from the additional data but has not been overly damaged as a consequence of a lack of collaborating data. The results from the maintenance and production experiment correlated generally with the results of the dated software survey in that the data from both show that as programming experience is gained, past that of the first three years, the percentage of identifier-naming style flaws decreases.

6.2.6 Contemporary Software Survey Limitations

The total amount of source code collected for analysis was less than 1M SLOC, which is significant as the motivation for the research was directed towards assisting the development and maintenance of large mission critical software systems. Consequently, no single software project analysed, as part of this research, approached 1M SLOC in size. Similarly, there is no guarantee that the software projects analysed could be classified as mission critical and indeed some were not mission critical. In addition, the comparison of data between the Ada software projects and the Java software projects may not necessarily be valid due to the temporal separation in the development period for which the Ada software projects and the Java software projects were produced. The issue being that a different chronological period of development could have affected the relevant programmer's attention to identifier-naming style practices. The choice of computer programming language has been previously addressed (see section 6.1.4). However, the results of the research are at least relevant to medium sized software systems.

6.2.7 Source Code Editor Limitations

The source code editor used to deliver the dynamic reporting of identifier-naming style flaws was lacking in functionality and performance compared to that supported by current editors. Hence potentially artificially constraining the normal editing style of the test subjects, this in turn may have affected the test subject's derivation and subsequent editing of identifier names. As the maintenance and production experiment design was purposefully insensitive to these differences and the case studies were unable to demonstrate these differences, the research is unable to respond to attempts to quantify this affect, if in fact it exists.

The source code editor also supplied capability not usually present in editors (i.e., the dynamic reporting of identifier-naming style flaws and the suggestions for replacement identifier names); hence some aspect of this new capability could have also affected the test subject's derivation of identifier names. This was the effect that the research was interested in identifying. However, as the GUI, supported by the source code editor, is novel to this research, comparison of the results against similar research was not possible.

6.2.8 Programmer Characteristics Questionnaire Limitations

The Programmer Characteristics questionnaire suffered from allowing questionnaire respondents to only select from one of five values for the number of years programming experience, with the last two selections corresponding to a range of year values. The data collected may have been more useful, if the actual number of years programming was disclosed by the questionnaire respondent. In addition, the poor response by the test subjects to the Programmer Characteristics questionnaire limited the possible analysis that could be conducted on the maintenance and production experiment data.

6.2.9 Maintenance and Production Experiment Limitations

The results from the maintenance and production experiment suffered from the unintentional allocation of undergraduate students to the experimental group and graduate students to the control group. This separation resulted in the generation of two population samples that statistically were more likely to come from different populations than from the same population, hence removing the possibility to meaningfully compare the results between the two sample populations in total. Hence, data comparisons were made using the number of years programming experience which in itself is not a good separator of programming capability (Pennington, 1987).

The maintenance and production experiment also suffered from a number of design limitations, in particular the complexity of the source code offered under the maintenance exercise may have been more complex than necessary. In particular, one of the maintenance activities required the test subject to delete the identifier declaration of an unused identifier. The choice of identifier name, i.e., “i” coupled with the presence of a duplicate declaration of the same identifier name within a “For” statement, resulted in the first declaration being effectively hidden from the attention of the test subject due to the small size of the identifier name and it is possible that the maintenance activity suffered from a lower than expected participation rate.

6.2.10 Novice Programmer Case Study Limitations

The short period of time that the novice programmer was developing software under the case study limited the value of the Novice Programmer case study. As is apparent from the Programming Team case study, approximately six software units were required to be generated

by an individual programmer before the results collected tended to settle into a trend. As the project was abruptly paused, for commercial reasons, the novice programmer only had sufficient time to generate four software units, hence potentially not allowing sufficient data to be collected to indicate the longer term affects of identifier-naming style flaw reporting on a novice programmer.

6.2.11 Programming Team Case Study Limitations

The CATS project, being a commercial endeavour, necessitated the development of a SDD before coding could commence. As the SDD typically mandates the names of classes, methods and object attributes to define meaningful identifier names, the names used in the SDD are highly dependent on the personal naming practices apparent to the author of the SDD (i.e., the author of this thesis). Hence, effort was expended to use a descriptive naming of the programming objects with the assurance given that the individual programmers need only edit the SDD in order to support their own preferred naming practices. However, there was potential for the descriptive naming supplied in the initial issue of the SDD to affect the choice of identifier names which were actually used by the programmers engaged as part of the two case studies. However, this potential may not be large, as the propensity of programmers to use their own identifier-naming practices against the weight of extensive suggestion to use a specific name was demonstrated during the introduction exercise of the maintenance and production experiment.

One concern that cannot be dismissed is the small size of the software source code generated during the case studies. 15k SLOC may be insufficient compared to a 1M SLOC software system and the programming process used during a larger project can differ in reality from those used for the development of a single team project. Hence the data collected from the Programming Team case study may not necessarily be indicative of data collected for a multi-team software development project.

6.3 Research Contribution

As a consequence of conducting the research, there are aspects of the background theory that have been supported and other aspects have not been supported. The following subsections identify what support has been offered to the background theory and what modifications are

relevant to be made to the background theory. However, before this discussion is presented, the relevance of the research is placed into perspective.

6.3.1 Research Relevance

Where the interactions of identifiers with other symbols have been studied, specifically in the realm of source code readability, the study has been limited to a single computer language within a singular working environment and using a singular software development environment. Often the study offers no insight into whether the results can be applied generally. The research considered software written in three different computer programming languages, written in different working environments and using different software development methodologies. Over this varied environment, the research found a correlation between the percentages of identifier-naming style flaws within the sample populations, which was supported with a very strong statistical significance. This very strong statistical significance is also present for software written by the same person and developed over two decades corresponding to their transition from intermediate to expert programmer. This result is more general than results where programmer experience has been singularly-valued (eg. number of years programming) and hence is of greater potential value to software engineering theory.

6.3.2 Software Maintenance Legacy

As demonstrated by the research, the percentage of meaningful identifier name declarations increases naturally during the transition of a programmer into an expert programmer. However, the progression along this path to expert programmer status initially results in a reduction in the percentage of meaningful identifier names past that produced when the programmer was at the neophyte stage in their exposure to programming. Programmers who work in sizable teams will normally progress to a position of team leader after approximately five to seven years in industry. The consequence of this promotion timetable causes software to be produced, by programmers who are at the intermediate programmer stage of their development, which results in a smaller proportion of meaningful identifier names entering the software than if the software was developed by an expert programmer. Left unchecked, this practice increases the maintenance issues, due to poor identifier naming, which will be inherited by future maintenance programmers. As previously discovered, poor identifier-naming can severally influence the ability of the programmer 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.

6.3.3 Confounding Variables

The research demonstrated that the number of years programming had a large affect on the generation of meaningful identifier names. This effect was mirrored to a lesser degree by the size of the largest computer program that the programmer had been involved in developing. What was surprising was the magnitude of the effect attributed to the Identifier-Naming Effort and the Flexible Work Practices confounding variables. Similarly, the affect on software quality potentially offered by a programmer who is a touch typist over that of a programmer who must use hunt-and-peck on the keyboard, appears to have been overlooked by research addressing software engineering practice.

Future research into identifier-naming would benefit by requesting potential test subjects to self report the relative effort that they would normally place into identifier-naming and the relative flexibility of their work practices. From a practical sense, it would appear that an effective way to improve the software quality of identifier-naming would be to require that programmers invest the effort to become touch typists.

6.3.4 Software Development Support

Poor identifier-naming practices can be reported during code inspections. However, this practice is expensive in terms of the expert programmer's time. In a Java computer program, approximately 0.34 identifier declarations would be expected to be declared per SLOC, based on the survey of the Java source code conducted during the research. For a novice programmer, where it would not be unusual to see in excess of 60% of the identifier declarations defining non-meaningful names, 100 SLOC corresponds to an investment of approximately 41 minutes [3] of an expert programmer's time to review and respond with comment. This is a relatively inconsequential figure. However, considering hundreds of programmers working on the development of a large mission critical software system of say 1M SLOC, the drain on the expert programmer's time to review the source code for poor identifier names becomes significant at four person years[4]. The typical software project manager would not make an issue

[3] This figure is based on an estimate of two minutes which corresponds to the minimum amount of time expected for an expert software engineer to recognise an identifier name as being flawed, to then document why the name is flawed and to further document a suggested meaningful identifier name to replace the flawed identifier name.

[4] The figure is based on a contact time of forty hours per week and forty weeks per year. This figure assumes absence from the office for conferences, holidays, sick leave, training and work related travel.

of the loss of 41 minutes but they do typically react to a loss of four person years even when it is spread over hundreds of individuals over a ten year period. Hence, the software project manager will often consider appropriate software tools where a time saving, during production, can be demonstrated.

As the novice programmer may not necessarily recognise an identifier name as being flawed, knowing which identifier-naming style flaws persist up to the code review will have the potential to improve source code readability as the identifier-naming style flaw can be automatically detected and reported to the programmer, thereby potentially saving review time for the expert programmer. By reporting poor identifier naming practices to the maintenance and production experiment 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 general 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. In support of this function, 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 to a central project data repository. In this instance, the source code editor has the following advantages over that of an expert programmer with regards to reporting poor identifier-naming practise: (a) it is a cheaper resource to employ, (b) it is always available, (c) it is consistent in its reporting and (d) it does not tire. In addition, dynamic reporting of identifier-naming style flaws allows the programmer to be alerted to an issue while the current programming specific information required to address the issue has not suffered from cognitive loss of focus.

6.3.5 Software Engineering Teaching

The research has implications to the teaching of software engineering in that a teacher can employ a tool, which reports identifier-naming style flaws, to assist in the automated marking of student assignments for software quality. This affords the marker additional time to concentrate on less pedestrian aspect of the student's source code. The student is also benefited in that they have immediate feedback regarding identifier-naming style best practices which they would normally only receive in industry during a code review. As industry mandates high quality software for mission critical systems, the feedback would also be useful to the programmer transitioning from developing small software systems to the development of large software systems.

It is possible that software engineering teaching would become more effective by the education of novice and intermediate programmers to emulate the identifier-naming style common to the source code generated by expert programmers. The largest reduction in identifier-naming style flaws, during the transition from intermediate to expert programmer, corresponded to the Short Name (05), Word Count (07) and English Words (12) identifier-naming style flaws. These same identifier-naming style flaws also appear as the most pervasive in production software, hence making the corresponding guidelines candidates for greater consideration during the initial programmer training.

In addition to the identifier-naming style flaws that should be reported to programmers, particularly novice and intermediate programmers, software engineering courses can now address the poor identifier-naming practices apparent in the course textbooks. The software industry has progressed past that of a cottage industry and the processes that work in the cottage industry should now be tempered by process important to large software system development.

6.4 Research Hypothesis Discussion

The research hypothesis is now discussed in terms of the fundamental proposition used as a basis for the research; the research results considered against the decomposition of the research hypothesis; and the research conclusion relevant to the research hypothesis.

6.4.1 Basic Research Proposition Discussion

The proposition inherent to the research was that programmers will tend towards the generation of source code with fewer identifier-naming style flaws over time. If this trend was not realised then the use of expert programmers to adjudicate whether an identifier name was meaningful or not would itself be a meaningless endeavour. This trend was seen for an individual programmer with data collected over a twenty-five year period which showed a reduction of identifier-naming style flaws over time. In addition, the test subjects engaged in the maintenance and production experiment demonstrated a net decrease of identifier-naming style flaws of 13% over at least a ten year period corresponding to the range of experience reported by the test subjects and a net increase of 20% in the percentage of meaningful identifier names over the same time period.

6.4.2 Research Hypothesis Parts Discussion

The research hypothesis has been separated into seven parts to aid in the following discussion and is represented as:

1. Providing dynamic reporting …
2. … of identifier-naming style flaws …
3. … during source code editing …
4. … can reduce the occurrence of these flaws, …
5. … where the identifier-naming style guideline, used to detect the flaw, …
6. … has been identified by an expert …
7. … as affecting source code readability.

Part 1: Dynamic reporting is addressed simply as an implementation method, which is employed to highlight an identifier-naming style flaw to the user. This capability was implemented by the source code editor.

Part 2: An identifier-naming style flaw is defined as an occurrence where an identifier name fails to satisfy the corresponding identifier-naming style guideline. Hence, identifier-naming style flaws are discussed in terms of the corresponding guideline. A literature search was employed to find suitable identifier-naming style guidelines that could be implemented unambiguously using source code as the only input. The purpose of this aspect of the literature search was to identify candidate identifier-naming style guidelines. It was not intended to find an exhaustive or even a consistent set of guidelines. Fifteen out of the nineteen identifier-naming style guidelines found corresponded to researchers' opinion or experience without solid empirical evidence. The remaining guidelines that did have empirical evidence only supported one aspect of source code readability. Hence programmers and in particular expert programmers were canvassed to confirm their acceptance of these guidelines. This activity reduced the list of identifier-naming style guidelines to eighteen as being considered relevant to source code readability. A survey of production source code reduced this list further to fourteen as four of the identifier-naming style guidelines were found to rarely occur in production software. This same result was also seen in the software generated by a single individual over a twenty-five year period. The remaining list of identifier-naming style guidelines could potentially be reduced by a further guideline (i.e., the Long Name (06) guideline) as in some cases expert programmers tend to ignore this guideline. However, this result is indeterminate as the research was not designed to test for the absolute truth of a guideline.

Part 3: Source code editing is a necessary and assumed activity undertaken as part of programming and as such does not require any further discussion.

Part 4: The reduction of identifier-naming style flaws was measured as a reduction in the percentage of flaws relative to the opportunity to raise the flaw. This definition does not relate to the absolute flaw density due to the different sizes of software modules and the different number of identifier declarations found in disparate software modules. The maintenance and production experiment offered dynamic reporting of identifier-naming style flaws to the test subjects. The test subjects, allocated to the control groups, devised identifier names that were characterised as containing at least one identifier-naming style flaw in 57% of the identifiers declared during the experiment, compared to 44% for the experimental groups. Similarly, the test subjects, allocated to the control groups, devised identifier names that were characterised as being meaningful for their use by an expert programmer in 29% of the identifiers declared during the experiment, compared to 44% for the experimental groups. Hence, the experimental groups devised identifier names that were identified less frequently as flawed and more frequently as being meaningful. However, this result may not necessarily be comparable as the population distributions of the test subjects who were allocated to the control groups and to the experimental groups were different. The Programming Team case study did show a greater than 20% reduction in identifier-naming style flaws over time for one programmer and more modest reductions of approximately 5% and 10% for two other programmers engaged during the Programming Team case study.

Part 5: The identifier-naming style guidelines, used by the research, were used as a mechanism only to test the identifier name that is the subject of an identifier declaration, for the occurrence of identifier-naming style flaws.

Part 6: An expert is identified by assumed experience gained over a minimum of a ten year period working in the development of large mission critical software systems. The definition is based on Simon's (1981, p108) work which identifies a minimum number of knowledge chunks that the individual should have assimilated and is qualified further by the software development environment as the development of small scale software does not necessarily guarantee that the programmer will experience the issues pertinent specifically to the development of large mission critical software systems.

Part 7: Source code readability measurement proved problematic, in that the various metrics advertised to be sensitive to source code readability ultimately proved to be insensitive as a measure of source code readability to both the source code generated during the maintenance

and production experiment and also to the source code generated by the case studies. Hence, the subjective option of employing multiple expert programmers to adjudicate over the meaningfulness of the identifier names, devised by the test subjects who were the subjects of the maintenance and production experiment and the case studies, was found to be necessary. This adjudication over the meaningfulness of the devised identifier names was highly correlated, hence supporting the authority of the expert programmers to consistently adjudicate over the meaningfulness of the identifier names. The meaningfulness of the identifier names was taken as synonymous with source code readability, as the opportunity to modify any aspect of the maintenance and production experiment source code, other than identifier names, was not taken by the test subject engaged by the maintenance and production experiment and the case studies were governed by an automatic pretty printer, which consistently formatted the source code on behalf of the test subjects.

6.4.3 Research Conclusions

The research investigation found thirteen identifier-naming style guidelines, out of an initial nineteen which were found during the literature search, that: (1) expert programmers consider to be relevant to improving source code readability; (2) that the corresponding flaws have been found to be pervasive throughout the programming textbooks surveyed and hence these flaws are potentially being positively reinforced by these textbooks; and (3) occur with sufficient frequently in production software to be relevant to being reported to a programmer for the purposes of improving the readability of the programmer's source code. These thirteen identifier-naming style guidelines were applied to the author's source code that had been generated over a twenty-four year period commencing in the author's fourth year of programming. The expected reduction of identifier-naming style flaws was seen over the period for which the source code was collected. This same effect was also present in the data collected from the maintenance and production experiment, where the data came from test subjects who had in excess of three years programming experience. This supported the expectation that increased programming experience would result in a reduction of these identifier-naming style flaws within the relevant source code.

Four expert programmers independently selected identifier names that they considered were meaningful from the list of identifier names which were devised by the test subjects who had participated in the maintenance and production experiment. The four lists of meaningful identifier names differed and so were used independently to count the number of meaningful identifier names present in the source code produced by each of the test subjects engaged in the

maintenance and production experiment. Statistical analysis showed that the count of meaningful identifier names correlated across the test subjects with a significance level of 0.0001. Hence the expert programmers demonstrated a very strong statistically significant predilection for the identification of identifier names that are superior in their meaningfulness.

The control test subjects, from the maintenance and production experiment, produced source code without assistance, with 29% of the identifier declarations being considered to be meaningful by an expert programmer. With assistance, in the form of dynamic reporting of identifier-naming style flaws, the experimental test subjects produced source code with 44% of the identifier declarations being considered to be meaningful by the same expert programmer. These values were calculated to be significant at the 5% level. Hence, the dynamic reporting of identifier-naming style flaws resulted in an increase in source code readability as demonstrated by a 15% increase of meaningful identifier declarations over that generated by the controls.

The case study also demonstrated that the dynamic reporting of identifier-naming style flaws on tested source code could result in the reduction of these flaws. As the professional programmers who participated in the case studies were requested to improve the readability of their source code in response to the reporting of identifier-naming style flaws, the net reduction of flaws measured demonstrates: (1) a willingness of the professional programmers to at least consider the reporting of these flaws; and (2) the utility of the method to highlight a potential readability issue to the professional programmer. This is important because without a means of alerting the programmer to an issue with the readability of their source code and without motivation to affect change, the method is impotent.

The research has not proved the foreground theory, which is encapsulated by the hypothesis H1, to be true. What has been achieved is the collection of empirical evidence that supports the truth of the hypothesis with high statistical significance and the realisation of the method's utility to affect action by the programmer.

6.5 Further Work

This research considered nineteen arbitrary identifier-naming style guidelines, which were expected to guide towards improved source code readability and which were implemented so as to provide dynamic reporting of a corresponding identifier-naming style flaw, as the programmer entered source code into a source code editor. The research can now be progressed further by an investigation of the underlying reasons why poor identifier-naming practise are

adopted by the programmer and an investigation of identifier-naming standardisation. In addition, there is also an opportunity to consider whether a causal relationship is apparent between poor software quality, measured by the percentage of meaningful identifier names and software quality, measured by the absence of software bugs.

6.5.1 Identifier-Naming Practices

The research uses the working assumption that programmers are both capable and willing to invest effort to devise meaningful identifier names. The research found instances where programmers were incapable without assistance and unwilling without external incentive to devise meaningful identifier names, in preference to using identifiers mealy as place holders to store run-time data. Fundamental to the further interpretation of the research would be a theory describing programmer characteristics that support an inherent ability and willingness to devise meaningful identifier names. Such research would be useful for the potential early identification of programmers who would be best suited to the development of maintainable source code.

Understanding why one programmer will invest effort into devising a meaningful identifier name and understanding why another programmer will not is becoming more important to software maintenance, particularly as the size of software systems grow. There may be cognitive, experiential or personality characteristics which explain why some programmers do not progress past (or are slower to progress past) the practice of using identifiers merely as data 'place holders', without giving consideration to using a meaningful identifier name. Understanding why one programmer will not generally invest effort to devise meaningful identifier names may lead to the ability to suggest a means with which to assist the programmer in the derivation of meaningful identifier names. Alternatively, identification of these individuals that are ultimately unsuited to developing readable source code for large mission critical software systems would also benefit industry.

6.5.2 Identifier-Naming Standardisation

The maintenance and production experiment test subjects showed considerable difficulty in devising suitable identifier names (i.e., identifier names that were defined in agreement with all of the identifier-naming style guidelines). The same test subjects also infrequently chose identifier names that coincided with the same choices made by their peers. The magnitude of this result was 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 natural language words that would be used 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.

One avenue of investigation would be to survey production software, find identifier names that were used for the same purposes and identify the most frequently defined and identify the most universally understood name forms. For instance, production source code has been found that contains identifier names using the following forms: Value_Get, Value_Put, Value_Set, Value_Update etc., the possibility to replace a multitude of forms for an identifier name with a single standard form has potential to simplify the software maintenance task. A second avenue of investigation would be to define a project word dictionary, including a definition of the word's usage and require that identifier names be composed only of these words. In addition, a list of word rules for the construction of identifier names would also be required such that it would be unlikely that a different name could be assigned to the same identifier or that the same identifier could be known by different names. Hence, an identifier name may be able to be automatically generated from a text description of the identifier's meaning, the use of the word dictionary and the word rules.

Another area of investigation would be to investigate the constrained vocabulary – i.e. arbitrarily limit the choice of words that could be used to form a particular identifier name. This arbitrary list would be identified by the survey described above. The execution of a software development case study would then be required in order to identify whether this arbitrary list had any effects on the software development project. A further software maintenance case study would be required to identify any effects on a future software maintenance project that required maintenance of the software that was the subject of the software development case study.

6.5.3 Software Engineering Practice

The research found, with strong statistical significance from the analysis of the production exercise source code of the maintenance and production experiment, that software bug free source code had approximately twice as many meaningful identifier names as source code that contained major bugs. The research has not proven an inverse relationship between software bugs and meaningful identifier names, only that when low relative percentages of meaningful identifier names in source code are detected the source code warrants additional scrutiny to be placed on the source code.

Boehm and Basili (2001) found that 80% of software defects can be attributed to 20% of the software units forming large software systems. As it is apparent that source code that is more difficult to read is also more difficult to debug, the relative readability of a software unit may be useful to potentially identify a reduced set of software units for closer inspection by the project. The ability to readily identify the software units which contain software defects, by some automated means, has implication to software engineering practice. Such a capability would allow software project resources to be concentrated on these modules to good effect.

6.6 Chapter Summary

The gift of truth excels all other gifts
Buddha

In summary, the conclusions are characterised by the following:

- The research has found that the most readable identifier names are composed of two to four words, with words separated by an underscore character, with the first letter of each word capitalised. In addition, the identifier name should not contain abbreviations.
- The research has found that the percentage of meaningful identifier names generated by student programmers decreases steadily during their instruction by as much as 20% and approximately a decade of experience is required to recover to their initial level. The percentage of meaningful identifier name declarations continues to increase past that of the neophyte as further experience is gained.
- The use of dynamic reporting of identifier-naming style flaws has the potential to increase the percentage of meaningful identifier names, on average, by 15% over that of a peer programmer that has not had the benefit of such reporting. This effect is most pronounced when the programmer is a novice. However, not all programmers will use

the information that an identifier name is flawed to devise a more readable name for the identifier.

Bibliography

Note: When a text is referenced, the reference within the body of the thesis will also typically indicate a page number. This page number corresponds to the page that the first word of the containing paragraph was printed on and not necessarily the page number that contains the most relevant component of the text being reference.

Aggarwal, 2002 — Aggarwal, Krisham K.; Singh, Yogesh; Chhabra, Jitender Kumar, *An Integrated Measure of Software Maintainability*, Proceedings of the Annual Reliability and Maintainability Symposium, 2002, pp: 235-241, 2002

Alsio, 2000 — Alsio, Gunilla; Goldstein, Mikael, *Productivity Prediction by Extrapolation: Using Workload Memory as a Predictor of Target Performance*, Behaviour & Information Technology, Vol: 19, No: 2, pp: 87-96, March 2000

Baker, 1997 — Baker, Richard A., Jr., *Code Reviews Enhance Software Quality*, ACM: Proceedings of the 19th International Conference on Software Engineering, pp: 570-571, May 1997

Barnes, 1989 — Barnes, J. G. P., Programming in Ada, third edition, Addison-Wesley, IASB: 0-201-17566-5, 1989

Barnes, 2001 — Barnes, John, *Half a Century of Programming and Not Much Progress*, Wiley: Software Focus, Vol: 2, No: 1, pp: 15-20, Spring 2001

Batory, 2003 — Batory, Don; Liu, Jia; Sarvela, Jacob Neal, *Refinements and Multi-Dimensional Separation of Concernsp*, ACM: Proceedings of the 9th European Software Engineering Conference (held jointly with the 11th International Symposium on Foundations of Software Engineering), Vol: 28, No: 5, pp: 48-57, September 2003

Bennett, 2000 — 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

Boehm, 1981 — Boehm, Barry, *Software Engineering Economics*, Prentice Hall PTR, Upper Saddle River, New Jersey, ISBN: 0-13-822122-7, 1981

Boehm, 2001 — Boehm, Barry; Basili, Victor, *Software Defect Reduction, Top 10 List*, IEEE: IEEE Computer, Vol: 34, No: 1, pp: 135-137, January 2001

Booch, 1994 — Booch, Grady, *Object-oriented Analysis and Design with Applications*, ISBN: 0-8053-5340-2, The Benjamin/Cummings Publishing Company, Inc., 1994

Boundy, 1991 — Boundy, David, *A Taxonomy of Programmers*, ACM: Software Engineering Notes, Vol: 16, No: 4, pp: 23-30, October 1991

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

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

Caprile, 1999 — Caprile, Bruno; Tonella, Paolo, *Nomen Est Omen: Analyzing the Language of Function Identifiers*, IEEE: Proceedings of the Sixth Working Conference on Reverse Engineering, pp: 112-122, October 1999

Card, 1986 — Card, S., K.; Moran, T., P.; Newell, A., *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ., 1986

Carr, 1986 — Carr, Wilfred; Kemmis, Stephen, *Becoming Critical: Education, Knowledge and Action Research, Revised Edition*, ISBN: 0-7300-0437-6, Deakin University Press, 1986

Chan, 2002 — Chan, Jien-Tsai; Yang, Wuu, *Advanced Obfuscation Techniques for Java Bytecode*, Elsevier: The Journal of Systems and Software, Vol: , No: , pp: 1-10, August 2002

Chillarege, 1996 — Chillarege, Ram, What is Software Failure?, IEEE: Transactions on Reliability, Vol: 45, No: 3, pp: 354-355, September 1996

Cohen, 1996 — Cohen, Ronald Joy; Swerdlik, Mark E; Phillips, Suzanne M., *Psychological Testing and Assessment, An Introduction to Tests and Measurement, 3rd Edition*, ISBN: 1-55934-427-X, Mayfield Publishing Company, 1996

Collins, 1969 — Collins, A., M.; Quillian, M., R., Retrieval Time from Semantic Memory, Journal of Verbal Learning and Verbal Behaviour, Vol: 8, pp: 240-247, 1969

Cordes, 1991 — Cordes, David; Brown, Marcus, *The Literate-Programming Paradigm*, IEEE: IEEE Computer, Vol: 24, No: 6, pp: 52-61, June 1991

Cox, 1993 — Cox, Kevin; Walker, David, *User-Interface Design, Second Edition*, ISBN: 0-13-952888-1, Prentice Hall, 1993

Crotty, 1998 — Crotty, Michael, *The Foundations of Social Research: Meaning and Perspective in the Research Process*, ISBN: 1-86448-604-X, Allen & Unwin, 1998

Crutchfield, 1994 — Crutchfield, Richard; Workman, David A, *Quality Guidelines = Designer Metrics*, ACM: Proceedings of the Conference on TRI-Ada, pp: 29-40, November 1994

Curtis, 1979 — Curtis, Bill; Sheppard Sylvia B; Milliman Phil; Borst M. A.; Love Tom, *Measuring the Psychological Complexity of Software Maintenance Tasks with the McCobe Metrics*, IEEE: Transactions on Software Engineering, Vol: SE-5, No: 2, pp: 96-104, March 1979

Curtis, 1984 — Curtis, Bill, *Fifteen Years of Psychology in Software Engineering: Individual Differences and Cognitive Science*, ACM: Proceedings of the 7th International Conference on Software Engineering, pp: 97-106, 1984

Das, 2000 — Das, Manuvir, *Unification-based Pointer Analysis with Directional Assignments*, ACM: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, Vol: 35, No: 5, pp: 35-46, May 2000

de Vaus, 2002 — de Vaus, D. A., *Surveys in Social Research, Fifth Edition*, ISBN: 1-86508-611-8, Allen & Unwin, 2002

Deibenbock, 2005 — Deibenbock, Florian; Pizka, Markus, *Concise and Consistent Namings*, IEEE: Proceedings of the 13th International Workshop on Program Comprehension, pp: 97-106, May 2005

Deimel, 1990 — Deimel, Lionel, E.; Naveda, J. Fernando, *Reading Computer Programs: Instructor's Guide and Exercises*, Software Engineering Institute, Carnegie Mellon University, CMU/SEI-90-EM-3, August 1990

Detienne, 2001 — Detienne, Francoise, *Software Design – Cognitive Aspects*, ISBN: 1852332530, Springer, 2001

DeYoung, 1979 — DeYoung, Gerrit E.; Kampen, Garry R, *Program Factors as Predictors of Program Readability*, IEEE: Proceedings of the IEEE Computer Software and Applications Conference, COMPSAC 79, pp: 668-673, 1979

DeYoung, 1982 — DeYoung, Gerrit E.; Kampen, Garry R.; Topolski, James M., *Analyzer-Generated and Human-Judged Predictors of Computer Program Readability*, ACM: Proceedings of the SIGCHI Conference on Human Factors in Computer Systems, pp: 223-228, 1982

Dix, 1998 — Dix, Alan; Finlay, Janet; Abowd, Gregory; Beale, Russell, *Human-Computer Interaction*, Second Edition, ISBN: 0-13-239864-8, Prentice Hall, 1998

DoD, 2000 — Defense Science Board Task Force on Defense Software, November 2000 http://www.acq.osd.mil/dsb/reports/defensesoftware.pdf#search=%22defense%20software%20size%20increases%22, Accessed: 31Aug06

Dromey, 1995 — Dromey, R. Geoff , *A Model for Software Product Quality*, IEEE: Transactions on Software Engineering, Vol: 21, No: 2, pp: 146-162, February 1995

Dromey, 2003 — Dromey, R. Geoff , *Software Quality – Prevention versus Cure?*, Springer: Software Quality Journal, Vol: 11, No: 3, pp: 197-210, 2003

Drucker, 1961 — Drucker, Peter F., *The Technological Revolution: Notes on the Relationship of Technology, Science, and Culture*, The International Quarterly of the Society for the History of Technology, Vol: 2, pp: 342-351, Winter 1961

Dumas, 1995 — Dumas, Joseph; Parsons, Paige, *Discovering the way Programmers Think About New Programming Environments*, ACM: Communications of the ACM, Vol: 38, No: 6, pp: 45-56, June 1995

Dwyer, 2004	Dwyer, Matthew B.; Clarke, Lovi A.; Cobleigh, Jamieson M.; Naumovich, Gleb, *Flow Analysis for Verifying Properties of Concurrent Software Systems*, ACM: Transactions on Software Engineering and Methodology, Vol: 13, No: 4, pp: 359-430, October 2004
Elshoff, 1982	Elshoff, James L.; Marcotty, Michael, *Improving Computer Program Readability to Aid Modification*, ACM: Communications of the ACM, Vol: 25, No: 8, pp: 512-521, August 1982
Fang, 2001	Fang, Xuefen, *Using a Coding Standard to Improve Program Quality*, Quality Software, 2001. Proceedings, Second Asia-Pacific Conference on, pp: 73-78, 2001
Fisher, 2001	Fisher, Marcus S.; Cukic, Bojan, *Automating Techniques for Inspecting High Assurance Systems*, IEEE: Proceedings of the Sixth IEEE International Symposium on High Assurance Systems Engineering, pp: 117-126, 2001
Fitzsimmons, 1978	Fitzsimmons, Ann; Love, Tom, *A Review and Evaluation of Software Science*, ACM: Computing Surveys, Vol: 10, No: 1, pp: 3-18, March 1978
Fix, 1993	Fix, Vikki; Wiedenbeck, Susan; Scholtz, Jean, *Mental Representations of Programs by Novices and Experts*, ACM: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp: 74-79, May 1993
Furnas, 1987	Furnas, G. W.; Landauer, T. K.; Gomez, L. M.; Dumais, S. T., *The Vocabulary Problem in Human-System Communication*, ACM: Communications of the ACM, Vol: 30, No: 11, pp: 964-971, November 1987
Geis, 1998	Geis, Jennifer M, *JavaWizard: Investigating Defect Detection and Analysis*, Master of Science in Information and Computer Sciences Thesis, May 1998

Ghan, 1971	Ghan, Lavern, *Better Techniques for Developing Large Scale Fortran Programs*, ACM: Proceedings of the 1971 26th Annual Conference, pp: 520-537, January 1971
Gibbs, 1994	Gibbs, W. Wyatt, *Software's Chronic Crisis*, Scientific American, pp: 86-95, September 1994
Gilb, 1993	Gilb, Tom; Graham, Dorothy, *Software Inspection*, Addison-Wesley, ISBN: 0-201-63181-4, 1993
Glass, 2002	Glass, Robert L., *Facts and Fallacies of Software Engineering*, ISBN: 0-321-11742-5, Addison-Wesley, 2002
Gleitman, 1981	Gleitman, Henry, Psychology, Norton & Company, ISBN: 0-393-95102-2, 1981
Gorla, 1990	Gorla, Narasimhaiah; Benander, Alan C.; Benander, Barbara A., *Debugging Effort Estimation Using Software Metrics*, IEEE: Transactions on Software Engineering, Vol: 16, No: 2, pp: 223-231, Feburary 1990
Grogono, 1979	Grogono, Peter, *On Layout, identifiers and Semicolons in Pascal Programs*, ACM: SIGPLAN Notices, Vol: 14, No: 4, pp: 35-40, April 1979
Halstead, 1977	Halstead, M. H., *Elements of Software Science*, Elsevier: North-Holland, New York, ISBN: 0444002057, 1977
Haneef, 1998	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
Hawkins, 1993	Hawkins, Brian, *Preventative Programming Techniques: Avoid and Correct Common Mistakes*, Charles River Media Programming Series, 2003

Hartzman, 1993 — Hartzman, Carl S.; Austin, Charles F., *Observations Based on an Experience in a Large Scale Environment*, ACM: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering Volume 1, IBM Press, pp: 138-170, October 1993

Heintze, 2001 — Heintze, Nevin; Tardieu, Olivier, *Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second*, ACM: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, Vol: 36, No: 5, May 2001

Helander, 1988 — Helander, Martin Editor, Handbook of Human-Computer Interaction, Elsevier Science Publishers, State University of New York at Buffalo, ISBN: 0-444-70536-8, 1988

Hiburn, 2000 — Hiburn, Thomas B., Towhidnejad, Massood, *Software Quality: A Curriculum Postscript?*, ACM: Proceedings of the thirty-first SIGCSE Technical Symposium on Computer Science Education, Vol: 32, No: 1, pp:167-171, March 2000

Horton, 1998 — Horton, Ivor, *Beginning C++, The Complete Language*, Wrox Press, ISBN: 1-861000-12-X, 1998

Humphrey, 1989 — Humphrey, Watts, *Managing the Software Process*, Addison-Wesley, New York, 1989

Humphrey, 1995 — Humphrey, Watts S., *Why Should You Use a Personal Software Process?*, ACM: Software Engineering Notes, Vol: 20, No: 3, pp: 33-36, July 1995

Humphrey, 2000 — Humphrey, Watts S., *The Personal Software Process: Status and Trends*, IEEE: IEEE Software, Vol: 17, No: 6, pp: 71-75, November/December 2000

Humphrey, 2002 — Humphrey, Watts S., *Winning with Software: An Executive Strategy*, Addison-Wesley, ISBN: 0-201-77639-1, 2002

IEEE Std 610.12-1990 — *IEEE Standard Glossary of Software Engineering Terminology*, 1990

ISO-9126:1991 — *Software Product Evaluation – Quality Characteristics and Guidelines for their use*, 1991

Jensen, 1978 — Jensen, Kathleen; Wirth, Niklaus, *Pascal User Manual and Report, second edition*, Springer-Verlag, ISBN: 0-387-90144-2, 1978

Johnson, 1977 — Johnson, Stephen C., Lint, a C Program Checker, Computer Science Technical Report 65, Bell Laboratories, December, 1977 http://plan9.bell-labs.com/7thEdMan/vol2/lint, Accessed: 01Jul05

Johnston, 1997 — Johnston, Simon, *Ada 95 for C and C++ Programmers*, Addison-Wesley, ISBN: 0-201-40363-3, 1997

Kearney, 1986 — Kearney, Joseph K.; Sedlmeyer, Robert L.; Thompson, William B.; Gray, Michael A.; Adler, Michael A., *Software Complexity Measurement*, ACM: Communications of the ACM, Vol: 29, No: 11, pp: 1044-1050, November 1986

Keller, 1990 — Keller, Daniel, *A Guide to Natural Naming*, ACM: SIGPLAN Notices, Vol: 25, No: 5, pp: 95-102, May 1990

Kernighan, 1988 — Kernighan, Brian W.; Ritchie, Dennis M., *The C Programming Language, second edition*, Prentice Hall Software Series, ISBN: 0-13-110362-8, 1988

Khwaja, 1993 — Khwaja, Amir Ali; Urban, Joseph E., *Syntax-Directed Editing Environments: Issues and Features*, ACM: Proceedings of the ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice, pp: 230-237, March 1993

Kline, 1995 — Kline, Stephen Jay, *Conceptual Foundations for Multi-Disciplinary Thinking*, ISBN: 0-8047-2409-1, Stanford University Press, 1995

Koenemann, 1991 — Koenemann, Jurgen; Robertson, Scott P., *Expert Problem Solving Strategies for Program Comprehension*, ACM: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Reaching through Technology, pp: 125-130, 1991

Laitinen, 1992 — Laitinen, Kari; Mukari, Timo, *DNN – Disciplined Natural Naming, A Method for Systematic Name Creation in Software Development*, Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 1992. Volume: ii., pp: 91-100, 1992

Laitinen, 1996 — Laitinen, Kari, *Estimating Understandability of Software Documents*, ACM: SIGSOFT Software Engineering Notes, Vol: 21, No: 4, pp: 81-92, 1996

Land, 1997 — Land, Lesley Pek Wee; Sauer, Chris; Jeffery, Ross, *Validating the Defect Detection Performance Advantage of Group Designs for Software Reviews: Report of a Laboratory Experiment Using Program Code*, ACM: Communications of the ACM, pp: 294-309, November, 1997

Langlois, 2003 — Langlois, Richard N., *Cognitive Comparative Advantage and the Organization of Work: Leassons from Herbert Simon's Vision of the Future*, Journal of Economic Psychology, Vol: 24, No: 2, pp: 167-187, April, 2003

Ledgard, 1979 — Ledgard, Henry, F.; Hueras, John, F.; Nagin, Paul, A., *Pascal with Style: Programming Proverbs*, Hayden Computer Programming series, ISBN: 0-8104-5124-7, 1979

Li, 2005 — Li, Xiaosong; Prasad, Christine, *Effectively Teaching Coding Standards in Programming*, ACM: Proceedings of the 6th Conference on Information Technology Education, pp: 239-244, October 2005

Lieberman, 2001 — Lieberman, Henry; Fry, Christopher, *Bridging the Gulf Between Code and Behavior in Programming*, http://belladonna.media,mit.edu/people/lieber/Lieberary/ZStep/Bridging/Bridging.html, Accesses: 18Oct01

Lipaev, 2005 — Lipaev, V. V., *Problems of the Development and Quality Control of Large Software Systems*, Springer: Programming and Computer Software, Vol: 31, No: 1, pp: 47-49, 2005

Matis, 1996 — Matis, David W., *The Graphic Design of Text: A Review of Research*, Proceedings of Technical Communications, pp: 230-235, 1996

Mauri, 1982 — Mauri, Ross A.; Williams, A. Harry, *Extending Halstead's Software Science for a More Precise Measure of APL*, ACM: ACM SIGAPL Quote Quad: Proceedings of the International Conference on APL, Vol: 13, No: 1, pp: 207-213, July 1982

McCabe, 1976 — McCabe, T. H., *A Complexity Measure*, IEEE: Transactions on Software Engineering, Vol: 2, No: 6, pp: 308-320, December 1976

McCall, 2004 — McCall, Gavin (Ed), *Guidelines for the use of the C Language in Critical Systems*, The Motor Industry Software Reliability Association, MISRA-C:2004, October 2004

McConnell, 1993 — McConnell, Steve, *Code Complete: A Practical Handbook of Software Construction*, ISBN: 1-55615-484-4, Microsoft Press, 1993

McConnell, 2001 — McConnell , Steve, *Who Needs Software Engineering*, IEEE: IEEE Software, Vol: 18, No: 1, pp: 5-8, January/February 2001

McCracken, 1972 — McCracken, Daniel D., *A Guide to Fortran IV Programming, second edition*, John Wiley & Sons, ISBN: 9-471-58281-6, 1972

McIver, 1996 — McIver, Linda; Conway, Damian, *Seven Deadly Sins of Introductory Programming Language Design*, Proceedings of the International Conference Software Engineering: Education and Practice, 1996, pp: 309-316, 1996

Mengel, 1999 — 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

Meyer, 1988 — Meyer, Bertrand, *Object-oriented Software Construction*, ISBN: 0-13-629031-0, Prenntice Hall, 1988

MIL-STD-1815 — *Ada Programming Language*, Military Standard, USA Department of Defense, 10th December 1980

Mohan, 2004 — Mohan, Andrew; Gold, Nicolas; Layzell, Paul, *An Initial Approach to Assessing Program Comprehensibility using Spatial Complexity, Number of Concepts and Typographical Style*, IEEE: Proceedings of the 11th Working Conference on Reverse Engineering, pp: N/A, November 2004

Nakashima, 1999 — 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

Neumann, 2002 — 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

Oliver, 1995 — Oliver, David W., *Systems Engineering & Software Engineering, Contracts and Synergism*, IEEE: Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering, pp: 186-193, July 1995

Oman, 1990a	Oman, Paul W.; Cook, Curtis R, *A Taxonomy for Programming Style*, ACM: Proceedings of the ACM Annual Conference on Cooperation, pp: 244-250, January 1990
Oman, 1990b	Oman, Paul W.; Cook, Curtis R, *Typographic Style is More than Cosmetic*, ACM: Communications of the ACM, Vol: 33, No: 5, pp: 506-520, May 1990
Oppenheim, 1992	Oppenheim, A. N., *Questionnaire Design, Interviewing and Attitude Measurement*, Continuum: New Edition, ISBN: 0-8264-51764
Oualline, 1992	Oualline, Steve, *C Elements of Style: The Programmer's Style Manaual for Elegant C and C++ Programs*, Hungry Minds, Inc. ISBN: 1558512918, November 1992
Pearse, 1995	Pearse, Troy; Oman, Paul, *Maintainability Measurements on Industrial Source Code Maintenance Activities*, IEEE: Proceedings of the International Conference on Software Maintenance, 1995, pp: 295-303, October 1995
Pennington, 1971	Pennington, Nancy, *Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs*, Elsevier: Cognitive Psychology, Vol: 19, pp:295-341, 1987
Perry, 1994	Perry, Chad, *A Structured Approach to Presenting PhD Theses: Notes for Candidates and their Supervisors*, ANZ Doctoral Consortium, University of Sydney, February 1994
Phillips, 2000	Phillips, Estelle, M.; Pugh, Derek, S., *How to get a PhD: A handbook for students and their supervisors, Third Edition*, ISBN: 0-335-20550-X, Open University Press, 2000

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

Porter, 1995 — Porter, A.; Siy, H.; Toman, C. A.; Votta, L. G., *An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development*, ACM: Proceedings of the 3rd SIFSOFT Symposium on Foundations of ACM SIGSOFT Software Engineering Notes, pp: 92–103, October 1995

Raghunathan, 2005 — Raghunathan, Srinivasan; Prasad, Ashutosh; Mishra, Birendra, K.; Chang, Hsihui, *Open Source versus Closed Source: Software Quality in Manopoly and Competitive Markets*, IEEE: Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans; Vol: 35, No: 6, pp: 903-918, November 2005

Rajlich, 2001 — Rajlich, Vaclav; Wilde, Norman; Buckellew, Michelle, *An Experimental Investigation of the Impact of Individual, Program, and Organizational Characteristics on Software Maintenance Effort*, Elsevier: The Journal of Systems and Software; Vol: 54, No: 1, pp: 137-157, October 2000

Ramanujan, 2000 — Ramanujan, Sam; Scamel, Richard W.; Shah, Jaymeen R., *Software Cultures and Evolution*, IEEE: IEEE Computer, Vol: 34, No: 9, pp: 24-28, September 2001

Rees, 1982 — Rees, Michael J., *Automatic Assessment Aids for Pascal Programs*, ACM: ACM SIGPLAN Notices, Vol: 17, No: 10, pp: 33-42, October 1982

Reps, 1989 — Reps, Thomas W.; Teitelbaum, Tim, *The Synthesizer Generator A System for Constructing Language-Based Editors*, ISBN: 0-387-96857-1, Springer-Verlag, 1989

Rising, 1989 — 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

Roberts, 2000 — Roberts, Simon; Heller, Philip; Ernest, Michael, *Complete Java 2 Certification, Study Guide, second edition*, Richard Mills Publisher, ISBN: 0-7821-2825-4, 2000

Rosen, 1995 — Rosen, J-P, *A Naming Convention for Classes in Ada 9X,* ACM: SIGAda Letters, Vol: XV, No: 2, pp: 54-56, March 1995

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

Rountev, 2000 — Rountev, Atanas; Chandra, Satish, *Off-line Variable Substitution for Scaling Points-to Analysis*, ACM: Proceedings of the ACM SIGPLAN 2000, Conference on Programming Language Design and Implementation PLOI'00, Vol: 35, No: 5, pp: 47, May 2000

Ryder, 2001 — Ryder, Barbara G.; Landi, William A.; Stocks, Philip A.; Zhang, Sean; Altucher, Rita, *A Schema for Interprocedural Modification Side-Effect Analysis*, Vol: 23, No: 2, pp: 105-186, March 2001

Sackman, 1970 — Sackman, Harold, *Man-Computer Problem Solving: Experimental Evaluation of Time-Sharing and Batch Processing*, Auerbach, New York, 1970

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

Schneidewind, 1996	Schneidewind, Norman F.; Fenton, Norman, *Point Counter Point: Do Standards Improve Quality?*, IEEE: IEEE Software, Vol: 13, No: 1, pp: 22-24, January 1996
Schorsch, 1990	Schorsch, Tomas Michael, *Increasing the Readability and Comprehensibility of Programs*, Report No: AFIT/CI/CIA-90-141, NASA Center for AeroSpace Information, 1990
Schroeder, 1977	Schroeder, D. Michael; Clark, David D.; Saltzer, Jerome H., *The Multics Kernel Design Project*, ACM: Proceedings of the Sixth ACM Symposium on Operating Systems Principles, pp: N/A, 1977
Shanklin, 1991	Shanklin, Robert E., Jr., *Developing Standards With and For the User*, IEEE: Proceedings of the Fourth IEEE Software Engineering Standards Application Workshop, pp: 14-17, 1991
Shepperd, 1988	Shepperd, Martin, *A Critique of Cyclomatic Complexity as a Software Metric*, IEEE: Software Engineering Journal, Vol: 3, No: 2, pp: 30-36, March 1988
Shneiderman, 1980	Shneiderman, Ben, *Software Comprehension*, Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishers, ISBN: 0-87626-816-5, 1980
Simon, 1955	Simon, Herbert A., *A Behavioral Model of Rational Choice*, The Quarterly Journal of Economics, Vol: LXIX, pp: 99-118, 1955
Simon, 1981	Simon, Herbert A., *The Sciences of the Artificial*, 2nd Edition, MIT Press, 1981
Siy, 2001	Siy, Harvey; Votta, Lawrence, *Does the Modern Code Inspection Have Value?*, Proceedings of the IEEE: IEEE International Conference on Software Maintenance, pp: 281-289, 2001

Slaughter, 1996	Slaughter, Sandra, *Assessing the Use and Effectiveness of Metrics in Information Systems*, ACM: Proceedings of the 1996 ACM SIGCPR/SIGMIS Conference on Computer Personnel Research, pp: 384-391, April 1996
Sneed, 1996	Sneed, Harry M., *Object-Oriented Cobol Recycling*, IEEE: Proceedings of the Third Working Conference on Reverse Engineering, pp: 169-178, November 1996
Sommerville, 1995	Sommerville, Ian, *Software Engineering, Fifth Edition*, ISBN: 0-201-42765-6, Addison-Wesley, 1995
SPC, 1995	*Ada 95 Quality and Style: Guidelines for Professional Programmers*, Software Productivity Consortium, Herndon, Virginia USA, SPC-94093-CMC, Version: 01.00.10, October 1995 http://www.informatik.uni-stuttgart.de/ifi/ps/ada-doc/style_guide/cover.html, Accessed: 31Aug06
Spinellis, 2003	Spinellis, Diomidis, *Reading, Writing, and Code*, ACM: Queue, Vol: 1, No: 7, pp: 84-89, October 2003
Teasley, 1994	Teasley, Barbee E., *The Effects of Naming Style and Expertise on Program Comprehension*, Elsevier: Human-Computer Studies, Vol: 40, No: 5, pp: 757-770, May 1994
Teleman, 1996	Teleman, Mark Anthony, *The Design of the User Interface for Software Development Tools*, Department of Computer Science, The University of Queensland, PhD Thesis, May 1996
Tenny, 1988	Tenny, Ted, *Program Readability: Procedures Versus Comments*, Software Engineering, IEEE: IEEE Transactions on, Vol: 14, No: 9, pp: 1271-1279, September 1988

Tilley, 1993	Tilley, Scott R., *Documenting-in-the-large vs. Documenting-in-the-small*, ACM: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Distributed Computing – Volume 2, IBM Press, pp: 1083-1090, October 1993
Truong, 2004	Truong, Nghi; Roe, Paul; Bancroft, Peter, *Static Analysis of Students' Java Programs*, ACM: Proceedings of the Sixth Australian Computing Education Conference, Vol: 30, pp: 317-325, January 2004
Vanter, 2002	Vanter, Michael Van De; Gosling, James, *Jackpot Project Targets Tools*, http://java.sun.com/features/2002/12/jackpot.html, Accessed: 18Dec02
Vaughan-Nichols, 2003	Vaughan-Nichols, Steven J., *Building Better Software with Better Tools*, IEEE: Computer, Vol: 36, No: 9, pp: 12-14, September 2003
Vermeulen, 2000	Vermeulen, Allan; Ambler, Scott, W.; Bumgardner, Greg; Metz, Eldon; Misfeldt, Trevor; Shur, Jim; Thompson, Patrick, *The Elements of Java Style*, SIGS Reference Library, Cambridge University Press, ISBN: 0521 777682, 2000
Virzi, 1992	Virzi, Robert, A., *Refining the Test Phase of Usability Evaluation: How Many Subjects is Enough?,* Human Factors & Ergonomics Society, Vol: 34, No: 4, pp: 457-468, August 1992
Vivanco, 2003	Vivanco, R. A., *Identifying Effective Software Metrics Using Genetic Algorithms*, IEEE: Canadian Conference on Electrical and Computer Engineering, Vol: 2, pp: 1305-1308, May 2003
Voas, 2004	Voas, Jeffrey, *Software's Secret Sauce: The "-ilities"*, IEEE: Software, Vol: 21, No: 6, pp: 14-15, November-December 2004
Vollman, 1993	Vollman, Thomas E., *Software Quality Assessment and Standards*, IEEE: Computer, Vol: 26, No: 6, pp: 118-120, June 1993

von Mayrhauser, 1994	von Mayrhauser, A.; Vans, A. M., *Comprehension Processes During Large Scale Maintenancen*, IEEE: Proceedings of the 16th International Conference on Software Engineering, pp: 39-48, May 1994
Weinberg, 1998	Weinberg Gerald M., *The Psychology of Computer Programming, Silver Anniversary Edition*, ISBN: 0-932633-42-0, Dorset House Publishing, New York, 1998
Weissman, 1974	Weissman, Larry, *Psychological Complexity of Computer Programs: An Experimental Methodology*, ACM: ACM SIGPLAN Notices, Vol: 9, No: 6, pp: 25-36, June 1974
Wells, 1986	Wells, Mark B., *A Potpourri of Notational Pet Peeves (and their Resolution in Modcap)*, ACM: ACM SIGPLAN Notices, Vol: 21, No: 3, pp: 21-30, March, 1986
Wells, 1995	Wells, Charles H.; Brand, Russel; Markosian, Lawrence, *Customized Tools for Software Quality Assurance and Reengineering*, IEEE: IEEE Proceedings of the 2nd Working Conference on Reverse Engineering, 1995., pp: 71-77, July 1995
Welsh, 1986	Welsh J.; Rose, G. A.; Lloyd, M., *An Adaptive Program Editor*, The Australian Computer Journal, Vol: 18, No: 2, pp: 67-74, May 1986
Wheeler, 1998	Wheeler, Sharon; Duggins, Sheryl, *Improving Software Quality*, ACM: Proceedings of the 36th Annual Southeast Regional Conference, pp: 300-309, April 1998
Wickens, 1992	Wickens, Christopher, D., *Virtual Reality and Education*, IEEE: International Conference on Systems, Man and Cybernetics, Vol: 1, pp: 842-847, October, 1992

Wilson, 1998	Wilson, David N.; Hall, Tracy, *Perceptions of Software Quality: A Pilot Study*, Springer: Software Quality Journal, Vol: 7, No: 1, pp: 67-75, 1998
Wirth, 1982	Wirth, Niklaus, *Programming in Module-2*, Springer-Verlag, ISBN: 3-540-11674-5, 1982
Wu, 1999	Wu, C. Thomas, *An Introduction to Object-oriented Programming with Java*, McGraw-Hill, ISBN: 0-07-116850-8, 1999
Yourdon, 1975	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, 1975
Zeller, 2000	Zeller, Andreas, *Making Students Read and Review Code*, ACM: ACM SIGCSE Bulletin, Proceedings of the 5th Annual GIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education, Vol: 32, No: 3, pp: 89-92, July 2000

Appendix A - Identifier-Naming Style Guidelines

This appendix contains a list of identifier-naming style guidelines that have been identified, by this research, and have been described in Chapter 4. For each of the identifier-naming style guidelines, the following additional information is presented:

1. A conversational description of the identifier-naming style guideline is offered. The text of which is presented to the test subject, by the source code editor, when the test subject is considering whether to correct the corresponding flaw in their source code.
2. A statement of the identifier-naming style guideline is presented as a heuristic. This heuristic was implemented by the source code editor.
3. The basis for the identifier-naming style guideline is given.
4. The declaration of an identifier presents the opportunity to define an identifier name that contains an identifier-naming style flaw. However, an identifier declaration cannot guarantee to raise a specific identifier-naming style flaw in all cases, only the opportunity to raise the flaw. For instance, to ascertain whether one identifier name is similar to another identifier name, there is a requirement that there be at least two identifiers declared within the same programming scope. A discussion is presented, which describes how the identification of an opportunity to derive an identifier-naming style flaw was achieved. The count of opportunities to raise specific identifier-naming style flaws is used by the research to derive the percentage of the specific identifier-naming style flaws generated by an arbitrary software unit.
5. Example source code written in both the Ada programming language syntax and the Java programming language syntax is presented. The example source code shows an instance of a specific identifier-naming style flaw that has been purposefully injected into the source code and a corresponding example which corrects the flaw that was initially inserted. An example of the identifier-naming style flaw and a correction is presented to the test subjects, when they consider whether to correct an identifier-naming style flaw, as a guide to help them understand the nature of the flaw.

A.1 Un-named Constant

All numeric values, except for -1, 0 and 1, should have associated with them an identifier name describing the meaning of the numeric value. The numeric value by itself is often undescriptive of its meaning. An advantage of naming all constants is that should the value require to be modified, it need only be changed in one place and not potentially in multiple places through out the entire program which may not necessarily all be successfully found.

Guideline:	Numeric literals should only be used in a constant declaration. However, the literals -1, 0 and 1 may be regarded as exceptions to this guideline.
Basis:	Assertion by researcher
Effort:	Low cognitive effort
Opportunity:	Count each numeric value other than -1, 0 or 1 used except those used within a constant declaration.

	Example	Suggested Correction

Ada:	`Readability_Index :=`
	`  Logic_Branch_Count * 0.013;`
		`Logic_Branch_Weighting :`
		`  constant Integer := 0.013;`
		`...`
		`Readability_Index :=`
		`  Logic_Branch_Count *`
		`  Logic_Branch_Weighting;`

Java:	`readabilityIndex =`
	`  logicBranchCount * 0.013;`
		`final static int`
		`  LOGIC_BRANCH_WEIGHTING =`
		`  0.013;`
		`...`
		`readabilityIndex =`
		`  logicBranchCount *`
		`  LOGIC_BRANCH_WEIGHTING;`

A.2 Multiple Underscore

Multiple contiguous underscore characters should not be used in an identifier name. The use of different identifiers composed of the same words and in the same order but separated by differing numbers of contiguous underscore characters will result in difficulty for the programmer to distinguish between these different identifiers.

Guideline:	Identifier names should not contain multiple contiguous underscore characters.
Basis:	Assertion by researcher
Effort:	Low cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant) within a method.

	Example	Suggested Correction
Ada:	`Apple__Count`	`Apple_Count`
Java:	`apple__Count`	`appleCount`

A.3 Outside Underscore

Application source code should not define identifier names with a leading or a trailing underscore character, as by convention, these identifier names are usually associated with the operating system, windowing system and other low-level machine functions.

Guideline:	Identifier names should not start or end with the underscore character.
Basis:	Assertion by researcher
Effort:	Low cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant).

	Example	Suggested Correction
Ada:	`_Apple_Count`	`Apple_Count`
Java:	`_appleCount`	`appleCount`

A.4 Numeric Digit(s)

Identifier names should not include numeric characters in their composition. Identifiers corresponding to numerically qualified identifier names (e.g. *Element_1*, *Element_2*) should be declared as an array and not as separate identifiers.

Guideline:	Identifier names should not contain numeric characters.
Basis:	Assertion by researcher
Effort:	Low cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant).

	Example	Suggested Correction
Ada:	`Apple_Count_1, Apple_Count_2`	`Apple_Count` (declared as an array)
Java:	`appleCount1, appleCount2`	`appleCount` (declared as an array)

A.5 Short Name

Identifier names that are short may be lacking in relevant meaning and should be avoided. Identifiers names should contain at least eight characters. However, short variable names, which are used in mathematical algorithms, are acceptable.

Guideline:	Identifier names should be composed of no fewer than eight characters. However, the algebraic identifier names (i.e., i, j, k, l, m, n, t, x, y or z), when declared as a variable, may be regarded as exceptions to this rule.
Basis:	Empirical result: based on debugging time
Effort:	Low cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant).

	Example	Suggested Correction
Ada:	`Count`	`Apple_Count`
Java:	`Count`	`appleCount`

A.6 Long Name

Identifier names should not be overly long. The use of overly long identifier names (i.e., greater than twenty characters) can result in source code formatting which is more difficult to read than if shorter identifier names were used.

Guideline:	Identifier names should be composed of at most twenty characters.
Basis:	Empirical result: based on debugging time
Effort:	Low cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant).

	Example	Suggested Correction
Ada:	`Number_Of_Jonathan_Apple_Counted`	`Apple_Count`
Java:	`numberOfJonathanAppleCounted`	`appleCount`

A.7 Word Count

Identifier names should consist of two to four words or project sanctioned acronyms. The use of overly long identifier names can result in source code formatting which is more difficult to read than if shorter identifier names were used and identifier names comprised of only one word lack full meaning.

Guideline:	Identifier names should be composed of two, three or four words. However, the algebraic identifier names (i.e., i, j, k, l, m, n, t, x, y or z), when declared as a variable, may be regarded as exceptions to this rule.
Basis:	Assertion by researcher
Effort:	Low cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant).

	Example	Suggested Correction
Ada:	`Number_Of_Jonathan_Apple_Counted`	Apple_Count
Java:	`numberOfJonathanAppleCounted`	appleCount

A.8 Identifier Encoding

Identifier names should not use encoding (i.e., Hungarian notation). The consistent application of an encoding schema gives information about the data type of the identifier at the point of use. However, the use of the encoding schema requires an investment in time to memorise the arbitrary character prefixes, which may change from one project to the next. There is also a cognitive overhead in reading the identifier name as a consequence of the encoding which can distract the programmer from other coding considerations. As the maintenance of this schema requires a discipline of the programmer, which is not enforced by a compiler, there is a potential for the encoding information to be misleading if incorrectly maintained.

Guideline:	Identifier names should not contain Hungarian notation encoding.
Basis:	Assertion by researcher
Effort:	Low cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant).

	Example	**Suggested Correction**
Ada:	`aApple_Count, iApple_Count` or `dApple_Count`	`Apple_Count`
Java:	`aAppleCount, iAppleCount` or `dppleCount`	`appleCount`

Encoding Qualifiers

a	ch	com	ds	fra	grdp
almn	chk	crdm	dst	frm	gry
ani	chkp	ctr	dt	fsrm	h
anim	chtm	d	e	g	hed
b	clkm	dat	f	gagm	hsb
bed	clp	db	fd	gagm	hsbm
c	cmd	dicm	fdc	gav	hsbp
c	cmdm	dir	fil	gd	hstm
calm	cmdm	dirs	film	gpb	i
cbo	cmdp	dlg	fils	gphd	img
cbop	cmgm	drv	flpm	gra	ink
cbos	cmin	drvs	fon	grd	invm

itgm	lstp	np	pvrm	src	txt
ix	lsts	o	r	ss	txtm
ixc	m	odpm	radp	st	txtp
key	mci	ole	rpc	strm	u
kstm	mdcm	opt	s	svrm	v
l	mdi	out	s	swtm	vnt
lbl	mnu	p	scr	tb	vsb
lblm	mnus	pa	shp	td	vsbm
limn	mpm	pic	sldm	tmr	vsbp
lin	mps	picm	spn	tmrm	w
lp	mrqm	picp	spnm	tolm	wn
lst	n	pn13d	sprm	trem	

A.9 Class/Type Qualification

Class/Type identifier names should be declared with the word “Class” for Java and “Type” for Ada concatenated to the end (i.e., right hand side) of the identifier name. This makes type cast operations apparent; otherwise they could be confused for a function call, particularly in the case of the Ada computer language, causing the programmer to loose focus while searching for the function declaration. This practice allows for the simplification of source code as the same name, minus the qualification, can be used for a variable identifier declaration as the programmer no longer has to search for a synonym for the variable name, which is an instance of the parent class or type. Otherwise this could present an opportunity for confusion as the programmer attempts to distinguish between the class/type name and the variable name.

Guideline:	Class identifier names should be qualified by the word “class” concatenated to the end of the identifier name. Type identifier names should be qualified by the word “type” concatenated to the end of the identifier name.
Basis:	Assertion by researcher
Effort:	Low cognitive effort
Opportunity:	Count each class declaration and each array, enumeration list, record and sub-range declaration.

	Example	**Suggested Correction**
Ada:	`Apple_Count`	`Apple_Count_Type`
Java:	`AppleCount`	`AppleCountClass`

A.10 Constant/Variable Qualification

Constant and variable identifier names that are qualified by common words i.e., maximum, minimum should have the qualification appended to the end (i.e., right) of the identifier name and not the front or middle of the identifier name. Similarly, any common abbreviations should be expanded (i.e. “ch”, “idx”, “max”, “min”, “num”, “ptr”, “rec”, “str”). The important part of the identifier name is hence read first and the qualifier is read last.

Guideline:	Identifier names should have common qualifications expanded and appended to the end of the identifier name.
Basis:	Assertion by researcher
Effort:	Low cognitive effort
Opportunity:	Count each constant and variable identifier declared.

	Example	Suggested Correction
Ada:	`Maximum_Apple_Count`	`Apple_Count_Maximum`
Java:	`maximumAppleCount`	`appleCountMaximum`

Abbreviation List

ch	max	num	rec
idx	min	ptr	str

Note: This list is incomplete as it only needs to contain the abbreviations expected to be used in an identifier name during the course of the maintenance & production experiment.

A.11 Abstract Words

Identifier names should not be composed entirely of words that name abstract concepts. The exclusive use of abstract words in an identifier name offers no understanding as to the nature or the intended use of the identifier.

Guideline:	Identifier names should not be composed only of abstract words.
Basis:	Assertion by researcher
Effort:	Moderate cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant).

	Example	Suggested Correction
Ada:	`Do_It, Handle_Division4`	N/A
Java:	`doIt, handleDivision4`	N/A

Abstract Word List

add	exponential	mass	procedure
and	force	method	radius
area	frequency	minus	resistance
current	function	multiply	return
data	greater	not	routine
distance	handle	numeric	stuff
divide	heat	or	subtract
do	height	perform	time
equal	it	plus	type
every	length	pointer	voltage
everything	less	power	volume
exclusive	list	pressure	width

Note: This list is incomplete as it only needs to contain the abstract words expected to be used in an identifier name during the course of the maintenance and production experiment.

A.12 English Words

Identifier names should be composed of English words or project sanctioned acronyms. The use of proper names in an identifier name offers no understanding as to the nature or the intended use of the identifier. The use of abbreviations and acronyms that are not sanctioned by the project results in cognitive overloading to the task of reading and understanding the source code. As first the abbreviation/Acronym must be recognised and the applicable concept identified and then associated with the identifier. The use of foreign language words may elude the programmer's ability to associate any concept with the identifier, if they are unfamiliar with the foreign language word or result in additional cognitive overloading if the word is known but they do not 'think' in the foreign language. The occurrence of a spelling error can distract the programmer from the task of understanding the source code and can add to cognitive overloading while the programmer searches for the correct spelling, a task in it self unrelated to understanding the source code.

Guideline:	Identifier names should not contain a word that cannot be found in the dictionary, or is not a common acronym or a project sanctioned acronym.
Basis:	Empirical result: based on source code understandability
Effort:	Moderate cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant).

	Example	**Suggested Correction**
Ada:	`App_Cnt`	`Apple_Count`
Java:	`appCnt`	`appleCount`

A.13 Numeric Name

Identifier names should not be composed entirely of numeric values or names of numeric values. An identifier composed solely of numeric values and names of numeric values offers no understanding as to the nature or the intended use of the identifier.

Guideline:	Identifier names should not be composed only of words used to name numeric values.
Basis:	Assertion by researcher
Effort:	Moderate cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant).

	Example	Suggested Correction
Ada:	One_Hundred_And_One, Ten4	N/A
Java:	oneHundredAndOne, ten4	N/A

Numeric Name Word List

one	twenty	eight	eightieth
two	thirty	ninth	ninetieth
three	forty	tenth	hundredth
four	fifty	eleventh	thousandth
five	sixty	twelfth	millionth
six	seventy	thirteenth	plus
seven	eighty	fourteenth	minus
eight	ninety	fifteenth	point
nine	half	sixteenth	power
ten	third	seventeenth	and
eleven	quarter	eighteenth	hundred
twelve	first	nineteenth	thousand
thirteen	second	twentieth	million
fourteen	third	thirtieth	billion
sixteen	forth	fortieth	googol
seventeen	fifth	fiftieth	googolplex
eighteen	sixth	sixtieth	infinity
nineteen	seventh	seventieth	

A.14 Plural Word

Identifier names should be composed of words in the singular. Mixing the use of both the plural and the singular form of words in an identifier name can result in confusion as to what the correct identifier name is. An arbitrary decision to remove all plural words from an identifier name in preference to the singular form, results in a step towards more consistent identifier naming.

Note: The algorithm, to identify plural words, marks any word that terminates with a single character "s" but not a double character "ss" as being potentially in the plural and then uses a lookup table to identify those words that are not in actuality plurals. .

Guideline:	Identifier names should contain words only in the singular form.
Basis:	Assertion by researcher
Effort:	Moderate cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant).

	Example	Suggested Correction
Ada:	`Apple_Counts`	`Apple_Count`
Java:	`appleCounts`	`appleCount`

Plural Exceptions Word List

abs	canvas	exists	radius
alias	celsius	focus	status
always	christmas	has	surplus
anonymous	diesis	minus	synchronous
asynchronous	dos	modulus	this
axis	electrolysis	plus	was
bus	equals	previous	yes

A.15 Naming Convention

Identifier names declared with non-standard identifier naming conventions (i.e., mixed character cases and the inconsistent use of the underscore character) should be avoided. The identifier-naming should correspond to the identifier naming convention appropriate to the project standard.

Guideline:	Identifier names should not deviate from the project selected identifier-naming convention.
Basis:	Asserted by researcher
Effort:	Moderate cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant).

	Example	**Suggested Correction**
Ada:	`ApPleCount`	`Apple_Count`
Java:	`ApPle_Count`	`appleCount`

A.16 Duplicate Names

A second identifier, with the same name as a prior identifier, should not be declared within the program scope of the first identifier when they are of different types (eg. constant, class, variable etc.). The practice of declaring an identifier with the same name within the program scope of another can cause confusion as to which identifier is being accessed at any particular place in the source code, particularly in the case where the programmer has missed seeing one of the declarations. However, overloading of function and procedure names (i.e., methods) is acceptable practice.

Note: That this identifier-naming style guideline is a special case of the guideline A.17 – Similar Names.

Guideline:	Identical identifier names that are different in kind should not be declared within the same program scope.
Basis:	Empirical result: based on time to 'crack' source code with over use of the same identifier names.
Effort:	High cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant) and subtract one for the software module (i.e., at least two identifiers are required in order to duplicate an identifier name).

	Example	Suggested Correction

Ada:	`Apple_Count : Integer;`
	
	`procedure Fruit_Counter is`
	`  Apple_Count : Integer;`
	…
		`Apple_Count_Global : Integer;`
		
		`procedure Fruit_Counter is`
		`  Apple_Count : Integer;`
		…

Java:	`int appleCount;`
	
	`void fruitCounter () {`
	`  int appleCount;`
	…
		`int appleCountGlobal;`
		
		`void fruitCounter () {`
		`  int appleCount;`

A.17 Similar Names

Identifier names differing by only one or two characters should be avoided. Similar looking identifier names can be easily confused by the programmer hence reducing readability of the source code.

Guideline:	Identifier names differing by one or two characters should not be declared within the same program scope.
Basis:	Assertion by researcher
Effort:	High cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant) and subtract one for the software module (i.e., at least two identifiers are required in order to register similar identifier names). Note: That identical identifier names are not countered.

	Example	**Suggested Correction**
Ada:	`Coordinate_Point` `Coordinate_Pointer`	N/A
Java:	`coordinatePoint` `coordinatePointer`	N/A

A.18 Unused Identifier

If an identifier has been declared but not used, the declaration should be removed. Unused identifiers have the potential to mislead the maintenance programmer as to the intent of the source code and can cause confusion during program debugging.

Guideline:	Identifiers that are declared but not used should have the declaration removed.
Basis:	Assertion by researcher High cognitive effort
Effort:	Count each constant, enumeration constant, exception, type and variable identifier declared.
Opportunity:	

	Example	Suggested Correction

Ada:	`Apple_Count : Integer;`
	`Orange_Count : Integer;`
	
	`procedure Counter_Initialise`
	`is`
	`begin`
	`  Apple_Count := 0;`
	`end Fruit_Counter;`
		`Apple_Count : Integer;`
		
		`procedure Counter_Initialise is`
		`begin`
		`  Apple_Count := 0;`
		`end Fruit_Counter;`

Java:	`int appleCount;`
	`int orangeCount;`
	
	`void counterInitialise () {`
	`  appleCount := 0;`
	`}`
		`int appleCount;`
		
		`void counterInitialise () {`
		`  appleCount := 0;`
		`}`

A.19 Same Words

Identifier names composed of the same words but in a different order should be avoided as this practice can cause confusion to the programmer.

Guideline:	Identifier names that are composed of the same words but occurring in a different order should not be declared within the same program scope.
Basis:	Assertion by researcher
Effort:	High cognitive effort
Opportunity:	Count each identifier declared (including each enumeration constant) and subtract one for the software module (i.e., at least two identifiers are required in order to duplicate the word list in an identifier name). Note: That identical identifier names are not countered.

	Example	**Suggested Correction**
Ada:	Coordinate_Point Point_Coordinate	 N/A
Java:	coordinatePoint pointCoordinate	 N/A

Appendix B - Identifier-Naming Style Guideline Programmer Acceptance Questionnaire

Considering the choice of identifier name only, which of the two source code fragments (A or B) is more readable?

- A Much More Readable
- A Slightly More Readable
- Same
- B Slightly More Readable
- B Much More Readable

The question number corresponds to the identifier-naming style guideline number, identified in Appendix A – Identifier-Naming Style Heuristic. Each question has two source code methods for comparison. The source code in the left-hand side of the table is the null case and the source code in the right-hand side of the table would cause the corresponding identifier-naming style heuristic to be activated. The source code in the top part of the table is written in the Ada computer language and the source code in the bottom part of the table is written in the Java computer language. Only one computer language listing, corresponding to the computer language chosen by the respondent, was actually displayed to the respondent.

<table>
<tr><td rowspan="2">Q1</td><td>

```
procedure Account_Balance_Sheet_Value is

  Count_Maximum : constant Integer
:= 10;
  Value_Count   : Integer;

begin

  Value_Count := Count_Maximum;

end Account_Balance_Sheet_Value;
```

</td><td>

```
procedure Account_Balance_Sheet_Value
is


  Value_Count  : Integer;

begin

  Value_Count := 10;

end Account_Balance_Sheet_Value;
```

</td></tr>
<tr><td>

```
void accountBalanceSheetValue ()
{

  final int COUNT_MAXIMUM = 10;
  int ValueCount;

  valueCount = COUNT_MAXIMUM;

}
```

</td><td>

```
void accountBalanceSheetValue ()
{


  int ValueCount;

  valueCount = 10;

}
```

</td></tr>
</table>

Q2		

	procedure Account_Balance_Sheet_Value is
	  Value_Count : Integer;
	begin
	  Value_Count := 1;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		  Value__Count : Integer;
		begin
		  Value__Count := 1;
		end Account_Balance_Sheet_Value;

	void accountBalanceSheetValue ()
	{
	  int **valueCount**;
	  valueCount = 1;
	}
		void accountBalanceSheetValue ()
		{
		  int **value__Count**;
		  value__Count = 1;
		}

Q3		

	procedure Account_Balance_Sheet_Value is
	  Value_Count : Integer;
	begin
	  Value_Count := 1;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		  _Value_Count : Integer;
		begin
		  _Value_Count := 1;
		end Account_Balance_Sheet_Value;

	void accountBalanceSheetValue ()
	{
	  int **valueCount**;
	  valueCount = 1;
	}
		void accountBalanceSheetValue ()
		{
		  int **_valueCount**;
		  _valueCount = 1;
		}

Q4

```
procedure Account_Balance_Sheet_Value is

  Value_Count  : array ( 1 .. 2 ) of
Integer;


begin

  Value_Count(1) := 1;
  Value_Count(2) := 1;

end Account_Balance_Sheet_Value;
```

```
procedure Account_Balance_Sheet_Value
is

  Value_Count_1  : Integer;
  Value_Count_2  : Integer;

begin

  Value_Count_1 := 1;
  Value_Count_2 := 1;

end Account_Balance_Sheet_Value;
```

```
void accountBalanceSheetValue ()
{

  int [1] valueCount;


  valueCount[0] = 1;
  valueCount[1] = 1;

} // accountBalanceSheetValue
```

```
void accountBalanceSheetValue ()
{

  int valueCount1;
  int valueCount2;

  valueCount1 = 1;
  valueCount2 = 1;

} // accountBalanceSheetValue
```

Q5

```
procedure Account_Balance_Sheet_Value is

  Value_Count  : Integer;

begin

  Value_Count := 1;

end Account_Balance_Sheet_Value;
```

```
procedure Account_Balance_Sheet_Value
is

  Count        : Integer;

begin

  Count := 1;

end Account_Balance_Sheet_Value;
```

```
void accountBalanceSheetValue ()
{

  int valueCount;

  valueCount = 1;

}
```

```
void accountBalanceSheetValue ()
{

  int count;

  count = 1;

}
```

Q6		

	procedure Account_Balance_Sheet_Value is
	  Value_Count : Integer;
	begin
	  Value_Count := 1;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		  Apple_Johnathon_Count : Integer;
		begin
		  Apple_Johnathon_Count := 1;
		end Account_Balance_Sheet_Value;

	void accountBalanceSheetValue ()
	{
	  int **valueCount**;
	  valueCount = 1;
	}
		void accountBalanceSheetValue ()
		{
		  int **appleJohnathonCount**;
		  appleJohnathonCount = 1;
		}

Q7		

	procedure Account_Balance_Sheet_Value is
	  Value_Count : Integer;
	begin
	  Value_Count := 1;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		  Apple_Johnathon_Variety_Value_Count : Integer;
		begin
		  Apple_Johnathon_Variety_Value_Count := 1;
		end Account_Balance_Sheet_Value;

	void accountBalanceSheetValue ()
	{
	  int **valueCount**;
	  valueCount = 1;
	}
		void accountBalanceSheetValue ()
		{
		  int **appleJohnathonVarietyValueCount**;
		  appleJohnathonVarietyValueCount = 1;
		}

Q8		

	procedure Account_Balance_Sheet_Value is
	  Value_Count : Integer;
	begin
	  Value_Count := 1;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		  iValue_Count : Integer;
		begin
		  iValue_Count := 1;
		end Account_Balance_Sheet_Value;

	void accountBalanceSheetValue ()
	{
	  int **valueCount**;
	  valueCount = 1;
	}
		void accountBalanceSheetValue ()
		{
		  int **iValueCount**;
		  iValueCount = 1;
		}

Q9		

	procedure Account_Balance_Sheet_Value is
	  type **Count_Type** is Integer;
	  Value_Count : **Count_Type**;
	begin
	  Value_Count := 1;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		  type **Count** is Integer;
		  Value_Count : **Count**;
		begin
		  Value_Count := 1;
		end Account_Balance_Sheet_Value;

	class **CountClass**
	{
	  void accountBalanceSheetValue ()
	  {
	  int valueCount;
	  valueCount = 1;
	  }
	}
		class **Count**
		{
		  void accountBalanceSheetValue ()
		  {
		  int valueCount;
		  valueCount = 1;
		  }
		}

Q10

```
procedure Account_Balance_Sheet_Value is

  Count_Minimum : constant Integer
:= -10;
  Count_Maximum : constant Integer
:= +10;

  type Count_Type is Integer
        range Count_Minimum ..
Count_Maximum;

  Count_Value : Count_Type;

begin

  Count_Value := Count_Minimum +
Count_Maximum;

end Account_Balance_Sheet_Value;
```

```
procedure Account_Balance_Sheet_Value
is

  Count_Min     : constant Integer
:= -10;
  Maximum_Count : constant Integer
:= +10;

  type T_Count is Integer
        range Count_Min ..
Maximum_Count;

  Value_Count : T_Count;

begin

  Value_Count := Count_Min +
Maximum_Count;

end Account_Balance_Sheet_Value;
```

```
void accountBalanceSheetValue ()
{

  final int COUNT_MINIMUM = -10;
  final int COUNT_MAXIMUM = +10;

  int valueCount;

  valueCount = COUNT_MINIMUM +
COUNT_MAXIMUM;

}
```

```
void accountBalanceSheetValue ()
{

  final int COUNT_MIN     = -10;
  final int MAXIMUM_COUNT = +10;

  int valueCount;

  valueCount = COUNT_MIN +
MAXIMUM_COUNT;

}
```

Q11

```
procedure Account_Balance_Sheet_Value is

  Value_Count  : Integer;

begin

  Value_Count := 1;

end Account_Balance_Sheet_Value;
```

```
procedure Account_Balance_Sheet_Value
is

  Handle_Data  : Integer;

begin

  Handle_Data := 1;

end Account_Balance_Sheet_Value;
```

```
void accountBalanceSheetValue ()
{

  int valueCount;

  valueCount = 1;

}
```

```
void accountBalanceSheetValue ()
{

  int handleData;

  handleData = 1;

}
```

Q12		

	procedure Account_Balance_Sheet_Value is
	  Value_Count : Integer;
	begin
	  Value_Count := 1;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		  Value_Cnt : Integer;
		begin
		  Value_Cnt := 1;
		end Account_Balance_Sheet_Value;

	void accountBalanceSheetValue ()
	{
	  int **valueCount**;
	  valueCount = 1;
	}
		void accountBalanceSheetValue ()
		{
		  int **valueCnt**;
		  valueCnt = 1;
		}

Q13		

	procedure Account_Balance_Sheet_Value is
	  Value_Count_Maximum : constant Integer := 100;
	  Value_Count : Integer;
	begin
	  Value_Count := **Value_Count_Maximum**;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		  One_Hundred : constant Integer := 100;
		  Value_Count : Integer;
		begin
		  Value_Count := **One_Hundred**;
		end Account_Balance_Sheet_Value;

	void accountBalanceSheetValue ()
	{
	  final int **VALUE_COUNT_MAXIMUM** = 100;
	  int valueCount;
	  valueCount = **VALUE_COUNT_MAXIMUM**;
	}
		void accountBalanceSheetValue ()
		{
		  final int **ONE_HUNDRED** = 100;
		  int valueCount;
		  valueCount = **ONE_HUNDRED**;
		}

Q14		

	procedure Account_Balance_Sheet_Value is
	  Value_Count : Integer;
	begin
	  Value_Count := 1;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		  Values_Count : Integer;
		begin
		  Values_Count := 1;
		end Account_Balance_Sheet_Value;

	void accountBalanceSheetValue ()
	{
	  int **valueCount**;
	  valueCount = 1;
	}
		void accountBalanceSheetValue ()
		{
		  int **valuesCount**;
		  valuesCount = 1;
		}

Q15		

	procedure Account_Balance_Sheet_Value is
	  Value_Count : Integer;
	begin
	  Value_Count := 1;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		  valueCount : Integer;
		begin
		  valueCount := 1;
		end Account_Balance_Sheet_Value;

	void accountBalanceSheetValue ()
	{
	  int **valueCount**;
	  valueCount = 1;
	}
		void accountBalanceSheetValue ()
		{
		  int **Value_Count**;
		  Value_Count = 1;
		}

Q16

```
Value_Count_Global : Integer;

procedure Account_Balance_Sheet_Value is

  Value_Count : Integer;

begin

  Value_Count := 1;

end Account_Balance_Sheet_Value;
```

```
Value_Count : Integer;

procedure Account_Balance_Sheet_Value
is

  Value_Count : Integer;

begin

  Value_Count := 1;

end Account_Balance_Sheet_Value;
```

```
int valueCountGlobal;

void accountBalanceSheetValue ()
{

  int valueCount;

  valueCount = 1;

}
```

```
int valueCount;

void accountBalanceSheetValue ()
{

  int valueCount;

  valueCount = 1;

}
```

Q17

```
procedure Account_Balance_Sheet_Value is

  Value_Count     : Integer;
  Regulator_Count : Integer;

begin

  Value_Count     := 1;
  Regulator_Count := 1;

end Account_Balance_Sheet_Value;
```

```
procedure Account_Balance_Sheet_Value
is

  Value_Count : Integer;
  Valve_Count : Integer;

begin

  Value_Count := 1;
  Valve_Count := 1;

end Account_Balance_Sheet_Value;
```

```
void accountBalanceSheetValue ()
{

  int valueCount;
  int regulatorCount;

  valueCount = 1;
  regulatorCount = 1;

}
```

```
void accountBalanceSheetValue ()
{

  int valueCount;
  int valveCount;

  valueCount = 1;
  valveCount = 1;

}
```

Q18		

	procedure Account_Balance_Sheet_Value is
	
	begin
	  null;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value
		is
		  Value_Count : Integer;
		begin
		  null;
		end Account_Balance_Sheet_Value;

	void accountBalanceSheetValue ()
	{
	
	  continue;
	}
		void accountBalanceSheetValue ()
		{
		  int **valueCount**;
		  continue;
		}

Q19		

	procedure Account_Balance_Sheet_Value is
	  Value_Count_Initial : Integer;
	  Value_Count : Integer;
	begin
	  Value_Count_Initial := 1;
	  Value_Count := 1;
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value
		is
		  Count_Value : Integer;
		  Value_Count : Integer;
		begin
		  Count_Value := 1;
		  Value_Count := 1;
		end Account_Balance_Sheet_Value;

	void accountBalanceSheetValue ()
	{
	  int **valueCountInitial**;
	  int **valueCount**;
	  valueCountInitial = 1;
	  valueCount = 1;
	}
		void accountBalanceSheetValue ()
		{
		  int **countValue**;
		  int **valueCount**;
		  countValue = 1;
		  valueCount = 1;
		}

Q20	procedure Account_Balance_Sheet_Value is
	
	  Value_Colour : (**Colour_Blue**,
	  Colour_Green,
	  Colour_Red);
	
	begin
	
	  Value_Colour := Colour_Red;
	
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		
		  Value_Colour : (**Colour_Red**,
		  Colour_Blue,
		  Colour_Green);
		
		begin
		
		  Value_Colour := Colour_Red;
		
		end Account_Balance_Sheet_Value;

	N/A	N/A

Q21	procedure Account_Balance_Sheet_Value is
	
	  Value_Colour : (**Colour_Blue**,
	  Colour_Green,
	  Colour_Red);
	
	begin
	
	  Value_Colour := Colour_Red;
	
	end Account_Balance_Sheet_Value;
		procedure Account_Balance_Sheet_Value is
		
		  Colour_Value : (**Blue**,
		  Green,
		  Red);
		
		begin
		
		  Colour_Value := Red;
		
		end Account_Balance_Sheet_Value;

	N/A	N/A

The following text corresponds to the help information that is displayed to the questionnaire respondent when they press the Help button.

You will be shown two complementary fragments of source code. The formatting, indenting and the actual function of the computer language statements, for the purposes of this questionnaire, are unimportant. What is important is that you make some judgment on the identifier names used by the two source code fragments. Please assume that the identifier is used to count some value where the word "value" only acts as a place holder for some more meaningful name.

Several different identifier naming formats will be presented. Some of these formats have empirical research justifying the format and some are considered best practice by one or more authors. These heuristics are identified 01 through 21 for the Ada programming language and 01 through 19 for the Java computer language, please see below.

You can respond to the questionnaire for either the Ada or the Java computer language. You can select which computer language you prefer by selecting the "Preferred Language" at the bottom-left of the questionnaire form.

The questionnaire application presents a number of source code fragments to you. You will note your progression through the files from the counter at the bottom-centre of the questionnaire form. You can enter a number or spin through the numbers to revisit any question that you have previously responded to. You are at liberty to change your response at any time.

When you exit the questionnaire application a text file will be generated that codes your responses. There are no hidden characters and no covert information is saved to this file. The text file identifies the computer language you used to respond to the questionnaire and it codes the selection you made against each source code fragment. The responses are correlated against the specific heuristic being considered (i.e., 01 through 21 or 01 through 19, as appropriate). The source code fragments have been presented in random order and that is why the heuristic numbering is not sequential.

Your responses will be collated and the identity of individuals will not be attached to the collated response. No person, other than the research student, will view the file content. All participants will be emailed a copy of the collated response. This response will define the specific identifier-naming style heuristic that were considered and the frequency of acceptance attributed to these heuristic.

Appendix C - Programmer Characteristics Questionnaire

Q1. Please enter your group name.

Q2. Have you ever worked as a professional programmer (i.e., have you ever been paid to write software)?
Yes, No

Q3. For how many years have you been writing software (including the years gained after high school)?
1, 2, 3, 4 - 9, 10 or more

Q4. What is the largest computer program (identified by Source Lines of Code – SLOC) you have ever personally contributed to?
0 – 1k, 1k – 2k, 2k – 5k, 5k – 10k, 10k+

Q5. Have you ever had your source code reviewed at a code review?
Yes, No

Q6. Have you ever reviewed someone else's source code for a code review?
Yes, No

Q7. Can you touch-type (i.e., type text accurately without looking at the keyboard)?
Yes, No

Q8. Compared to the people you study/work with, how much effort would you say you put into choosing identifier names when you write source code?
Much more than average, More than average, About average, Less than average, Much less than average

Q9. Compared to the people you study/work with, how flexible would you say you are to changing the way you work?
Much more flexible, More flexible, About average, Less flexible, Much less flexible

Appendix D - Test Subject Ethics Release & Instruction

Source Code Readability Improvement
Using Heuristic-Based Dynamic Error Reporting during Editing

You are invited to participate in the research project, on a voluntary basis, with the title of: "Source Code Readability Improvement Using Heuristic-Based Dynamic Error Reporting during Editing" i.e., an experiment using an editor that reports deviations from a coding standard, similar to the way Microsoft Word reports grammatical errors. The research is being conducted by Phillip Relf, a student of the University of Technology, Sydney (UTS), for the purpose of his Ph.D. degree.

The purpose of this experiment is to demonstrate that a source code editor would be able to encapsulate, at least some of the knowledge (in the form of a collection of heuristic, i.e., rule-of-thumb) amassed by a software engineering expert and deliver the benefit of this knowledge to the software engineer via the editor interface. By doing so, potential distractions to the programming activity, such as adherence to a coding standard, need not cause actual distraction to the software engineer while they are developing or maintaining software.

Your participation in this research will involve editing source code using a source code editor that will be provided for the purpose of the experiment. The source code editor automatically records your actions to a log file. The meaning of the text lines in the log file should be transparent. You are free to open the log file to view the content. No coded or hidden information is written to the log file. You will be invited to release these log files to Phillip Relf but are not required to do so, should you have any concerns regarding the nature of the research or concerns regarding the content of these log files. The log files will not be seen by any person, other than Phillip Relf.

You can contact Phillip Relf directly on 0403070370 or you can contact his research supervisor: Associate Professor David Lowe on (02) 9514 2526, if you have any concerns about the research. You are free to withdraw your participation from this research project at any time and without giving a reason. Your decision to participate in this research will not affect your standing as a UTS student.

You are free to ask Phillip Relf any questions regarding this research at any time before, during or after you have participated in the experiment.

The data you supply will be collectively analysed and may be reported in a research paper. However, any data published will not identify you individually and any results published will not be attributable to an identifiable individual. Data collected from individuals will not be disclosed to UTS staff.

Note: This research has been approved by the University of Technology, Sydney (UTS) Human Research Ethics Committee. If you have any complaints or reservations about any aspect of your participation in this research which you cannot resolve with the researcher, you may contact the Human Research Ethics Committee through the Research Ethics Officer, Ms. Susanna Davis on (02) 9514 1279, Susanna.Davis@UTS.edu.au.. Any complaints you make will be treated in confidence and investigated fully, and you will be informed of the outcome.

Please date and sign below, to indicate your understanding of the above and willingness to participate in the case study. Your signature is a requirement of the UTS Human Research Ethics Committee. The purpose of the form is to ensure that you understand your rights and that you are not subject to unethical practice. You are not required to return this form to the researcher.

_____ _____

Researcher: Signature/Date Participant: Signature/Date

Instruction

Your participation in this research will involve editing source code using a source code editor. When the editor is started, the editor will enable access to a questionnaire for your completion. In addition, there are three exercises. The first exercise allows you to become familiar with the user interface (approximately 10 minutes). The second exercise requests you to modify a software unit (approximately 15 minutes). The third exercise is a software development exercise (approximately 10 minutes).

Please chose one or two partners or you can choose to work alone. Whether you work with a partner or not, you will be asked to complete the experiment individually. You are free to talk to your partner(s) for suggestions and advice. However, what ever actions you take should be your own (i.e. you should not feel obliged to take the advice of others, particularly if you don't agree with it). This is not a psychology test; the experiment only expects you to work as though you are employed as a professional software engineer who is part of a team. The experiment will collect data on your choice of identifier names and requires that you use the source code editor supplied during the entire experiment.

To start the experiment, please double click on the source code editor icon (i.e., **SCE.exe**). Please note that the source code editor may on occasion be engaged in some protracted background processing. Normally, you can ignore any background processing and the status display is presented only to give you an indication of which source lines are currently being updated on the display.

Questionnaire

Please select from the main menu: **view/questionnaire** and answer the questions. You can decline to answer any or all questions. To exit the questionnaire, please press the "Save and Exit" button.

The intent of the questionnaire is to correlate your particular choices of identifier name against the individual categories that you selected within the questionnaire.

Experiment Zero

The identifier names in the file "zero.java" have been chosen to be practically meaningless. Your task is to replace the meaningless identifier names with identifier names that you personally consider will improve the readability of the source code.

Please select from the main menu: **File/open,** open the file "**zero.java**" and expand the window to full size. After the file opens you will note that identifier names that are the subject of an identifier declaration will have a **red squiggly** line underneath.

Please do the following:

1. Right-click on the numeric literal "10" (that has a red squiggly line underneath), a popup menu should appear.
2. Select the first menu item (i.e., Unnamed Constant) and left-click on this menu item, a dialog box should appear.
3. In the Replacement edit area, type in an identifier name that you consider is more readable (i.e., the value 10 by itself could just as easily mean ten apples, the first base-ten multiplier or have some other meaning). In other words, please supply a constant name that identifies the intended meaning of the value.
4. Press the "Define" button.

Depending on your choice of identifier name, the dialog box may not necessarily close. In that case you may wish to repeat steps 3 and 4 above. However, you can push the "Cancel" button at any time to close the dialog box.

Please repeat the above process for the other three identifier declarations (i.e., "fred", "A_global" and "Zero"). However, for the identifier names you will need to press the "Replace" button at step 4. When the popup menu appears, please select the first menu item in the list.

You will note that a number of comments have been inserted into the source code listing. These comments contain the history of changes that have been made under automatic control of the editor. You can remove these comments, at any time, by selecting **Edit/ SCE comment remove** from the main menu. Please save the file, by selecting **File/Save**, when you have finished editing.

The intent of experiment zero is to introduce you to the editor interface.

Experiment One

Please select from the main menu: **File/Open** and open the file "**one.java**" in the dialog box. Please save when you have finished editing the file.

This file contains source code written for the purpose of the experiment. However, you are being asked to believe that the source code forms part of some production software system that now requires maintenance to add to the existing capability. For the purpose of this experiment, please assume that you can modify any identifier name, within the file, without consequence to external software that would use this software.

Your task is to modify the source code to accomplish the following:

1. Include a new colour: **Violet**, this also requires that you modify the **Succ** method.
2. Extend the **Prime Get** method to return the fourth prime number (i.e., 7).
3. Extend the Colour Table size from **10** elements to **20** elements (i.e., **maxRec**).
4. Modify the **Record Element Order Correct** method to use the **Fibonacci Number** as the second sort key (instead of the Prime Number).
5. Fix the bug in the **Swap** method. You will note that a line has been commented out as it does not compile (i.e., there is a typographical error in the identifier name). You should fix this bug and un-comment the source code line. Hint: the word "Value" is missing in the identifier name.
6. Remove the declaration of any unused identifiers in the **Exchange Sort** method.
7. Modify the **Measure Values** method to add one to the **Function Result** if there is at least one instance where a call to the Prime Get method, with a parameter corresponding to the Colour Table Index, returns a value greater than 10. Note: you should only add one once and not every time that the Prime Get method returns a value greater than 10. You should assume that the Prime Get method could be modified at any time in the future, and this will not be under your control. Hint: declare a Boolean identifier initialised to the value False and set it to the value True inside the *For* loop as appropriate, and test the value after you exit the *For* loop to identify whether the Function Result should be incremented.

The intent of experiment one is to emulate a software maintenance task.

Experiment Two

Please select from the main menu: **File/Open** and open the file "**two.java**" in the dialog box. Please save when you have finished editing the file.

Your task is to **write an Exchange Sort method**. A source code skeleton that defines the design of the Exchange Sort method, represented by pseudo-code, is supplied for you to edit. Please change the identifier name of the array (that is to be sorted) as you feel appropriate.

The intent of experiment two is to emulate a software development task.

Appendix E - Experiment Files

The following source code listing was stored in the file zero.java.

```
class Zero
{

  int   A_global = 0;
  // used to record the Time Index value

  //-----------------------------------------------------------------------------
  //  This method increments a Time Index variable using modulo 10 arithmetic //
  //  (i.e., the value ranges from 0 .. 9).  The Time Index is used to        //
  //  control various external functions that directly read the Time Index.   //
  //-----------------------------------------------------------------------------

  public void fred
  {

    // Increment the Time Index
    A_global = A_global + 1;

    // Reset the Time Index to a modulo 10 value
    if (A_global == 10)
    {
      A_global = 0;
    }

  }

}
```

The following source code listing was stored in the file one.java.

```
//--------------------------------------------------------------------------
//                                                                        --
//  Class Name:  CT                                                       --
//                                                                        --
//  Purpose:     This package generates a table of colours associated with  --
//               Prime number and Fibonacci sequence number pairs.        --
//               The algorithm used is <total fabrication and is used only  --
//               to make the source code look like it actually does       --
//               something>.  Once the table has been generated, it is    --
//               sorted in ascending order.                               --
//                                                                        --
//--------------------------------------------------------------------------

public class CT {

  public static final int  RED       = 0;
  public static final int  GREEN     = 1;
  public static final int  Blue      = 2;
  public static final int  Yel       = 3;
  public static final int  I         = 4;
  public static final int  Forty_Two = 5;
  // Identifies the colour enumerations, the order is arbitrary and
  // may be changed without consequence.  The last element represents
  // an unknown colour.


  public int succ(int index) {
    switch (index) {
      case 0:  return GREEN;
      case 1:  return Blue;
      case 2:  return Yel;
      case 3:  return I;
      case 4:  return Forty_Two;
      case 5:  return RED;
    }

    return 0; // Needed to keep the compiler happy.

  } // succ


  private class Rec {
      public int  Colo;    // used to hold the Colour
      public int  P_;      // used to hold the Prime Number
      public int  F;       // used to hold the Fibonacci Number
  } // Rec
```

```
//---------------------------------------------------------------------------
//                                                                           --
//  Class Name:  NumericFunction                                             --
//                                                                           --
//  Purpose:     This class encapsulates the numeric functions.              --
//                                                                           --
//---------------------------------------------------------------------------
public static class NumericFunction {


  //-------------------------------------------------------------------------
  //                                                                       --
  //  Name:      fibReturn                                                 --
  //                                                                       --
  //  Purpose:   This function generates the Ith number in the Fibonacci  --
  //             sequence.                                                 --
  //                                                                       --
  //-------------------------------------------------------------------------
  public static int fibReturn (int  wF) {

    if (wF <= 2) {
      return 1;
    }
    else {
      return fibReturn(wF - 2) + fibReturn(wF - 1);
    }

  } // fibReturn


  //-------------------------------------------------------------------------
  //                                                                       --
  //  Name:      primeGet                                                  --
  //                                                                       --
  //  Purpose:   This function returns the Ith Prime number.              --
  //                                                                       --
  //-------------------------------------------------------------------------
  public static int primeGet (int wP) {

    switch (wP) {
        case  1:  return  2;
        case  2:  return  3;
        case  3:  return  5;
        default:  return  0;
    }

  } // primeGet


} // numericFunction
```

```
public class CTabClass {

  static final int maxRec = 10;
  // Defines the maximum number of records in a colour table

  Rec cTabType[] = new Rec[maxRec];
  // Defines the colour table type

} // CTabClass

CTabClass  CTab;
// Used to hold a copy of the colour table

int  functionResult;
// Used to return the results of the Measure function

boolean finished = false;
// Used to indicate that the colour table has been loaded and sorted



//-------------------------------------------------------------------------
//                                                                       --
//  Name:        recordElementOrderCorrect                               --
//                                                                       --
//  Purpose:       This function identifies whether the two record        --
//                 elements are in the correct sort order i.e., that the  --
//                 first should appear before the second.                 --
//                                                                       --
//-------------------------------------------------------------------------
  boolean recordElementOrderCorrect (Rec  Rec_1,
                                     Rec  Rec_Two) {

    boolean  function__Result;
    // Used to return the function result

    function__Result = true;
    if (Rec_1.Colo > Rec_Two.Colo) {
      function__Result = false;
    }
    else if ((Rec_1.Colo == Rec_Two.Colo)
         &&  (Rec_1.P_   >  Rec_Two.P_)) {
      function__Result = false;
    }

    return function__Result;

  } // recordElementOrderCorrect
```

```
//-----------------------------------------------------------------------
//                                                                       --
//  Name:        Swaps                                                   --
//                                                                       --
//  Purpose:     This procedure swaps the order of two record elements.  --
//                                                                       --
//-----------------------------------------------------------------------
void Swaps (CTabClass cTab,
            int       recToo,
            int       recTwo)  {

  Rec  colourRecordElementUsedForTemporaryValueHolding = new Rec();
  // Used to hold a temporary copy of a record element

  colourRecordElementUsedForTemporaryValueHolding = cTab.cTabType[recToo];
  cTab.cTabType[recToo] = cTab.cTabType[recTwo];
//   cTab.cTabType[recTwo] = colourRecordElementUsedForTemporaryHolding;

} // Swaps


//-----------------------------------------------------------------------
//                                                                       --
//  Name:        exSort                                                  --
//                                                                       --
//  Purpose:     This procedure uses the Exchange Sort algorithm to sort  --
//               the records of the colour record array.                 --
//                                                                       --
//-----------------------------------------------------------------------
void exSort () {

  int  i;
  // Used to index the colour table array

  boolean  finished = false;
  // Used to indicate that at least one set of records were swapped

  while ( !finished ) {
    finished = true;
    for (int i = 0; i < CTab.cTabType.length; i++) {
      if (i < CTab.cTabType.length) {
        if (recordElementOrderCorrect(CTab.cTabType[i], CTab.cTabType[i-1])){
          finished = false;
          Swaps(CTab, i-1, i);
        }
      }
    }
  }

} // exSort
```

```
//---------------------------------------------------------------------------
//                                                                           --
//  Name:        measureValues                                               --
//                                                                           --
//  Purpose:     This function measures an attribute of the colour table  --
//               calculated by the Fictitious algorithm.                     --
//                                                                           --
//---------------------------------------------------------------------------
public int measureValues () {

  CTabClass  Colour_Table = new CTabClass();
  // Used to hold the colour table

  functionResult = 0;
  for (int coloursTableIndex = 0;
           coloursTableIndex < CTab.cTabType.length;
           coloursTableIndex++) {
    functionResult = functionResult
                   + CTab.cTabType[coloursTableIndex].F
                   + primeGet(coloursTableIndex);
  }

  return functionResult;

} // measureValues


//---------------------------------------------------------------------------
//                                                                           --
//  Name:        DO_IT                                                       --
//                                                                           --
//  Purpose:     This procedure loads the colour table with required      --
//               values.                                                     --
//                                                                           --
//---------------------------------------------------------------------------
public void DO_IT() {

  for (int i = 0;  i < CTab.cTabType.length; i++) {
    CTab.cTabType[i].Colo = indexColour(i);
    CTab.cTabType[i].P_   = NumericFunction.primeGet(i);
    CTab.cTabType[i].F    = NumericFunction.fibReturn(i);
  }
  exSort();
  finished = true;

} // DO_IT
```

```
  //---------------------------------------------------------------------------
  //                                                                         --
  //  Name:        indexColour                                               --
  //                                                                         --
  //  Purpose:     This function returns a colour corresponding to some      --
  //               arbitrary enumeration.  At the end of the enumeration     --
  //               list the next element considered rotates back to the      --
  //               first.                                                    --
  //                                                                         --
  //---------------------------------------------------------------------------
  public int indexColour(int colouredIndex) {

    int  colourIndex;
    // Used to index the colour

    int  functionResult;
    // Used to return the function result

    colourIndex = colouredIndex;

    while (colourIndex > Forty_Two) {
       colourIndex = colourIndex - Forty_Two;
    }

    functionResult = RED;
    for (int i = 0; i < colourIndex; i++) {
      functionResult = succ(functionResult);
    }
    return functionResult;

  } // indexColour


} // CT
```

The following source code listing was stored in the file two.java.

```
class


  int[]
  // Identifies an arbitrary array of integer values


  // Define and array of ten integer values

  //---------------------------------------------------------------------------
  //  This method sorts the array into numerically increasing values using    //
  //  the Exchange sort algorithm.                                            //
  //---------------------------------------------------------------------------

  public void

    // Repeat until no array elements were swapped within the For loop


      // Identify that no array elements have been swapped


      // Loop through the array elements


        // Compare the Ith array element against the (I+1)th array element


          // Swap the two array elements


          // Identify that the array elements have been swapped
```

Appendix F - Source Code Editor Software Modules

The Source Code Editor was written in Borland's Delphi Studio version 7 and incorporates Addict Software's spell checking component for execution on a Microsoft Windows platform.

The software design is object based, in that programming objects were identified and functionality was encapsulated into object methods. Programming objects were not instantiated, rather they were simply used.

SCE

This file contains the main entry point for the application and controls the main menu and associated functions, the popup menu, any edit tabs, user keyboard entry and the background task. The background task further controls the parsing of the edited source code into symbols, the subsequent allocation of these symbols to computer language constructs (depending on the particular computer language currently selected), the logging of user actions pertinent to identifier naming, display update to highlight coding features of potential interest to the user, automatic replacement of identifier names and status information update.

Project_Data

This file is used to maintain the default identifier style and currently selected computer language for the source code files being edited by the user.

Editor

This file contains a collection of utility functions used in the manipulation of an edit area. The utility functions include clipboard functions (eg. copy selected text into the clipboard), cursor positioning functions (eg. line/column position set), line control (eg. file line count), file functions (eg. load text file into the edit area), text selection control functions (eg. count the number of characters in the selected text), edit area modification functions (eg. identification of whether the edit area has been modified since last update), text control functions (eg. text cut from the edit area), an edit area verification function and linked list functions (eg. add a new edit area to a list of edit areas).

Heuristic

This file enumerates the identifier-naming style guideline heuristic and contains a collection of utility functions used in the evaluation of the heuristic. These utility functions include functions that converts an enumeration into the corresponding name and maintains a linked list of heuristic (eg. identifies whether a particular list contains a specific heuristic).

Heuristic_Construct

This file contains a collection of utility functions that check the evaluation of the identifier-naming style heuristic and suggest corrections to flawed identifier names.

Heuristic_Control

This file contains a collection of functions that maintain a file record of the identifier-naming style guideline heuristic checking that have been activated/deactivated by the user.

Heuristic_Display

This file contains the functions which control the user interface to the Identifier-Naming Heuristic Activation List form. In particular, there are functions to control the suggestion of replacement identifier names, acceptance of a user modification to the identifier name, heuristic name list display for the identifier-naming style guideline heuristic that have been activated and the display of corresponding information regarding a selected identifier-naming style guideline heuristic.

Heuristic_Update

This file contains the functions which control the user interface to the Identifier-Naming Heuristic Checking form. In particular there are functions to recover to the default checking list and to save the user updates to the list of heuristic that will be checked.

Identifier

This file contains a collection of functions that make recommendations for the correction of identifier names so that the identifier name will cease to cause the particular Identifier-naming style guideline heuristic to activate.

Symbol

This file enumerates the computer language symbols recognised by the source code editor and contains a collection of utility functions used in the manipulation of the

computer language symbols. These utility functions include a set function (i.e., counting the number of symbols in a set of symbols), heuristic reporting control (eg, disabling reporting for one particular instance of an identifier name only), identifier manipulation (eg. replacement of an identifier name with a different name), source code line control (eg. deletion of all the symbols in a particular source code line), and pointer manipulation (eg. disposal of the symbols in a linked list of symbols), symbol comparison (eg. identification of equivalent but different symbols) and validity verification of the remaining symbols in a linked list after modification to an earlier symbol.

Symbol_Compose

This file contains a method that converts a linked list of symbols into the corresponding character string lines that would normally be entered by a user.

Symbol_Decompose

This file contains a method that decomposes (i.e., parses) source code lines into the corresponding list of symbols, maintained in a linked list.

Symbol_Update

This file contains methods for the update control of symbols in an edit area and the logging of user events that cause an identifier-naming style guideline heuristic to activate.

Construct

This file enumerates the computer language construct kinds (i.e., kinds of declarations and kinds of statement), and contains a collection of utility functions used in the manipulation of computer language constructs. These utility functions include functions that mark an identifier symbol as being used, find a particular construct within a construct tree, find a construct containing a particular symbol, identify whether an identifier-naming style guideline heuristic for a declaration has activated, insert/remove a comment symbol, modify an identifier name and support pointer manipulation (eg. disposal of the constructs in a tree).

Construct_Ada

This file contains the methods that parse the following Ada computer language declaration constructs: Entry, For, Function, Generic, Package, Pragma, Procedure, Separate compilation, Subtype, Task, Type, Use, Constant/Variable and With, the Ada

computer language statements: Abort, Accept, Assignment, Case, Declare, Delay, Elsif, Exit, For, Goto, If, Label, Loop, Null, Pragma, Procedure call, Raise, Requeue, Return, Select, Terminate and While, and the method that controls the parsing of a text file.

Construct_Java

This file contains the methods that parse the following Java computer language declaration constructs: Class, Constructor, Import, Interface, Package, Procedure and Constant/Variable, the Java computer language statements: Assignment, Break, Continue, Do, For, If, Label, Procedure call, Return, Switch, Synchronized, Throw, Try and While, and the method that controls the parsing of a text file.

Construct_Pascal

This file contains the methods that parse the following Pascal computer language declaration constructs: Function, Package, Procedure, Type, Use and Constant/Variable, the Pascal computer language statements: Abort, Assignment, Case, For, Goto, If, Label, Procedure call, Repeat, While and With, and the method that controls the parsing of a text file.

Log

This file contains a collection of functions that control the writing of information to be logged concerning particular user actions that relate to the editing of identifier names and the modification to the Identifier-naming style guideline heuristic checking status.

Metric

This file contains a selection of functions that calculate the various source code metrics and maintains counts of whether there has been an opportunity for an Identifier-naming style guideline heuristic to be activated and whether the heuristic was actually activated.

Help

This file contains the display component that controls the display of help information to the user, concerning the operation of the source code editor.

Questionnaire

This file contains the functions which controls the user interface to the questionnaire form. This function controls the display and user responses to the individual questions and writes the user selections to file.

Questionnaire_Help

This file contains the function that controls the display of help information to the user concerning questions that the user may have relating to the questionnaire.

String_List

This file contains a collection of utility functions applicable to maintaining and sorting a linked list of character strings.

Word_List

This file contains a collection of utility functions applicable to maintaining a linked list of words. The utility functions control the reading of words from file, the building of an index, the handling of the underscore character to separate words forming an identifier name and the sorting of words in a linked list.

UFile

This file contains utility functions used in the reading and writing of text files.

UInteger

This file contains utility functions used in the conversion of integer values to character strings and visa versa.

UString

This file contains utility functions used in the manipulation of character string.