2010 14th European Conference on Software Maintenance and Reengineering

Exploring the Influence of Identifier Names on Code Quality: an empirical study

Simon Butler, Michel Wermelinger, Yijun Yu, Helen Sharp
Centre for Research in Computing, The Open University, Milton Keynes, UK

***Abstract*—Given the importance of identifier names and the value of naming conventions to program comprehension, we speculated in previous work whether a connection exists between the quality of identifier names and software quality. We found that flawed identifiers in Java classes were associated with source code found to be of low quality by static analysis. This paper extends that work in three directions. First, we show that the association also holds at the finer granularity level of Java methods. This in turn makes it possible to, secondly, apply existing method-level quality and readability metrics, and see that flawed identifiers still impact on this richer notion of code quality and comprehension. Third, we check whether the association can be used in a practical way. We adopt techniques used to evaluate medical diagnostic tests in order to identify which particular identifier naming flaws could be used as a light-weight diagnostic of potentially problematic Java source code for maintenance.**

***Keywords*-programming; software metrics; software quality;**

I. Introduction

Identifier names constitute the majority of tokens in source code [1] and are the primary source of conceptual information for program comprehension [2]. Identifier names are created by designers and programmers and reflect their understanding, cognition and idiosyncrasies [3]. The impact of low quality identifier names on program comprehension is reasonably well understood [1], [4], [5], but little is known about the extent to which the quality of identifier names might influence the quality of source code.

Given that poor quality identifier names are a barrier to program comprehension, and that they may indicate a lack of understanding of the problem, or the solution articulated in the source code, we hypothesise that poor quality identifier names indicate poor quality source code that translates into poor quality software. In previous work [6], we showed connections between flawed identifier names and FindBugs warnings [7] in Java classes. In this paper we expand on our previous work by investigating the quality of identifiers and source code in Java methods. At this finer-grained level of analysis we employ the cyclomatic complexity metric [8] and the maintainability index [9] to evaluate the quality of source code. In addition, we evaluate the readability of methods using a readability metric [10]. We also repeat our investigation of source code quality with FindBugs [7] with the expectation of finding more focused results because the class-level specific FindBugs warnings included in our previous work, are excluded from this study at the method level. We also seek to verify the link between the readability of source code and FindBugs warnings found by Buse and Weimer [10]. In addition, we explore whether our findings may be applied as a low-cost heuristic to identify potentially problematic regions of source code.

The remainder of this paper is structured as follows: in Section II we examine related work before turning in Sections III and IV to the underlying concepts of identifier and source code quality used in this paper. We give an account of our methodology in Section V and report our results in Section VI. In Sections VII and VIII we discuss our results and draw conclusions.

II. Related Work

Existing research on source code readability focuses on the contribution the components of source code make to readability [10], and the way in which the semantic content of identifiers contributes to readability and program comprehension [1], [5], [2].

A longitudinal study of identifier names by Lawrie et al. [4] showed that identifier name quality has improved during the last thirty years. The same study also found that identifiers in proprietary source code typically contained more domain-specific abbreviations than open source code. However, the study also found that identifiers change little following the initial period of software development. This is confirmed by Antoniol et al. [11] who also argue that programmers may be more reluctant to change identifier names than source code, because of the lack of tool support for managing identifier names. In [5], Lawrie et al. detail an empirical study which found identifier names composed of dictionary words were more easily understood than those composed of abbreviations or single letters.

Rajlich and Wilde emphasise the importance of identifiers as the primary source of conceptual information for program comprehension [2]. Deissenboeck and Pizka [1] developed a formal model for the semantics of identifier names in which each concept is represented by just one identifier throughout a program. The model excludes the use of homonyms and synonyms, thus reducing the opportunities for confusion. The authors found the model to be an effective tool for resolving difficulties with identifier names found during program development.

A study of the morphological and grammatical features of identifier names in C, C++, Java and C# by Liblit et al. [12] found that identifiers are best understood within their

1534-5351/10 $26.00 © 2010 IEEE
DOI 10.1109/CSMR.2010.27

working context. Instance variables, for example, are coupled with method names in object-oriented languages, and method names are often conceived with this relationship in mind. Field and variable names have grammatical structures that reflect their independence. The grammatical structure of method names is further differentiated by the need to reflect the action the method performs and whether it has side effects, or takes one or more arguments.

Relf [13] identified a set of cross-language identifier naming style guidelines from the programming literature, and investigated their acceptance by programmers in an empirical study. Relf implemented the naming style guidelines in a tool to help programmers create good quality identifiers and to refactor existing identifiers [14]. Abebe et al. [15] developed a system to recognise 'lexicon bad smells' – grammatical and other flaws – in identifiers, thereby identifying identifier names for possible refactoring.

Little work, however, has been done to explore the possible connections between identifier naming, source code readability and software quality.

Two studies by Boogerd and Moonen [16], [17] applied the MISRA-C: 2004 coding standard [18] to measure the quality of source code before and after bug fixes during the development of two closed source embedded C applications. They found that while compliance with some of the rules increased as defects were fixed, bug fixes also introduced violations of other rules. In other words, code with fewer defects, and hence of higher quality, is deemed to be of lower quality by some of the coding rules. The authors also found that though they could identify rules with a positive influence on software quality in each of the two studies, the rules did not have consistent effects, including the four rules related to identifiers common to both studies.

Buse and Weimer [10] developed a readability metric for Java derived from measurements of, among others, the number of parentheses and braces, line length, the number of blank lines, and the number, frequency and length of identifiers. Using machine learning, the readability metric was trained to agree with the judgement of human source code readers. Buse and Weimer found a significant statistical relationship between the readability of methods and the presence of defects found by FindBugs [7] in open source code bases. Although their work makes a link between readability and software quality, their notion of readability ignores the quality of identifier names.

In work classifying the lexicon used in Java method identifiers, Høst and Østwold advance the idea that, because of the effort required to select a good identifier name, identifiers reflect the cognitive processes of programmers and designers [3]. Consequently, identifiers may then reflect the misunderstandings of the creator of the identifier and misdirect the readers of source code.

The existing literature establishes the need for good identifier names to support program comprehension. However, only tentative steps have been taken to demonstrate their relationship to source code quality. In previous work [6], we explored the relationship between flawed identifiers and FindBugs defects in Java classes. We found some relationships, which we explore further in this paper with finer-grained analysis, and by increasing the number of metrics used to measure source code quality.

III. Identifier Quality

The multifactorial nature of identifier quality makes measurement problematic. For the purposes of this study we constrained our measurement of identifier quality to typography and the use of known natural language elements, and ignored detailed assessments of semantic content and the use of grammar. Rather than apply an arbitrary set of rules derived from a single set of programming conventions, we used a set of empirically evaluated identifier naming guidelines.

Relf derived a set of twenty-one identifier naming style guidelines for Ada and Java from the programming literature [13]. Most of the guidelines, which were evaluated in an empirical study, do not deviate significantly from the Java identifier naming conventions [19], [20] and as they have been developed in other widely used conventions [21].

Relf's identifier naming style guidelines combine typography and a simple approach to natural language, but were not intended to be used as rules to evaluate the quality of identifier names. Accordingly we found it necessary to update some guidelines to define more precisely what was not permitted, and renamed some to reflect the proscriptive sense in which we applied them.

We implemented a subset of Relf's guidelines as tests. The remaining guidelines were not adopted because either they do not reflect recent changes in Java programming practice, or they are general guidelines of good practice that are difficult to derive practical proscriptive rules from. For example, Relf defines the Same Words guideline as prohibiting the use of identifiers composed of the same words, but in a different order. Whilst superficially attractive, a rule based on this guideline prohibits clear names for reciprocal operations (e.g. `htmlToXml` and `xmlToHtml`) and pairs of words that create semantically distinct identifiers (e.g. `indexPage` and `pageIndex`). Generally, the implementation of each guideline is apparent from its name, and is described and illustrated in Table I. However, the precise implementation of some guidelines requires further explanation:

Capitalisation Anomaly: For identifiers other than constants we test for capitalisation of only the initial letter of acronyms as prescribed in [19], [20], i.e. only the initial letter of a component word is capitalised either at word boundaries, or the beginning of the identifier, if appropriate.

Non-Dictionary Words: We defined a dictionary word as belonging to the English language, because all the projects investigated are developed in English. We constructed a dictionary consisting of some 117,000 words, including

Table I
THE IDENTIFIER NAMING STYLE GUIDELINES APPLIED

Name	Description	Example of flawed identifier(s)
Capitalisation Anomaly	Identifiers should be appropriately capitalised.	`HTMLEditorKit`, `pagecounter`, `fooBAR`
Excessive Words	Identifier names should be composed of no more than four words or abbreviations.	`floatToRawIntBits()`
External Underscores	Identifiers should not have either leading or trailing underscores.	`_foo_`
Long Identifier Name	Identifier names of more than twenty-five characters should be avoided where possible.	`getPolicyQualifiersRejected`
Naming Convention Anomaly	Identifiers should not consist of non-standard mixes of upper and lower case characters.	`FOO_bar`
Non-Dictionary Words	Identifier names should be composed of words found in the dictionary and abbreviations and acronyms that are more commonly used than the unabbreviated form.	`strlen`
Number of Words	Identifiers should be composed of between two and four words.	`ArrayOutOfBoundsException`, `name`
Numeric Identifier Name	Identifiers should not be composed entirely of numeric words or numeric words and numbers.	`FORTY_TWO`
Short Identifier Name	Identifiers should not consist of fewer than eight characters, with the exception of `c, d, e, g, i, in, inOut, j, k, m, n, o, out, t, x, y, z`	`name`
Type Encoding	Type information should not be encoded in identifier names using Hungarian notation or similar	`iCount`

inflections and American and Canadian English spelling variations, using word lists from the SCOWL package up to size 70, the largest lists consisting of words commonly found in published dictionaries [22]. We added a further 90 common computing and Java terms, e.g. 'arity', 'hostname', 'symlink', and 'throwable'. A separate dictionary of abbreviations was constructed, using the criterion that "the abbreviation is much more widely used than the long form, such as URL or HTML" [20].

A concern is that development teams may use project, or domain, specific abbreviations and terms, which are not in our dictionary, yet are well understood by the programmers. To address the issue we created additional dictionaries for each application of unrecognised component words that were used in three, five and ten or more unique identifiers. For example, an unrecognised word or abbreviation used in ten or more unique identifiers may be inferred to be a commonly understood term. The frequencies of three, five and ten are arbitrary, but may be seen as representative of the familiarity the development team might have with a given term. Following the creation of the dictionaries, each identifier was tested again for compliance to the Non-Dictionary Words guideline by using a combination of the main dictionary, the abbreviation dictionary, and each of the dictionaries of application-specific words and abbreviations.

Number of Words: Relf's Number of Words guideline was intended to encourage programmers to create identifiers between two and four words long. In applying the guideline as a proscriptive rule both identifiers composed of one word and those composed of five or more words are categorised together, which does not allow us to determine the contribution made by the occurrence of either. The issue is addressed, in part, by the creation of an Excessive Words flaw, defined in Table I, which determines identifiers of five or more words to be flawed.

Short Identifier Name: We updated Relf's guideline to include more single letter and short identifiers commonly used in Java [20], [21] (see Table I).

IV. SOURCE CODE QUALITY

Our objective is to measure source code quality in a way that reflects the influence of the programmer on source code and the possible impact on the reader. We used cyclomatic complexity [8] and the three metric maintainability index [9] to measure the quality of Java methods. Additionally, Buse and Weimer's readability metric (see Section II) was used to provide assessments of the readability of methods. We also used FindBugs to analyse the bytecode of each application for any potential defects.

Cyclomatic complexity provides a ready assessment of the complexity of a method in terms of the number of possible execution paths. We acknowledge that cyclomatic complexity is a somewhat controversial metric [23], but believe that it provides an indication of source code complexity sufficient for our purposes.

The three metric maintainability index (MI) [9] is given by:

$$MI = 171 - 5.2 \times \ln(HV) - 0.23 \times V(G) - 16.2 \times \ln(LOC)$$

where LOC is the number of lines of code, V(G) is the cyclomatic complexity and HV is the Halstead Volume [24], a source code metric determined by the number of operators and operands used, including identifiers. The Halstead volume is the product of the Halstead Vocabulary and the logarithm of the Halstead Length. The Halstead Vocabulary is the number of unique operators and unique operands, and the Halstead Length is the sum of the number of operators and operands. By incorporating the Halstead Vocabulary, the MI is influenced by the complexity of a unit of source code in terms of the number of identifiers required to implement a solution.

FindBugs is a static analysis tool for Java that analyses bytecode for 'bug patterns'. The type of defects identified by the bug patterns range from dereferences of null pointers, which may halt program execution, to Java specific problems associated with an incomplete understanding of the Java language [25]. The latter class of defects include code constructs likely to increase the maintenance effort and code constructs that may have unintended side-effects. FindBugs was used extensively during two days in May 2009 at Google, and software engineers found some 4,000 significant issues with Java source code as a result [7]. While we accept that FindBugs creates false positives, as does any static analysis tool, we feel that FindBugs' perspective on source code quality is suitable for our needs.

V. Methodology

A. Data Collection

We selected eight established Java open source projects for investigation, including GUI applications, programmers' tools, and libraries. The particular projects were chosen to reduce the potential influence of domain and project-specific factors in this study. Table II shows the version and number of methods analysed for each project.

Table II
Source Code Analysed

Project	Version	Methods
Ant	1.71	9146
Cactus	1.8.0	926
Freemind	0.9.0 Beta 20	4883
Hibernate Core	3.3.1	12309
JasperReports	3.1.2	12349
jEdit	4.3 pre16	5835
JFreeChart	1.0.11	8230
Tomcat	6.0.18	11394

We developed a tool to automate the extraction and analysis of identifiers from Java source code. Java files were parsed and identifiers analysed on the parse tree to establish adherence to the typographical rules for their context, e.g. method names starting with a lowercase character. Then, identifiers were extracted and added to a central store, with information about their location, and divided into hard words – their component words and abbreviations – using the conventional Java word boundaries of internal capitalisation and underscores. Identifiers were then analysed by our tool for conformance to Relf's guidelines in Table I, our own Excessive Words guideline, and the Non-Dictionary Words guideline where the dictionary is extended by a set of commonly used hard words.

Where subject applications were found to contain source code files generated by parser generators, or to incorporate source code from third party libraries, those files were ignored to try to ensure only source code written by the applications' development teams was analysed.

We collected the primitive Halstead metrics for each method – counts of operators and operands – by adapting the standard developed for C by Munson [23] and applying it to Java in our tool. We also recorded McCabe's cyclomatic complexity (V(G)) [8] and LOC for each method, to compute the maintainability index. To create a binary classifier from the maintainability index we used the threshold of 65, established by empirical study [9], to identify methods as 'more-maintainable' and 'less-maintainable'.

The readability of source code was evaluated using a readability metric tool developed by Buse and Weimer [10]. The readability metric follows a bimodal distribution and is interpreted as binary classifier that identifies source code as 'more-readable' or 'less-readable'

We also applied the cyclomatic complexity metric as a binary classifier. The popular programming literature often advocates that programmers take steps to keep the cyclomatic complexity of individual methods low. Some texts suggest refactoring should be considered when cyclomatic complexity is six or more, and that the cyclomatic complexity of a method should not exceed ten [26]. It is outside the scope of our study to examine the merits of such practices or the justification for the chosen thresholds. However, to create binary classifiers from the cyclomatic complexity metric, we adopted thresholds of six and ten to represent methods of moderate and high complexity. This provides two binary classifiers distinguishing between methods with low complexity and those with a cyclomatic complexity of six or more, and between methods with low to moderate complexity and those with a cyclomatic complexity of ten or more.

For the purposes of this study we recorded details for methods that constitute discrete readable units to ensure that the readability metric assessed source code as the human reader would see it. Java source code files contain one or more top-level classes, each of which may contain member classes. Both types of classes may contain methods. We recorded as methods, only methods contained either by top-level classes or by member classes directly contained by top-level classes. Any local and anonymous classes contained within those methods were recorded as part of the containing method and not separately. For example, if a method contains an anonymous class, the total cyclomatic complexity for the anonymous class is added to the cyclomatic complexity of the containing method.

The Java archive (JAR) files resulting from the compilation of the source code were analysed with FindBugs.

FindBugs employs a heuristic to determine the severity of the defects it finds and, in its default mode, issues 'priority one' and 'priority two' warnings, with priority one deemed the more serious. Counts of priority one and priority two warnings were recorded for each method. We used the default settings for FindBugs with the exception of a filter to exclude warnings of the use of unconventional capitalisation of the first letter in class, method and field names, which would overlap with the findings of our tool. We also filtered out the 'Dead Local Store' warning, which can result from the actions of the Java compiler. We found that FindBugs warnings are sparsely distributed in Java methods and used the presence of a FindBugs warning as a binary classifier.

The identifier naming and metrics data collected for each Java method was stored in XML files and collated with the XML output of FindBugs and the readability metric tool, using a tool we developed. Data extracted from the source code was matched with classes recorded by FindBugs to ensure that only identifiers from classes compiled into the JAR files were analysed. The collated data for each method was then written to R [27] dataframes for statistical analysis.

B. Statistical Analysis

For each pair of binary classifiers, a contingency table like Table III was created using R, and the chi-squared (χ^2) test [28] was performed, with the null hypothesis that the binary classifiers were independent. For Table III the value of χ^2 is 81.2, which is statistically significant ($p = 2 \times 10^{-19}$). For each contingency table, a table of expected values was derived from the marginal totals to help determine the nature of any association. In Table III our interest lies in the top-left cell; if the observed frequency exceeds the expected frequency then there is a statistically significant association between the presence of identifiers with the Non-Dictionary Words flaw and FindBugs Priority Two warnings in a method. The expected value for the top-left cell is the product of the sum of the observed values in the lefthand column and the top row divided by the total population, i.e. $(103+37)\times(103+2925)\div(103+37+2925+5165) = 51.5$, which is less than observed frequency of 103. Where any of the expected frequencies for a contingency table were less than five, the Fisher exact test [28] was used.

Table III
EXAMPLE CONTINGENCY TABLE

JFreeChart **Non-Dictionary Words**	**FindBugs Priority Two Warnings** **methods with**	 **methods without**
methods with	103	2925
methods without	37	5165

In addition to the χ^2 tests, we applied a technique used in medicine to evaluate diagnostic tests to determine whether the observed phenomena have a practical application. The same contingency tables used for the χ^2 tests were analysed by treating FindBugs warnings, the maintainability index, cyclomatic complexity and readability as reference classifiers. For example, for the contingency table above (Table III) we take the occurrence of FindBugs priority two warnings in methods as the reference classifier, and test to see how well the Non-Dictionary Words flaw performs as a classifier in comparison.

To evaluate the relative performance of the test classifier, two quantities are derived from the contingency table: the sensitivity and the specificity, which represent agreement between the two classifiers. The sensitivity is the proportion of the population classified as positive by the reference classifier that are classified positively by the classifier being tested. In our example in Table III, the sensitivity is the proportion of methods for which FindBugs warnings are issued, that also contain identifiers with the Non-Dictionary Words flaw; i.e. $sensitivity = 103 \div (103 + 37) = 0.74$. The specificity is the proportion of population classified negatively by the reference classifier that are also classified negatively by the test classifier. In Table III, the specificity is the proportion of the methods without FindBugs priority two warnings that have no identifiers with the Non-Dictionary Words flaw: $specificity = 5165 \div (2925 + 5165) = 0.64$. An advantage of this method is that sensitivity and specificity are independent of the rate of incidence, or prevalence, of the phenomenon being investigated.

The characteristics of a given test can be illustrated using receiver operating characteristic (ROC) curves, where the $sensitivity$ of a test is plotted on the y-axis, against $1 - specificity$ (the false positive rate) on the x-axis. The area under the curve (AUC) (see Figure 1) indicates the efficacy of the test. A useless test, one that is equivalent to guessing, is indicated by a diagonal line drawn from the origin to the top-right corner, representing the equation $sensitivity = 1 - specificity$, which has an AUC of 0.5. For a test to be useful the points plotted should lie above and to the left of the diagonal line. We use the ROC graphs as a means of visualising the predictive power of the observed associations.

Our example results in a point at $(0.36, 0.74)$, above and to the left of the diagonal, meaning that, in the case of JFreeChart, using the Non-Dictionary Word flaw as a binary classifier is a *better than chance* method of predicting the presence or absence of FindBugs priority two warnings. The predictive power of a result is related to its perpendicular distance from the diagonal line, and is equal to the area under a line drawn from the origin to the point representing the result and from the result to the point $(1, 1)$. In our example, the predictive power is 0.69, which means that the Non-Dictionary Word flaw has a 0.69 probability of indicating whether or not a method contains a FindBugs Priority two warning in JFreeChart.

The majority of methods in JFreeChart are correctly classified by the test classifier and are grouped in the top-left and bottom-right cells of Table III. As we will see in the next section, especially for Cactus, it is possible for the members of a population to be grouped in these cells, resulting in values of *sensitivity* and *specificity* that give a useful probability, without the distribution in the contingency table giving a statistically significant result for either the χ^2 or Fisher exact tests.

VI. Results

In Tables IV, V and VI statistically significant associations between the flawed identifiers and each of the source code quality measures are represented in black where $p < 0.001$ and dark grey where $p < 0.05$. Where the trend of association was negative, i.e. the presence of the particular identifier flaw is associated with better quality source code, the cell is marked with a white dash. White cells represent the lack of a statistically significant association (i.e. $p > 0.05$), and asterisks indicate where the particular identifier flaw was not found. The digits contained in selected cells show the probability with which the identifier flaw, when applied as a binary classifier, correctly predicts the quality of methods. Only probabilities of 0.55, marginally better than guessing, or greater, have been included in the tables. The probabilities not shown are largely close to 0.5, and only less than 0.5 for some of the negative associations.

Each table lists three further categories labelled ‘Extended 3’, ‘Extended 5’ and ‘Extended 10’. The results for the three ‘Extended’ flaws should be compared with those for the Non-Dictionary Words flaw to determine the influence of application-specific words and abbreviations on the relationship between the linguistic content of identifiers and FindBugs warnings. The bottom line of Table IV shows the relationships between methods classified as less-readable by the readability metric and FindBugs warnings. Where we found associations, our results largely confirm the connection between readability and FindBugs warnings found by Buse and Weimer [10]. Indeed, our results show that the connection between readability and FindBugs warnings extends to projects such as Ant and Freemind, which Buse and Weimer did not investigate. However, our work differs in the statistical methods used, the versions of projects investigated, and because we discriminated between priority one and two warnings, which they did not.

Table IV shows the associations between identifier flaws and FindBugs priority one and priority two warnings in the methods of each project. The statistical associations are largely confined to particular identifier flaws indicating the general cross-project trends. However, there are also apparent project-specific relationships as illustrated by Cactus and jEdit for both priority one warnings, and Cactus, Hibernate and JasperReports for priority two warnings.

While Cactus and jEdit have just one statistically significant association with priority one warnings between them, we found useful predictive qualities in the relationships for some identifier flaws. The probabilities given in the left hand side of Table IV emphasise the cross-project nature of the relationships between the Extended, Non-Dictionary Words, Number of Words and Short Identifier flaws. The relationships for the priority two warnings are less clear. There are hints of similar, general, cross-project relationships; however, the project-specific relationships are more apparent. Cactus, again, has few statistical associations, but some relationships have probabilities greater than 0.55. Hibernate and JasperReports both have negative statistical associations. Hibernate has a few relationships with probabilities greater than 0.55, whereas JasperReports has none.

The relationships for the Non-Dictionary Words flaw and priority two warnings are plotted in Figure 1. While six points are above the diagonal line and illustrate the utility of the Non-Dictionary Words flaw as a light-weight classifier, there are two points below the line. The point for Hibernate, where no statistically significant association was found, is closest to the line and the other is for JasperReports which has a negative association.

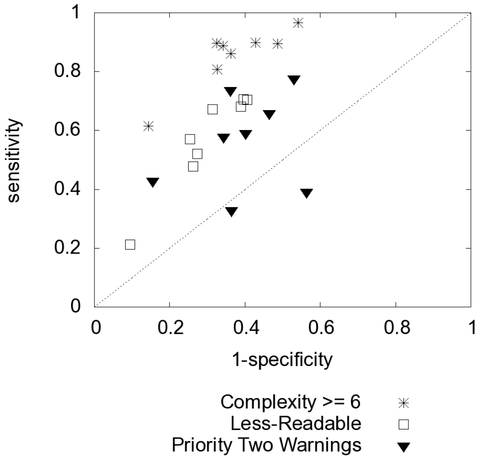

Figure 1. ROC Plot for the Non-Dictionary Words Flaw

Tables V and VI show much more consistent relationships for identifier flaws with complexity, maintainability and readability. There remain, however, hints of project-specific relationships, which are most apparent for Cactus. The predictive probability associated with each relationship illustrates the utility of the identifier flaws as light-weight classifiers for source code quality. The relationships between the Non-Dictionary Words flaw and complexity and readability are plotted in Figure 1.

A. Threats to Validity

Construct Validity: The definition of the Short Identifier Name guideline is much more restrictive than the Java programming conventions [20], [21] and common practice. Consequently the number of identifiers categorised as flawed

Table IV
ASSOCIATIONS BETWEEN NAMING FLAWS AND PRIORITY ONE AND TWO WARNINGS

	Priority One Warnings								Priority Two Warnings							
	Ant	Cactus	Freemind	Hibernate	JasperReports	jEdit	JFreeChart	Tomcat	Ant	Cactus	Freemind	Hibernate	JasperReports	jEdit	JFreeChart	Tomcat
Capitalisation Anomaly	▨ .71			▨ .63	▨ .59			.56	■ .62		■ .62	▨ –	▨ –			■ .57
Excessive Words			.55						■ .55	.55		■ .58	■ –			
External Underscores		*		*	*		*			*		*	*		*	
Long Identifier					■ .59					.59		■ .57	■ –			
Naming Convention Anomaly									▨					▨		
Number of Words	.57	.61	▨ .62	▨ .62	■ .64	.56	■ .59	▨ .55	■ .56		■ .59	■ –		▨	▨ .55	■ .55
Numeric Identifier	▨ .55	*	*			*		*		*	*			*		*
Short Identifier Name	.59	.64	▨ .63	▨ .65	■ .66		■ .61	■ .59	■ .56	.58	■ .62	■ –		▨	■ .56	■ .57
Type Encoding		*		*				*		*		*				*
Non-Dictionary Words	▨ .72	.92	■ .71	■ .70	■ .66	.60	■ .81	▨ .57	■ .60	.64	■ .62		■ –	■ .63	■ .69	■ .59
Extended 3	▨ .71	.94	■ .66	■ .81				▨ .55	■ .64	▨ .66	■ .59			■ .63		■ .59
Extended 5	■ .76	.94	■ .66	■ .80	▨	.57	■ .88	▨ .56	■ .64	▨ .65	■ .64	▨ –		■ .63	■ .72	■ .59
Extended 10	▨ .72	.92	▨ .65	■ .75		▨ .67	■ .87	.55	■ .63	.64	■ .64	■ –	▨	■ .61	■ .72	■ .61
Less-readable	■ .82	.74	■ .72	▨ –	■ .65		■ .72	■ .60	■ .67		■ .67	■ .67	■ –		■ .66	■ .68
	■	$p < 0.001$			▨	$p < 0.05$				$p >= 0.05$			*	No flaw		

Table V
ASSOCIATIONS BETWEEN NAMING FLAWS AND CYCLOMATIC COMPLEXITY

	Cyclomatic Complexity $>= 6$								Cyclomatic Complexity $>= 10$							
	Ant	Cactus	Freemind	Hibernate	JasperReports	jEdit	JFreeChart	Tomcat	Ant	Cactus	Freemind	Hibernate	JasperReports	jEdit	JFreeChart	Tomcat
Capitalisation Anomaly	■ .68	■ .68	■ .65	■ .65	■ .65	■ .61	■ .68	■ .73	■ .67	■ .72	■ .63	■ .64	■ .66	■ .61	■ .73	■ .75
Excessive Words	■	▨	■ .58	■ .62	■ .55	■	■ .58	■	■ .55	▨ .55	■ .58	■ .65	■ .58	■	■ .60	■
External Underscores		*		*	*	■	*	■		*		*	*	■	*	▨
Long Identifier	■	▨ .56	■ .56	■ .64	■ .62	■	■ .57	■ .55	■	.56	■ .57	■ .68	■ .66	■	■ .58	■ .57
Naming Convention Anomaly	■		■	■	■	■	▨	■	■			■	■	■ .55	■	■
Number of Words	■ .56	■ .62	■ .57	■ .61	■ .65	■	■ .59	■ .60	■ .55	▨ .61	■ .57	■ .60	■ .64	■	■ .58	■ .59
Numeric Identifier		*	*			*	▨	*		*	*			*		*
Short Identifier Name	■ .58	■ .65	■ .58	■ .64	■ .64	■ .55	■ .61	■ .61	■ .63	▨ .65	■ .57	■ .62	■ .62	■ .55	■ .60	■ .62
Type Encoding	▨	*		*			■	*		*		*	▨		■	*
Non-Dictionary Words	■ .70	■ .65	■ .68	■ .75	■ .69	■ .64	■ .78	■ .74	■ .67	■ .70	■ .67	■ .74	■ .70	■ .64	■ .78	■ .76
Extended 3	■ .69	■ .65	■ .63	■ .69	■ .65	■ .62	■ .69	■ .72	■ .69	■ .70	■ .61	■ .73	■ .68	■ .64	■ .75	■ .75
Extended 5	■ .71	■ .64	■ .65	■ .72	■ .69	■ .64	■ .80	■ .73	■ .70	■ .69	■ .65	■ .75	■ .73	■ .66	■ .82	■ .76
Extended 10	■ .71	■ .65	■ .66	■ .74	■ .70	■ .65	■ .80	■ .74	■ .70	■ .70	■ .66	■ .76	■ .74	■ .66	■ .81	■ .77
	■	$p < 0.001$			▨	$p < 0.05$				$p >= 0.05$			*	No flaw		

may be inflated, and accordingly the observed associations may need to be treated with caution.

False positives are inevitable with static analysis tools such as FindBugs. The false positive rate for each application cannot be established without manual inspection of the source code in the proximity of each warning, which is outside the scope of the current study.

External Validity: The apparently project-specific influences on the relationships between flawed identifiers and FindBugs warnings in Table IV, suggest that, though general principles may be derived from our findings, caution is necessary when applying them to other projects. Some project-specific variation is apparent even in the more consistent findings shown in Tables V and VI, again suggesting that

Table VI
ASSOCIATIONS BETWEEN NAMING FLAWS AND READABILITY AND THE MAINTAINABILITY INDEX

	Less-Readable								Less-Maintainable							
	Ant	Cactus	Freemind	Hibernate	JasperReports	jEdit	JFreeChart	Tomcat	Ant	Cactus	Freemind	Hibernate	JasperReports	jEdit	JFreeChart	Tomcat
Capitalisation Anomaly	■ .62	■ .55	■ .61	■ .60	■ .62	■ .62	■ .63	■ .66	■ .78	■ .78	■ .76	■ .67	■ .67	■ .64	■ .81	■ .77
Excessive Words	■	■	■ .59	■ .58	■ .61	■	■ .57	■	■ .59	▨ .58	■ .67	■ .68	■ .62	■ .57	■ .63	■ .55
External Underscores		*		*	*	■	*	■		*		*	*	■ .57	*	▨
Long Identifier	■	■ .56	■ .58	■ .60	■ .58	■	■ .56	■ .56	■ .57	■ .68	■ .67	■ .73	■ .71	■ .57	■ .61	■ .58
Naming Convention Anomaly	■		▨	■	■	■		■	■ .55		■	■ .57	■ .56	■ .55	■	■
Number of Words	■		■ .56	■	■ .60	■	■	■ .55	■ .57	▨ .61	■ .62	■ .62	■ .65	■ .56	■ .59	■ .60
Numeric Identifier	■	*	*		■	*		*	▨	*	*			*		*
Short Identifier Name	■		■	■	■	■		■ .57	■ .59	▨ .65	■ .62	■ .65	■ .66	■ .56	■ .61	■ .63
Type Encoding	▨	*		*			▨	*	▨	*		*	▨		■	*
Non-Dictionary Words	■ .65	■ .56	■ .61	■ .66	■ .65	■ .65	■ .62	■ .68	■ .76	■ .77	■ .79	■ .82	■ .72	■ .72	■ .80	■ .78
Extended 3	■ .62	■	■ .56	■ .58	■	■ .62	■ .60	■ .65	■ .81	■ .76	■ .69	■ .83	■ .72	■ .71	■ .84	■ .80
Extended 5	■ .64	■	■ .57	■ .60	■	■ .63	■ .63	■ .66	■ .82	■ .76	■ .75	■ .85	■ .78	■ .74	■ .85	■ .80
Extended 10	■ .65	■ .56	■ .58	■ .63	■	■ .65	■ .63	■ .68	■ .80	■ .77	■ .77	■ .85	■ .80	■ .74	■ .84	■ .80

■ $p < 0.001$ ▨ $p < 0.05$ $p >= 0.05$ * No flaw

care may be required when applying our findings.

VII. DISCUSSION

The statistically significant associations found for FindBugs priority one and two warnings contain common features (Table IV). There appear to be general, cross-project associations for some identifier flaws, but the distribution of associations appears to be largely project specific. Cactus is the most extreme example with statistically significant associations found with the χ^2 and Fisher exact tests only between the extended dictionaries and priority two warnings. jEdit has only one statistically significant association with priority one warnings, but more with priority two. The negative associations in Table IV (marked with white dashes) emphasise the application-specific nature of some relationships. That the negative associations are positive for the more serious priority one warnings, suggests that the developers in both projects face more complex issues with identifiers than we can explain without further investigation.

The negative associations for the Excessive Words and Long Identifier flaws for JasperReports may be connected through the widespread use of longer identifier names, with which the development team have become familiar. The negative association for the Non-Dictionary Word flaw is not found with the lower frequency extended dictionaries and becomes a positive association with the 'Extended 10' flaw, indicating the importance of a widely used application-specific terms in JasperReports. The use of application-specific terms is consistent with the commercialised nature of JasperReports and the finding of Lawrie et al. [4] that domain-specific natural language and abbreviations are more common in identifiers found in commercial source code than in open source.

In previous work [6], conducted at the class level on the same projects, we found fewer relationships between identifier flaws and priority one warnings, and more general relationships with priority two warnings. At the method level a proportion of FindBugs warnings, which apply only to classes, are eliminated from the study. The finer-grained analysis could be the sole explanatory factor for the difference between the two sets of results for FindBugs warnings. However, it is possible that FindBugs warnings applicable at the class level alone, may have been a source of noise.

The evaluation of the predictive quality of each relationship offers further insights. Some relationships, despite the statistical independence of the two classifiers, may be applied as heuristics. The Non-Dictionary Word flaw for Cactus, for example, could be applied as reasonably reliable classifier of source code for FindBugs priority one warnings, with a probability of > 0.9. In general, the Non-Dictionary Words flaw is a fair to good classifier for FindBugs warnings; however, it is not perfect. The Number of Words and Short Identifiers flaws are much weaker classifiers, with probabilities largely between 0.55 and 0.60, but are still better than guessing.

Tables V and VI show largely consistent associations between the presence of identifier flaws and lower quality source code. In both cases the Capitalisation Anomaly and Non-Dictionary Words flaws provide the stronger classifiers. For complexity and maintainability the Excessive Words, Long Identifier Name, Number of Words, and Short Identifier Name flaws also perform better than chance. However, only the Capitalisation Anomaly and Non-Dictionary Words flaws have consistent relationships with readability.

Identifier length is the only characteristic of individual identifiers that is a component of the readability metric. However, the readability metric developers found that identifier length was not a significant influence on the readability of source code [10]. Our findings, shown in the left hand side of Table VI, suggest the human subjects, against whose judgements of source code readability the metric was trained, were influenced by the conformance of identifier names to familiar typographical conventions, and the use of dictionary words and well-known abbreviations. Further, our findings suggest that longer identifiers do have a negative influence on readability, as evidenced by the statistical associations found for the Excessive Words and Long Identifier flaws in Table VI.

The ROC plots for the Non-Dictionary Words flaw in Figure 1 illustrate that the flaw may be applied to predict lower quality source code. Tables V and VI record probabilities generally greater than 0.6 and sometimes as high as 0.8, showing that the Non-Dictionary Words flaw provides a usable, light-weight classifier for the complexity, maintainability and readability of source code. The probabilities for other identifier flaws given in Tables V and VI show similar predictive values for identifying less-readable, less maintainable and more complex source code. However, the probabilities given in Table IV show that identifier flaws may not be reliably used to predict FindBugs warnings, because of the variation between projects. We previously reported [6] that the Cactus project requires the use of static style checking before code is committed to version control, which influences identifier quality. Also, the commercialised nature of the Hibernate and JasperReports projects may influence the composition of their identifiers [4]. It may be that there are relevant project or domain specific factors into which our current study cannot offer any insights. Boogerd and Moonen [16], [17] attributed many of the differences in their studies to ‘domain factors’. As we deliberately chose not to include projects from identical domains, our results cannot offer clear conclusions on this question.

VIII. Conclusions

The literature establishes the importance of identifier naming to program comprehension [2], [5]. However, there have been few investigations of the relationship between identifier name quality and source code quality [6], [16]. The contribution of this study is to provide a deeper understanding of this important but largely unexplored relationship.

Our investigation was conducted at a finer-granularity than previous work [6], using a variety of source code quality measures, to gain a richer perspective and discriminate among potentially confounding factors. We evaluated the quality of identifier names using accepted naming conventions validated by empirical study [13], and the natural language content of identifiers, including Java- and application-specific terms.

We evaluated source code quality using four perspectives: the identification of potentially problematic code with FindBugs, the three-metric maintainability index, a human-trained readability metric, and cyclomatic complexity. We used the χ^2 and Fisher exact tests to test the independence of poor quality identifiers and more-complex, less-maintainable, and less-readable source code. We found, generally, that poor quality identifiers are associated with lower quality source code. To establish whether the observed associations might have a practical application, we applied a technique used in medicine to evaluate diagnostic tests. We found that some associations occurred with sufficient consistency that they could be applied in a practical setting to identify areas of source code as candidates for intelligent review. We also found that some relationships not found to be statistically significant with the χ^2 and Fisher exact tests were potentially useful classifiers.

We investigated 8 open source Java applications using 10 identifier flaws, and the 3 extended dictionaries, and 6 indicators of source code quality. From our analysis of the 624 relationships, the following lessons for researchers and developers emerge:

- poor quality identifier names are strongly associated with more-complex, less-readable and less-maintainable source code;
- the use of natural language and recognised abbreviations in identifier names may be applied as a light-weight classifier for source code quality;
- the length of identifiers, both in terms of characters and number of component words, can be applied as a light-weight classifier for complexity and maintainability;
- poor quality identifier names are associated with FindBugs warnings; however, the relationships are complex and appear to be application-specific; and
- the only negative associations found were in commercialised projects, indicating there may be relevant differences between open source and commercial code.

Previous work has provided limited perspectives on the relationships between identifier naming, readability and source code quality. Identifiers formed a small part of Boogerd and Moonen’s [17] study of programming conventions and software quality. Buse and Weimer [10] found a relationship between source code readability and FindBugs warnings, and we [6] found associations between identifier quality and FindBugs warnings. This paper is the first to associate multiple naming and source code quality factors at a finer level of granularity. By working at the level of Java methods we were able to investigate the relationships in detail and to provide practical, light-weight and low-cost classifiers for identifying source code which is potentially less-maintainable, less-readable, more-complex and more fault-prone. Further work is required to expand on our findings through the use of other source code quality metrics, including bug reports, the inclusion of semantic information in the measurement of identifier quality, and the investigation of commercial, closed source projects.

Acknowledgements

We thank Álvaro Faria, coordinator of the Statistics Advisory Service at The Open University, for his help in choosing the χ^2 statistical method. We also thank Ray Buse and Westley Weimer of the University of Virginia for allowing us to use their readability metric tool. Finally, we thank the anonymous reviewers for their thoughtful comments, which have helped improve this paper.

References

[1] F. Deissenboeck and M. Pizka, “Concise and consistent naming,” *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, Sep 2006.

[2] V. Rajlich and N. Wilde, “The role of concepts in program comprehension,” in *Proc. 10th Int’l Workshop on Program Comprehension*. IEEE, 2002, pp. 271–278.

[3] E. W. Høst and B. M. Østvold, “The programmer’s lexicon, volume 1: the verbs,” in *Proc. Int’l Working Conf. on Source Code Analysis and Manipulation*. IEEE, October 2007, pp. 193–202.

[4] D. Lawrie, H. Feild, and D. Binkley, “Quantifying identifier quality: an analysis of trends,” *Empirical Software Engineering*, vol. 12, no. 4, pp. 359–388, 2007.

[5] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, “What’s in a name? A study of identifiers,” in *14th IEEE Int’l Conf. on Program Comprehension*. IEEE, 2006, pp. 3–12.

[6] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, “Relating identifier naming flaws and code quality: an empirical study,” in *Proc. of the Working Conf. on Reverse Engineering*. IEEE, 2009, pp. 31–35.

[7] FindBugs, “Find Bugs in Java programs,” http://findbugs.sourceforge.net/, 2008.

[8] T. J. McCabe, “A complexity measure,” *Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[9] K. D. Welker, P. W. Oman, and G. G. Atkinson, “Development and application of an automated source code maintainability index,” *Journal of Software Maintenance*, vol. 9, no. 3, pp. 127–159, 1997.

[10] R. P. Buse and W. R. Weimer, “A metric for software readability,” in *Proc. Int’l Symp. on Software Testing and Analysis*. ACM, 2008, pp. 121–130.

[11] G. Antoniol, Y.-G. Gueheneuc, E. Merlo, and P. Tonella, “Mining the lexicon used by programmers during sofware [sic] evolution,” in *Proc. of Int’l Conf. on Software Maintenance*. IEEE, Oct. 2007, pp. 14–23.

[12] B. Liblit, A. Begel, and E. Sweetser, “Cognitive perspectives on the role of naming in computer programs,” in *Proc. 18th Annual Psychology of Programming Workshop*. Psychology of Programming Interest Group, 2006.

[13] P. A. Relf, “Achieving software quality through identifier names,” 2004, presented at Qualcon 2004 http://www.aoq.asn.au/conference2004/conference.html.

[14] ——, “Tool assisted identifier naming for improved software readability: an empirical study,” in *Int’l Symp. on Empirical Software Engineering*. IEEE, 2005, pp. 53–62.

[15] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, “Lexicon bad smells in software,” in *Proc. Working Conf. on Reverse Engineering*. IEEE, 2009, pp. 95–99.

[16] C. Boogerd and L. Moonen, “Assessing the value of coding standards: An emprical study,” in *Proc. Int’l Conf. on Software Maintenance*. IEEE, 2008, pp. 277–286.

[17] ——, “Evaluating the relation between coding standard violations and faults within and across software versions,” in *Proc. of the Int’l Working Conf. on Mining Software Repositories*. IEEE, 2009, pp. 41–50.

[18] MIRA Ltd, *MISRA-C:2004 Guidelines for the use of the C language in Critical Systems*, MIRA Std., Oct 2004. [Online]. Available: www.misra.org.uk

[19] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java language specification*, 3rd ed. Addison-Wesley, 2005.

[20] Sun Microsystems, “Code conventions for the Java programming language,” http://java.sun.com/docs/codeconv, 1999.

[21] A. Vermeulen, S. W. Ambler, G. Bumgardner, E. Metz, T. Misfeldt, J. Shur, and P. Thompson, *The Elements of Java Style*. Cambridge University Press, 2000.

[22] K. Atkinson, “SCOWL readme,” http://wordlist.sourceforge.net/scowl-readme, 2004.

[23] J. C. Munson, *Software Engineering Measurement*. Auerbach, 2003.

[24] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.

[25] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, “Evaluating static analysis defect warnings on production software,” in *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM, 2007, pp. 1–8.

[26] S. McConnell, *Code Complete: A practical handbook of software construction*, 2nd ed. Microsoft Press, 2004.

[27] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, 2008, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[28] M. J. Crawley, *Statistics: an introduction using R*. John Wiley, 2005.