

Research

An Empirical Study of Rules for Well-Formed Identifiers

Dawn Lawrie^{1,**}, Henry Feild^{1†}, and David Binkley^{1‡}

¹ *Computer Science Department, Loyola College, 4501 N. Charles St., Baltimore, MD
21210-2699, USA.*

SUMMARY

Readers of programs have two main sources of domain information: identifier names and comments. In order to efficiently maintain source code, it is important for the identifier names (as well as comments) to communicate clearly the concepts that they represent. Deißeböck and Pizka recently introduced two rules for creating well-formed identifiers: one considers the consistency of identifiers and the other their conciseness. These rules require a mapping from identifiers to the concepts that they represent, which may be costly to develop after the initial release of a system.

*Correspondence to: Computer Science Department, Loyola College, 4501 N. Charles St., Baltimore, MD
21210-2699, USA.

*E-mail: lawrie@cs.loyola.edu

†E-mail: hfeild@cs.loyola.edu

‡E-mail: binkley@cs.loyola.edu

Contract/grant sponsor: National Science Foundation; contract/grant number: CCR-0305330



An approach for verifying that identifiers are well-formed without any additional information (e.g., a concept mapping) is developed. Using a pool of 48 million lines of code, experiments with the resulting syntactic rules for well-formed identifiers illustrate that violations of the syntactic pattern exist. Two case studies show that three-quarters of these violations are “real”. That is, they would be identified using a concept mapping. Three related studies show that programmers tend to use a rather limited vocabulary, that, contrary to many other aspects of system evolution, maintenance does not introduce additional rule violations, and that open and proprietary source differ in their percentage of violations.

KEY WORDS: Identifier Quality, Part-of-speech, Source Code Analysis

1. Introduction

Well-formed variable names, as described by Deißeböck and Pizka, can improve code quality [6]. The motivation for their work is the observation that “lousy naming in one place spoils comprehension in numerous other places,” while the basis for their work is found in the quote “research on the cognitive processes of language and text understanding shows that it is the semantics inherent to words that determine the comprehension process” [6]. Other studies have also pointed to the importance of good identifier names. For example, Rilling and Kelmola observe “In computer programs, identifiers represent defined concepts” [18], while Caprile and Tonella point out that “Identifier names are one of the most important sources of information about program entities” [5].



Deißenböck and Pizka define two characteristics of well-formed identifiers: conciseness and consistency. In order to verify that a variable is concise and consistent, a mapping from the domain of identifiers to the domain of concepts is required. Identifiers are termed consistent based on the concepts that they map to. Such a mapping must be constructed by a domain expert. For new projects, this mapping can be constructed alongside the program with minimal additional cost. For existing programs, however, the cost can be prohibitive. This paper considers *syntactically* concise and consistent naming, which requires no expert-constructed mapping. This work explores whether, by only considering the syntactic makeup of identifiers, a useful approximation to the techniques of Deißenböck and Pizka can be achieved.

Following Takang et al., Brook's theory of program comprehension underpins the theoretical framework of this work [20]. Brooks argues that programming involves the construction of mappings from a problem domain via intermediate domains into a programming domain – represented by program text. He contends further that the process of program comprehension is one of reconstructing knowledge about these domains and the relationships between them. The less ambiguity and the more precision in the program text, the easier this task. This process is aided by methodical identifier choices, such as those that follow the correctness and conciseness rules laid out by Deißenböck and Pizka [6].

Identifiers that fail to be concise or consistent increase comprehension complexity and its associated costs [6]. Such failures can be identified using Deißenböck and Pizka's techniques provided a mapping from identifiers to concepts is available. In the absence of such a mapping, it is still possible to identify a subset of these naming failures. A technique for doing so



introduced and empirically investigated in this paper. In more detail, the primary contributions of this paper are the following:

1. Syntax-based Conciseness and Consistency First and foremost, the paper lays out a definition for syntax-based concise and consistent identifier naming. Being syntax-based, the technique does not require an expert-constructed mapping from identifiers to concepts.

2. Verification The verification has two parts. First, it shows the magnitude of syntax-based conciseness and consistency failures in production code. It then considers the correctness of the syntactic definition through two case studies.

3. Statistical Models Statistical models are used to better understand the collected data. These include testing the *Vocabulary Hypothesis* – that programmers use a rather limited vocabulary, the *Longitudinal Hypothesis* – that evolution introduce conciseness and consistency failures, and finally the *Programming Model Hypothesis* – that open and proprietary source differ in their violations.

The verification study first considers if syntactic violations occur in practice. If not, the technique is of little value. Assuming violations do exist, the second part of this study considers if the violations found are the same as those found using Deißeböck and Pizka’s concept-based approach. Finally, an investigation of the three formal hypotheses is used to better understand the sources of violations.

The rest of this paper first presents some necessary background material in Section 2. Definitions of syntax-based conciseness and consistency are given in Section 3, followed by



the layout of the experimental design in Section 4. The empirical investigation of the syntactic definition, using almost 50 million lines of code, is presented in Sections 5 and 6. Related work is then considered in Section 7. Finally, the paper concludes with a discussion of some topics for future investigation and a summary in Sections 8 and 9.

2. Background

This section provides context for the technique described in Section 3 and the empirical studies of Sections 5 and 6. It first describes, in more detail, Deißböck and Pizka rules for well-formed identifiers. This is followed by a description of identifier composition: the constituent parts of an identifier used in the remaining sections.

Well Formed Identifiers

Deißböck and Pizka describe a formal model for well-formed identifier naming that includes rules for the *consistent* and *concise* naming of identifiers [6]. Their rules make use of the set of all concepts relevant to a program and provide “a formal model based on bi-jjective mappings between concepts and names.”

They define one rule for *concise* identifier names and two rules for *consistent* identifier names. An identifier i for concept c is *concise* provided its semantics exactly match the semantics of the concept it is used to represent. For example, `output_file_name` concisely represents the concept of the name of an output file. (A related notion, *correctness* allows an identifier to represent a more general concept. For example, `file_name` correctly, but not concisely, represents the



concept of the name of an output file, while the identifier `foo` neither correctly nor concisely represents the concept.)

There are two rules related to *consistent* identifiers. They identify inconsistencies caused by identifier *homonyms* and identifier *synonyms*. In natural language, a homonym is one of two or more words pronounced and perhaps spelled alike but different in meaning (*e.g.*, ‘waste’ and ‘waist’) [22]. A *synonym* is one of two or more words or expressions of the same language that have the same or nearly the same meaning in some or all senses (*e.g.*, ‘baby’ and ‘infant’) [22].

In a program, an identifier *i* is a *homonym* if it represents more than one concept from the program (*e.g.*, the identifier `file` in Figure 1a). As Deißböck and Pizka emphasize, accurately knowing the set of all concepts used in a program is important. The identifier `path` is not a homonym in a program with only one path concept. Thus, it is important that only concepts from the program be considered. Otherwise, the concept space becomes too large and unwanted inconsistencies arise.

The second inconsistency involves *synonyms*: two identifiers *i1* and *i2* are *synonyms* if the concepts associated with *i1* have a non-empty overlap with the concepts associated with *i2* (*e.g.*, the identifiers `file` and `file_name` share the concept *file name* in Figure 1b). As a second example, the identifiers `list_head` and `list_front` are also synonyms (`head` and `front` are natural language synonyms in this context).

Using these definitions, Deißböck and Pizka present a case study in which maintenance introduces the seven identifiers `pos`, `apos`, `abspos`, `relpos`, `absolute_position`, `relative_position`, and `position` representing two concepts $c_1 = \text{“absolute position”}$ and $c_2 = \text{“relative position”}$. The identifier `position` would correctly but not concisely represent the concept *absolute_position*

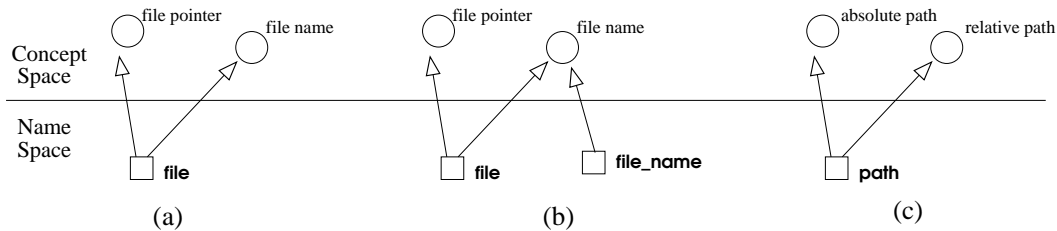


Figure 1. Illustration of the two types of syntactic violation. Figure (a) shows a homonym violation. Figure (b) shows how a synonym violation is also introduced by the function that opens a file. Finally, Figure (c) shows a plausible homonym only example.

provided that the program did not include any other position concepts (*e.g.*, *relative position*). As the program includes multiple specific kinds of positions, the identifier `position` fails the correctness and conciseness requirements. In addition, the identifier `position` also fails the homonym consistency requirement as it is associated with more than one concept from the program (in this case concepts c_1 and c_2). Finally, the study determined that `relpos` and `relative_position` are both used for concept c_2 , which violates the synonym consistency requirement.

In most instances, when the homonym requirement is violated the synonym requirement is also violated. Figure 1 illustrates this. The identifier `file` is a homonym associated, in different parts of the program, with the concept of a *file name* and elsewhere a *file pointer*. If the two concepts are to be referred to in the same scope (at least in a strongly typed language) then at least one additional identifier would be required as shown in Figure 1b. However, the inclusion of this second identifier introduces a synonym violation as the identifiers `file` and `file_name` both refer to the same concept.



In this example any function that opens a file would need to refer to both the *file name* and *file pointer* concepts. As an example in which it is plausible that a homonym would exist in the absence of a synonym, consider the situation shown in Figure 1c in the context of a program that reads a directory path into the variable `path` and then passes it to either function *f1* or *f2* depending on whether the path is relative or absolute. If *f1* and *f2* use the name `path` for their formal parameter, then the program includes two concepts *relative path* and *absolute path* and uses only a single identifier, `path`, to refer to them. This violates the homonym rule, but not the synonym rule.

Identifier Composition

Following others who study identifiers' makeup [2, 6, 5, 7, 18], identifiers are assumed to be composed of parts herein, referred to as “*words*” – sequences of characters with which some meaning may be associated. Two kinds of words are considered: *hard words* and *soft words*. Hard words are separated by the use of word markers (*e.g.*, the use of CamelCase or underscores) [2]. For example, the identifiers `sponge_bob` and `spongeBob` both contain the well separated hard words `sponge` and `bob`.

For many identifiers, the division into hard words is sufficient. This occurs when all hard words are dictionary words or known abbreviations. When a hard word is in neither category, the identifier may contain non-well-separated words. The division of a hard word into one or more “*soft words*” is the goal of identifier splitting [7]. The algorithm used employs a greedy approach that recursively finds the longest prefix and suffix that are in the dictionary or a known abbreviation list [7]. Take, as an example, the splitting of the identifier `hashtable_entry`.



This identifier consists of one division marker (an underscore) and, thus, two hard words, `hashtable` and `entry`. The hard word `hashtable` is composed of two soft words – `hash` and `table`, while the hard word `entry` is composed of a single soft word.

3. Syntax-based Conciseness and Consistency

This section introduces and formalizes syntactic rules for well formed identifiers. The absence of a concept mapping precludes the discovery of identifiers that violate the homonym restriction only. Testing that identifiers satisfy a restricted form of synonym consistency and conciseness can be achieved *syntactically* (*i.e.*, without the identifier to concept mapping). In essence, if an identifier is *contained* within another identifier, there is a violation of the syntactic synonym consistency requirement or the syntactic conciseness requirement or both. Here containment results when one identifier includes, in the same order, all the soft words from another. For example, the identifier `relative_position` includes two hard words each composed of a single soft word. Thus it includes, in order, all the soft words from the identifier `position`.

There are two possible violations that can occur when containment is detected between a pair of identifiers. One possibility is that there is a single concept associated with the two identifiers; thus, violating Deißeböck and Pizka's synonym consistency requirement. For example, as shown in Figure 2a, if the program contains only the concept *relative position* and there are the same two identifiers, `position` and `relative_position`, a synonym consistency violation exists. This is not a conciseness problem because `position` adequately describes the concept *relative position* when it is the only type of position used in the program. The other possibility is that each identifier refers to a different concept in the concept mapping; therefore,

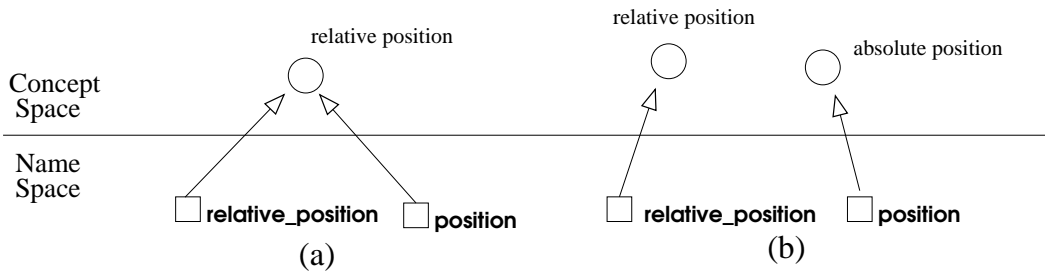


Figure 2. Shows the Type I violations. Figure (a) shows a Synonym Consistency Violation. Figure (b) shows a Conciseness Violation.

the contained identifier violates Deißeböck and Pizka's conciseness requirement. As shown in Figure 2b, consider the two identifiers `position` and `relative_position`. The identifier `position` maps to the concept *absolute position*, and `relative_position` maps to the concept *relative position*. The identifier `position` violates the conciseness requirement by being too general a name for the concept *absolute position*. When the containment pattern is detected involving a pair of identifiers, the violation will be referred to as a Type I violation, which is formalized as follows:

Definition 1 (Type I Violation) *Let identifier id_1 be the sequence of soft words $sw_1 sw_2 \dots sw_{n_1}$. Identifiers id_1 and id_2 fail either the syntactic synonym consistency requirement or the syntactic conciseness requirement (a Type I violation) if id_2 includes the sequence of soft words $w_1 w_2 \dots sw_1 sw_2 \dots sw_{n_1} \dots w_{n_2}$ (i.e., $id_2 = w_1 w_2 \dots id_1 \dots w_{n_2}$).*

The presence of a second containing identifier (e.g., `absolute_position`, which also contains `position`) implies that Deißeböck and Pizka's rule for conciseness has been violated, and there is a possible synonym consistency violation as well. It does so because the two containing identifiers imply the program includes at least two separate concepts, but the contained

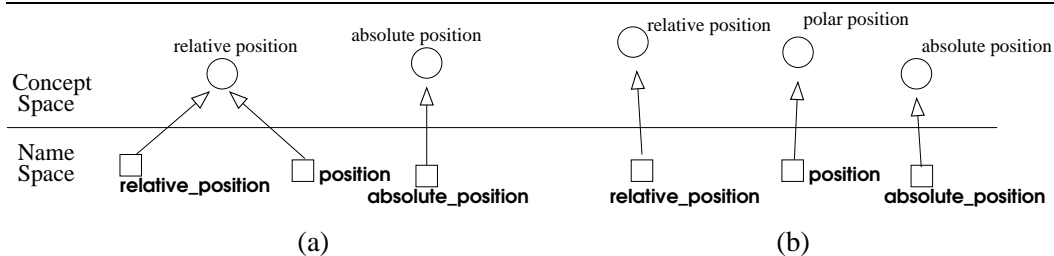


Figure 3. Shows the Type II violations. Figure (a) shows a Conciseness and Synonym Consistency Violation. Figure (b) shows a Conciseness Violation Only.

identifier does not precisely indicate to which of the two it refers; therefore, a conciseness violation must exist.

Figure 3 illustrates the two scenarios. In both cases, there is conciseness violation because at least two concepts exist and the contained identifier (`position` in both cases) could represent either concept. In the first scenario, shown in Figure 3a, a synonym violation also exists. In this case there are only two concepts but three identifiers, so `position` must be a synonym of at least one of the two. As shown in Figure 3b, it is possible to violate only the conciseness rule. Here, each identifier refers to a unique concept; however, given the presence of `absolute_position` and `relative_position`, `position` does not concisely describe *polar position*. When a containee is contained in more than one container, it is referred to as a Type II violation, which is formalized as follows:

Definition 2 (Type II Violation) *Let identifier id_1 be the sequence of soft words $sw_1 sw_2 \dots sw_{n1}$. Identifiers id_1 , id_2 , and id_3 fail the syntactic conciseness requirement and may fail the syntactic synonym consistency requirement (a Type II violation) if id_2 includes*



the sequence of soft words $w_1 w_2 \cdots sw_1 sw_2 \cdots sw_{n1} \cdots w_{n2}$ and id_3 includes the sequence of soft words $u_1 u_2 \cdots sw_1 sw_2 \cdots sw_{n1} \cdots u_{n3}$.

For completeness, Definition 2 should address one technical point that it omits for simplicity. The complete definition requires that id_2 not contain id_3 or vice-versa. In practice, the simple definition is correct for 99.64% of the 2,699,289 identifiers studied. Thus, the additional complexity of requiring non-containment is ignored.

4. Experimental Design

This section describes the experiments design. It first lays out the motivation for the study. It then provides information on the subject programs studied and the statistical tests used to study them. Finally, the section considers threats to validity of the empirical studies.

Motivation

The study presented in Section 5 empirically investigates two important questions related to Deißeböck and Pizka's definition for well-formed identifiers. First, do syntactic synonym consistency and conciseness failures exist in real code? Obviously, the technique is of little interest if violations are infrequent or non-existent. Second, are syntactic violations indicative of the violations found using the Deißeböck and Pizka concept-map-based definitions? If the syntactic approach can identify a useful subset of the violations, without the need for a concept mapping, then it forms the core of a useful tool.

The study also considers, in Section 6, three hypotheses aimed at understanding the origin



of violations of the rules for well-formed identifiers. The first hypothesis, the *Vocabulary Hypothesis*, asserts that programmers use a limited vocabulary. The second hypothesis, the *Longitudinal Hypothesis*, investigates if maintenance and evolution introduce violations. Finally, the *Programming Model Hypothesis* considers the difference in violations found in open and proprietary source.

Subject Programs

The analysis includes empirical data collected from 186 programs, some of which are different versions of the same program. Up to 70 versions of a program were considered to support the longitudinal study. Ignoring multiple versions, 78 unique programs were considered. All but 12 were open source programs. There exist, however, sufficient proprietary code to obtain statistically significant results. Programs ranged in size from 1,423 to 3,087,545 LoC and covered a range of application domains (*e.g.*, aerospace, accounting, operating systems, program environments, movie editing, games, etc.) and styles (command line, GUI, real-time, embedded, etc.). Most of the code was written in C. Significant C++ and Java code were also studied along with a small amount of 30 year old Fortran code. Several of the programs were written by programmers whose native language was not English. For these programs the analysis was performed using a dictionary of the programmer's native language or, if multiple languages were evident in the code, the union of the respective dictionaries. (The publicly available dictionaries that accompany the Linux spell checker *ispell* version 3.1.20 were used.)

Table I shows 10 representative C, C++, and Java subject programs. The two Fortran programs are not shown in the table. They are PLM compilers from 1975 and 1981 and



Table I. Subject Programs (proprietary programs are named I#).

program	wc				sloc Total	year	
	C	C++	Java	Total		start	release
cinelerra-2.0	1,044,996	106,357	0	1,151,353	820,980	1996	2004
cpm68k1-v1.2a	132,171	0	0	132,171	102,252	1978	1984
empire_server	85,548	0	0	85,548	62,793	1985	1998
eMule0.46c	1,759	172,164	0	173,923	135,567	1999	2005
I4.2	2,109,050	398,463	502,965	3,010,478	1,704,823	1993	2004
jakarta-tomcat-5.5	68,003	0	353,604	421,607	219,766	1999	2005
LEDA-3.0	41,610	0	0	41,610	27,425	1988	1992
minux-2.0	326,210	0	0	326,210	244,033	1980	1996
mozilla-1.4	1,047,741	1,949,292	6,493	3,003,526	2,107,436	1998	2003
quake3-1.32b	353,806	57,431	0	411,237	281,432	1999	2005
Totals for all code not just that shown above							
open source	19,170,546	14,587,482	6,327,380	40,106,590	27,129,263		
proprietary source	7,167,689	787,094	582,107	8,536,890	5,391,815		
all	26,338,235	15,374,576	6,909,487	48,643,480	32,521,078		

include 9,704 and 11,478 LoC, respectively. The table reports code sizes for C, C++, and Java (and their sum) as counted by the Unix utility `wc` (excluding header files). In addition, the total number of non-comment non-blank lines of code, as reported by `sloc` [23], is shown. The average percentage of non-comment non-blank lines varies by language with 66% of the C code, 72% of the C++ code, and 58% of the Java code being non-comment non-blank lines. The last two columns present the start year of the project and its release year. These dates were extracted from program documentation (internal and external). In general, the release year is more accurate as it can be difficult to determine the start year for a program that includes third party libraries written before the program “started”.



Table II. Basic counts from 14 selected programs. Some of the programs from Table I are repeated for comparison, other's were selected to provide diversity in the presented data. [†]Percent increase is the percent increase from hard words to soft words.

program	dominant language	start year	release year	LoC (wc)	unique ids	id instances	hard words	soft words	percent [†] increase
cinelerra-2.0	C	1996	2004	1,151,353	84,612	1,833,424	209,059	261,793	25.2%
cpm68k1-v1.1	C	1974	1983	73,172	4,167	79,660	4,560	8,193	79.7%
eclipse-3.2m4	Java	2001	2005	3,087,545	167,662	3,893,272	554,068	612,632	10.6%
gcc-2.95	C	1987	1999	841,633	44,941	897,728	110,060	146,474	33.1%
I1	C	1987	1997	454,609	30,092	482,228	48,125	82,307	71.0%
I4.2	C	1993	2004	3,010,478	113,662	2,694,901	328,079	422,364	28.7%
I6.6	C	2000	2002	237,257	10,791	104,290	29,207	34,549	18.3%
jakarta-tomcat-5.5.11	Java	1999	2005	421,607	19,202	351,487	48,537	54,471	12.2%
mozilla-1.6	C++	1998	2004	2,919,307	189,916	3,649,329	563,448	659,396	17.0%
mysql-5.0.17	C++	1996	2005	1,293,270	50,383	1,023,362	132,249	163,363	23.5%
plm80s	Fortran	1975	1977	9,704	581	22,314	581	886	52.5%
quake3-1.32b	C	1999	2005	411,237	31,114	542,664	75,474	94,144	24.7%
sendmail-8.7.5	C	1983	1996	78,757	2,877	62,075	4,492	6,828	52.0%
spice3f4	C	1985	1993	298,734	12,388	452,423	24,599	34,882	41.8%

Table II summarizes statistics regarding the identifiers along with some demographic information (*e.g.*, dominant programming language, and the start and release years of the program). The table presents a representative sample of the programs. Table III summarizes the data over all programs (not just that of the representative programs from Table II). Summaries include two orthogonal groupings (open source versus proprietary, and by programming language) and all the data taken collectively.



Table III. Total counts from all programs.

Totals for (over all code not just that shown)	instances per id	hard words per id	soft words per id	LoC (wc)	unique ids	id instances	hard words	soft words	percent increase
open source	19.2	2.7	3.2	40,106,590	2,504,937	48,098,029	6,817,779	8,040,625	17.9%
proprietary	19.6	2.7	3.5	8,536,890	385,792	7,543,663	1,055,329	1,331,327	26.2%
C	18.6	2.5	3.1	26,338,235	1,566,289	2,9074,119	3,956,372	4,821,045	21.9%
C++	19.3	2.9	3.5	15,375,576	965,402	18,836,801	2,835,896	3,341,987	17.8%
Java	22.1	3.0	3.4	6,909,487	356,225	7,885,428	1,076,709	1,203,537	11.8%
Fortran	18.0	1.4	1.8	21,182	2,238	40,273	3,141	3,993	27.1%
All	19.3	2.7	3.2	48,643,480	2,890,153	55,638,621	7,872,119	9,370,562	19.0%

Statistical Tests

Several statistical techniques are used in the interpretation of the data gathered during the study. This section introduces these techniques. First, the Mann-Whitney test provides a non-parametric comparison that determines whether two samples come from the same distribution. Because the test is non-parametric, the underlying distribution need not be normally distributed.

When a simple linear correlation between quantitative variables is of interest, Pearson's linear regression is used. To measure the effect of explanatory variables X , Y , and Z on response variable A , the resulting model coefficients, m_i , belong to the linear equation



$$A = m_1X + m_2Y + m_3Z + b.$$

Each coefficient has an associated p -value. A p -value less than 0.05 indicates a significant explanatory variable. These models assume the absence of interactions between the explanatory variables; thus, they are often used to provide an initial impressions of the data or to determine the influence of a particular explanatory variable.

For more complex models, linear mixed-effects regression models are used to analyze the data [21]. Such models allow the examination of important effects that are associated with the response variables. The initial model includes explanatory variables and a number of interaction terms. The interaction terms allow the effects of one variable on the response variable to change depending upon the value of another variable. Backward elimination of statistically non-significant terms ($p > 0.05$) yields the final model. Note that some non-significant variables are retained to preserve a hierarchical well-formulated model [17]. This occurs when a non-significant variable is involved in a significant interaction.

In these models, Tukey's highly significant difference method for multiple comparisons is used. However, computing a standard t -value for each comparison and then using the standard critical value increases the overall probability of a Type I error. Thus, Bonferroni's correction is used when computing p -values to account for multiple comparisons. In essence each p -value is multiplied by the number of comparisons, and the adjusted p -value is compared to the standard significance level (0.05) to determine significance. Tukey's method and Bonferroni's correction were chosen because they are both rather conservative tests.



With both Pearson's test and the linear mixed-effects regression models, the coefficient of determination, R^2 , is reported. This coefficient is interpreted as the proportion of the variability in the response variable that is explained by the selected explanatory variables. This coefficient ranges from 0 to 1; the closer to 1, the better the model.

Threats to Validity

With any empirical study it is important to consider threats to validity (*i.e.*, the degree to which the experiment measures what it claims to measure). There are four types of validity to consider: external validity, internal validity, construct validity, and statistical conclusion validity [3].

External validity, sometimes referred to as selection validity, is the degree to which the findings can be generalized to other organizations or settings. In this experiment, selection bias is possible in the programs studied; thus, for example, perhaps Java programs in general exhibit different behavior than observed in the study. The programs were selected based on availability (a convenience sample). In addition, results and trends from the experiment may not apply to other programming languages, time periods, or natural languages. The selection of programming language includes those of interest to the authors. The period and natural language again reflect availability. By considering a large volume of code over a large time frame, and the variety of programming styles and environments (*e.g.*, real-time systems, embedded systems, event-driven systems, open source, and proprietary programs), there is high confidence that similar findings would occur for "closely related parameters". The "farther" away (*e.g.*, 1960 or 1950 codes) one gets the lower this confidence.



Second is the threat to internal validity: the degree to which conclusions can be drawn about the causal effect of the explanatory variable on the response variable. Since soft words can be accurately compared for containment, the only serious threat to internal validity comes from potential errors in the extraction tools. Other potential threats to internal validity, for example, history effects, attention effects, and subject maturation [3] are non-issues in this study given the absence of human subjects. To mitigate the one existing threat to internal validity, mature tools were used where possible and newly written tools were extensively tested. This reduces the impact that implementation faults may have on the conclusions reached.

Construct validity assesses the degree to which the variables used in the study accurately measure the concepts they purport to measure. The variables used in the study can be measured with high accuracy. Thus, threats to construct validity are not expected to be a serious concern. Note that if human judgments on, for example, quality were included in the correlations then construct validity would become a more serious concern.

Finally, a threat to statistical conclusion validity arises when inappropriate statistical tests are used or when violations of statistical assumptions occur. The statistical tests used were chosen based on past experiments and with guidance of those trained in statistics. Finally, Tukey's highly significant difference test with Bonferroni's correction is a very conservative test. These factors serve to reduce the possibility of an inappropriate statistical conclusions being drawn.



5. Verification

This section begins by empirically investigating the rules for well-formed identifiers introduced in Section 3 using the identifiers extracted from 186 programs comprising just under 50 million lines of code. It presents two studies. The first considers if syntactic violations occur in practice. If not, the technique is of little value. It turns out that violations do exist; thus, the second study considers two case studies aimed at understanding if they are the same as those found using Deißeböck and Pizka's concept-based approach.

Existence of Type I and Type II violations

The ability to identify Type I and Type II failures is of little value if the pattern does not occur in practice. Table IV shows the percentage of failures for 42 representative programs along with the number of unique identifiers in each program and the percentage of *severe* failures in which the contained identifier includes at least three soft words. Figure 4 shows the distribution of these failures for all 186 programs. Based on the last row in Table IV, an average program includes just over 2900 identifiers that exhibit a Type I failure and just over 1300 that exhibit a Type II failure. This indicates that sufficient violations exist in practice to warrant further study.

Two Case Studies

The second empirical investigation includes two case studies. The first exhaustively considers all conciseness and consistency failures from three small programs. This study compares the tool's output to that produced by a human "oracle". The second case study considers a



Table IV. Percent of Type I and Type II failures for selected programs. (A “*” marks programs with a minimum or a maximum value. Proprietary programs are named I#.)

Program	Unique Identifiers	Failures		Severe Failures	
		Type I	Type II	Type I	Type II
LEDA-2.1.1	2226	21%	10%	2%	2%
LEDA-3.1.2	2946	20%	8%	1%	1%
a2ps-4.12	3593	22%	10%	3%	2%
apache_1.3.29	8040	19%	8%	4%	3%
barcode-0.98	344	21%	6%	5%	1%
*byacc.1.9	507	20%	9%	1%	0%
cinelerra-2.0	71995	21%	9%	6%	4%
*compress	164	11%	5%	0%	0%
cpm68k1-v1.3	2417	12%	6%	0%	0%
cvs-1.11.1p1	5552	20%	9%	3%	2%
eMule0.46c	21372	17%	8%	6%	4%
eclipse-2.1	83207	26%	11%	10%	6%
eclipse-3.2m4	155932	25%	9%	11%	4%
*genesis-all-3.0	2110	33%	12%	13%	5%
ghostscript-7.07	26546	19%	9%	6%	4%
gnuchess-4.0	1198	16%	8%	1%	1%
*gnugo-1.2	114	15%	8%	0%	0%
gnugo-2.0	627	15%	6%	1%	1%
gnugo-3.0.0	3118	21%	9%	3%	2%
httpd-2.0.48	16975	19%	9%	5%	3%
I1	29619	17%	12%	5%	4%
*I4.1	92547	21%	11%	10%	7%
I4.2	110727	21%	11%	10%	7%
I6.1	9869	16%	8%	5%	4%
I6.6	10583	16%	8%	5%	4%
I9	41189	20%	9%	7%	5%
I12	1098	19%	11%	4%	3%
jakarta-tomcat-3.0	3920	24%	9%	5%	3%
jakarta-tomcat-5.5	18416	25%	10%	7%	4%
javabb_073	1716	27%	9%	5%	2%
linux-2.0	21076	15%	7%	1%	1%
mozilla-1.0	173124	22%	9%	8%	5%
mozilla-1.6	176318	22%	9%	8%	5%
mysql-5.0.17	46297	21%	9%	7%	4%
*pacifi3d0.3	1139	11%	5%	1%	1%
*plm80s	539	8%	4%	0%	0%
quake3-1.32b	28676	18%	8%	5%	3%
samba-3.0.0	22553	20%	9%	8%	4%
spice3f4	9845	18%	10%	5%	4%
*tile-forth-2.1	661	34%	22%	2%	2%
uupc	147	12%	7%	1%	1%
Min	114	8%	4%	0%	0%
Max	181032	34%	22%	13%	7%
Average	14512	20%	9%	4%	3%

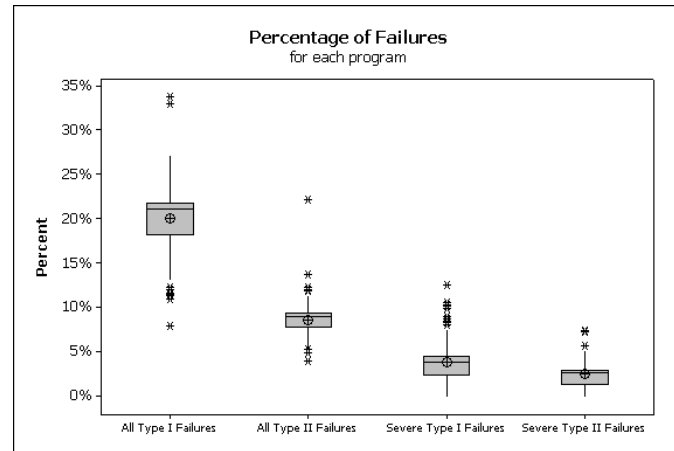


Figure 4. Summary of Type I and Type II failures for all programs.

sampling of the conciseness and consistency failures from the larger program eMule, a 170 KLoC C++ program.

Given that a significant number of syntactic synonym and conciseness violations occur, this section considers the next question to address: “are these violations real?” There are two possible differences between the violations that the syntactic approach reports and those obtained using a concept mapping. Clearly the syntactic approach will miss violations when the identifiers do not share common soft words. For example, the identifiers *file*, *fp*, and *fin* might be synonyms (all representing the *input file pointer* concept), but the syntactic approach cannot at present determine this. The other difference involves identifiers for which the syntactic approach identifies a violation, but no violation exists when using the associated concepts.

To determine how many such false positives the syntactic approach produces, this section first presents results from an *exhaustive* case study of three of the smaller programs and then



a *sampling* case study of the larger program eMule. In the first case study, hand inspection of all violations from three of the smaller programs was performed (two C programs, `which.c` and `uucp.c`, and one Java program `VE.java`). As shown in Table V, this inspection produced six categories. The encouraging news is that, varying with language, 51% to 72% of the Type I violations, and 62% to 76% of the Type II violations were true violations. For example, the identifiers `status` and `file_status` violate the synonym consistency requirement while the identifiers `home_dir` and `in_home` indicate two refinements of the concept *home*, which means the identifier `home` fails the syntactic conciseness requirement.

As is shown Table V, in both the C programs and the Java program, about 10% of the Type I and Type II violations are false positives caused by the lack of a concept model. For example, `prefix` is contained in `isolate_tilde_prefix`. While `prefix` could be replaced with `string_prefix` in the string concatenation routine where it is found, `isolate_tilde_prefix` is a function whose associated concept does not overlap with that of the variable `prefix`. A conciseness example from `uucp`, includes the identifiers `FILE`, `copy_file`, `file_mode`, and `log_file`. As `FILE` is a type, its concept is separate from the others, although the syntactic algorithm cannot, at present, make this determination. Finally, the program which includes the identifier `saw_slash`, which is contained in `saw_slash_dot` and `saw_slash_dot_dot`. However, the concepts for these three are distinct; thus no violation of either rule is present.

The remaining four categories all suggest refinements to the technique. The third category includes a structure field such as `adr`, which overlaps with the local variable `next_adr`. Deißböck and Pizka do not explicitly discuss structure fields, but including the structure name (`letter` in this case), seems a straight forward extension of their work that removes the



Table V. Hand inspection of Type I and Type II violations for three programs to determine false positives.

Description	Type I		Type II	
C Programs which.c and uucp.c				
(1) clear violation	49	72%	22	76%
(2) non violations	6	9%	3	10%
(3) struct fields	1	1%	0	0%
(4) attribute	9	13%	1	3%
(5) verb-noun phrase	3	4%	3	10%
Total	68	100%	29	100%
Java Program VE.java				
(1) clear violation	95	51%	57	62%
(2) non violations	19	10%	8	9%
(3) struct fields	19	10%	13	14%
(4) attribute	15	8%	5	5%
(5) verb-noun phrase	28	15%	7	8%
(6) a_type	9	5%	2	2%
Total	185	100%	92	100%
All 3 Programs				
(1) clear violation	144	57%	79	69%
(2) non violations	25	10%	11	9%
(3) struct field	20	8%	13	11%
(4) attribute	24	9%	6	5%
(5) verb-noun phrase	31	12%	10	8%
(6) a_type	9	4%	2	2%
Total	253	100%	121	100%



synonym failure in this example. The higher percentage in the Java program comes from the use of object-oriented programming techniques. In particular, many failures are a result of similar messages names sent to objects of different static types. Often these identifiers can be disambiguated based on the static type of the receiver.

The fourth category includes what Ada refers to as attributes and C# as properties. For example, the two identifiers `cwd` and `cwd_len` (a Type I failure), and the three identifiers `result`, `result_index`, and `result_size` (a Type II failure) include *variable properties*. Here, by convention, programmers recognize identifiers such as `cwd` and `result` as the underlying value of which the other identifiers are properties. This kind of identification may be automateable using grammar-based techniques such as those used by Tonella et al. [5].

The fifth category considers incorporating part-of-speech information to determine identifier patterns. An example includes the identifiers `home_dir` and `get_home_dir`, which present a Type I violation; however, they are associated with different concepts and `home_dir` is a concise name for its underlying concept. Using part-of-speech information, this case can be identified since the two identifiers differ by a verb. This pattern is explored further in Section 8.

The final category was found only in the Java program, but appears to be stylistic. Violations consisted of a type, such as `JTable`, and a variable prefixing the name of the type such as “a”, yielding `aJTable`. This pattern would be common in Smalltalk code, given its naming convention for formal parameters. In one respect the names starting with a do not represent good choices. However, the programmer used these in a very controlled way as part of test stubs.

In summary, if the last four categories can be automatically identified (only the attribute category presents any real challenge), then the violations in these categories can be removed.



This leaves only the first two categories: clear violations and non-violations. Encouragingly, post removal, between 85% and 90% of the syntactic violations are real violations.

The first case study considered all the violations from three small programs. The next case study is a “sampling case study” of eMule, a 170 KLoC C++ program chosen at random from the larger programs. Exhaustively examining eMule’s 3725 Type I failures and 1762 Type II failures is prohibitively expensive. Instead seven informative examples were selected. They represent the different types of violations encountered in the program as a whole. Each example includes three parts: the base (contained) identifier, the identifiers that contain it, and a discussion.

- (1) `m_strHost` (the contained identifier)
`m_strHostName`

The first case is the classic Type I violation in which a concept that already has a name receives another. In this case, the identifier `m_strHost` and the identifier `m_strHostName` both refer to the same concept (the string representation of the host computer to connect to), so this is a synonym consistency violation.

- (2) `CheckDiskSpace`
`CheckDiskSpaceTimed`

As a second classic Type I example, eMule includes two methods for checking if sufficient disk space exists to write a file. The identifiers fail to satisfy Deißeböck and Pizka’s definition of conciseness since `CheckDiskSpace` actually refers to the concept *untimed disk space check*. In this instance, one obvious fix would be to rename the first method `CheckDiskSpaceUntimed` or something similar. This would disambiguate the names for



the two concepts of timed and un-timed disk space checks and would eliminate the conciseness violation.

(3) `lcmpCloseHandle`
`lpfnlcmpCloseHandle`

The third example illustrates a case in which Type I violation occurs, but knowing a little about the identifiers removes any real issue. EMule includes the class type `lcmpCloseHandle` and the variable `lpfnlcmpCloseHandle` of that type. Both identifiers represent the same concept, but knowing that one is a type name disambiguates the two.

(4) `m_n_file`
`m_n_file_size`

The identifiers `m_n_file` and `m_n_file_size` form a less egregious Type I violation because `m_n_file` occurs as a local variable in a separate scope from `m_n_file_size`. The method “`int CZIPFile::GetCount() { return m_nFile; }`” suggests that consistency could be attained by renaming `m_nFile` to `m_nFileCount`. This violation at first appears to fall into the “attribute” category, but it does not. It is a true violation.



-
- (5) `m_wndSplitter`
 `m_wndSplitterchat`
 `m_wndSplitterirc`
 `m_wndSplitterstat`
 ...

The eMule class `CSplitterControl` implements a window splitter control. The server window includes a window splitter, under the name `m_wndSplitter`, as do several other windows. For example, the “chat” window includes `m_wndSplitterchat` which, like `m_wndSplitter` is of type `CSplitterControl`. (Note that this identifier is not well separated and thus identifier splitting into soft words is required to uncover this conciseness failure.) It is hard to tell if the program’s evolution began with a single splitter (in the server class) and the others were subsequently introduced or not, but in order to have concise names, `m_wndSplitter` should be renamed `m_wndSplitterServer`, which removes the Type II violation.

- (6) `GetFileType`
 `GetFileTypeDisplayStr`
 `GetFileTypeByName`
 `GetFileTypeSystemImageldx`
 `GetFileTypeDisplayStrFromED2KFileType`

The penultimate example involves five identifiers, which indicates a Type II violation. First, the identifier `GetFileType` needs to be renamed so that it no longer fails the conciseness requirement. One naive way of doing so, which focuses on separating it from the second identifier, is to replace the first identifier with `GetFileTypeNonDisplayStr` or `GetFileTypeInternalStr`.

Although the naive approach would remove the Type II violation, it does not address the root problem. All five of the identifiers need to be distinguished from one another



```
// Return file type as used internally by eMule,  
// examining the extension of the given filename  
CString GetFileTypeByName(LPCTSTR pszFileName)  
{  
    ...  
}
```

(a)

```
// Returns a file type which is used eMule internally only (GUI)  
CString GetFileTypeDisplayStrFromED2KFileType(LPCTSTR pszED2KFileType)  
{  
    ...  
}
```

(b)

Figure 5. Code snippets for the Type II case study.

so that the concept of each is clear. A snippet showing the definition of the third identifier appears in Figure 5a. As is clear from the comments preceding the definition, to achieve conciseness, the third identifier should be replaced with something like `GetFileTypeInternalByName`. Similarly, to achieve conciseness, with the fourth identifier, the first would need to be separated from the concept of an “image index”.

Finally, part of the definition of the fifth identifier is shown in Figure 5b. Here the comment preceding the definition confuses the situation as the method produces an internal file type, but unlike `GetFileTypeByName`, this one appears to be appropriate for the GUI. This implies that internal file type names can be suitable for the GUI or not. This is something the names of the two methods fail to make clear. For example, the identifier `GetFileTypeByName` should bear more in common with



`GetFileTypeDisplayStrFromED2KFileType` as it too returns an internal type name. As with the others, this identifier also conflicts with the first. The name for `GetFileType` would need to take all these concepts into account. In addition, `GetFileTypeDisplayStr` introduces a Type I violation, which also needs to be addressed. To the extent that this example seems confusing, it is an excellent indication of the value of concise and consistent identifiers, as they would have had helped make clear the various concepts related to type names.

(7) `ident`

```
IPHeader.ident (a field)
m_bLogSecureIdent
m_htiLogSecureIdent
...
```

The final example is really a non-example. The identifier `ident`, which is contained in 37 other identifiers, exists in two separate contexts. First, it is a local variable of the method `CircMain::Connect()`. As there is no real conflict with the associated concepts for this local variable, it suggests that scope information might play a role in helping an engineer determine when a violation might be a false positive. The second use of `ident` is as a field of the structure `IPHeader`. One might view its *full* name as `IPHeader.ident` which would be a more concise name. Deißeböck and Pizka do not discuss using context provided by a scope or a type (class or structure), but it seems a straightforward improvement.



6. Statistical Models

This section presents three statistical models that consider potential causes of rule violations. The first considers an observation by Antoniol et al., that programmers tend to use a rather limited vocabulary [2, 4]. The second presents a longitudinal study to address the question “does evolution introduce conciseness and consistency failures?” The final statistical model considers a comparison of open and proprietary source.

Limited Programmer Vocabulary

The first of the three hypotheses addresses an observation made by several authors (*e.g.*, Antoniol et al. [2] and Caprile and Tonella [4]) that “Programmers tend to process application-domain knowledge in a consistent way when writing code: program item names of different code regions related to a given text document are likely to be, if not the same, at least very similar” [2]. Thus, programmers tend to use a rather limited vocabulary.

To formally investigate this observation, the consistency and conciseness failures in all programs were recomputed after factoring in natural language synonyms. This is done using WordNet, a powerful tool for processing natural language [8]. In particular, WordNet’s synonym sets are used to look for violations that involve natural language synonyms. For example, doing so allows the tool to correctly determine that the identifiers `last_opt` and `end_opt` are synonyms as `last` and `end` are natural language synonyms in English. Formally, Definitions 1 and 2 were extended as follows: assume that for soft word w , $S(w)$ denotes the natural language synonyms of w . In the definition for Type I and Type II violations replace



$w_1 w_2 \cdots sw_1 sw_2 \cdots sw_n \cdots w_m$ with $w_1 w_2 \cdots s_1 s_2 \cdots s_n \cdots w_m$, where $s_i \in S(sw_i)$ with the corresponding replacement performed on id_3 .

Vocabulary Hypothesis

H_0 : Programmers use an extensive vocabulary and thus the number of Type I and Type II violations will increase when natural language synonyms are taken into account

H_a : Programmers use a limited vocabulary and thus the number of Type I and Type II violations will be unaffected when natural language synonyms are taken into account

Figure 6 shows a graphical comparison of the relation between the percentage of violations with and without using WordNet's synonym sets. For Type I (black) and Type II (grey), the jagged (solid) line shows the impact of using synonyms as compared to the original data shown by the dashed line. Visually, the difference is minimal. Statistically, Pearson's linear regression is used to model the relation between the percentage of violations with and without using synonym sets. For Type I violations, incorporating natural language synonyms increases the number of violations 2.8% ($R^2 = 0.998$). For Type II violations, the increase is only 2.1% ($R^2 = 0.996$).

The models for Type I and Type II violations show a statistically significant increase in violations when natural language synonyms are incorporated. However, as the magnitude of the increase is quite small, the models support rejecting the null hypothesis. In other words, the rather minimal increase for both Type I and Type II violations supports the observation that programmers use a limited vocabulary. In particular, they do not use a significant number of natural language synonyms.

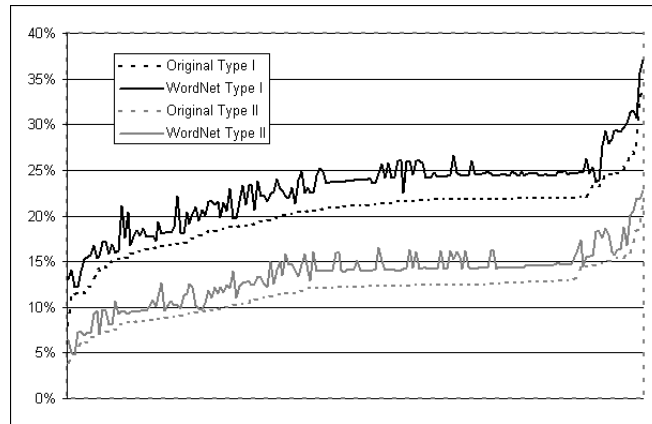


Figure 6. Incorporating natural language synonyms from WordNet. The x-axis shows each program sorted separately for Type I and Type II violations without using WordNet.

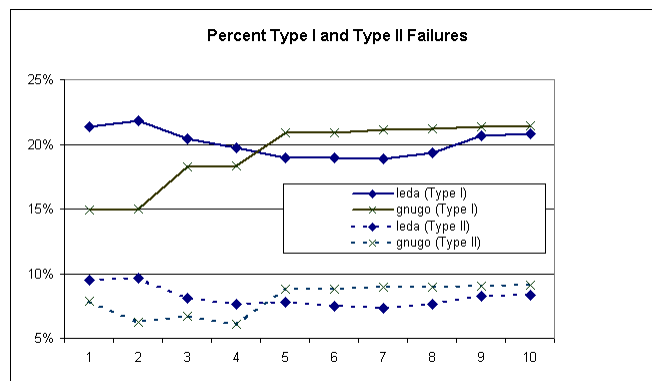


Figure 7. Two example programs from the longitudinal study.

Longitudinal Study

The second of the three hypotheses addresses the question “Does evolution introduce conciseness and consistency failures?” In principal, if a program takes on new concepts as



it ages, then identifiers that were previously consistent and concise may become inconsistent and “un-concise”. This occurs when software evolution introduces new identifiers (and their associated concepts). For example, the program `which`, actually the `getopt` library, originally only processed short-form command line options. Later, a long form was added. The current code includes the identifiers `options` and `long_options`. Knowing the code’s history, `options` is understood to be associated with the concept of *short options*. While `options` was originally a consistent identifier, the introduction of the concept of *long options* means that it is no longer concise. More formally this subsection investigates the hypothesis:

Longitudinal Hypothesis

H_0 : Software maintenance does not introduce conciseness and consistency failures.

H_a : Software maintenance introduces conciseness and consistency failures.

Seven of the programs studied included four or more versions. Pearson’s linear correlations predicting the percentage of Type I and Type II failures as a function of the version number were constructed for each program. Overall there is insufficient evidence to reject the null hypothesis; thus one cannot conclude that evolution introduces failures. This is visually apparent with the two examples shown in Figure 7. `Leda` is typical of most of the programs showing some ups and downs but no significant trend. `Gnugo` shows a slight increase early, but then levels out and remains flat from versions 10 through 70.

The relevant statistics are shown in Table VI. The first thing to notice in the data is that the slope for each regression line is less than a third of 1%. Only one model, for the Type II violations of Mozilla, shows a significant correlation where 63% of the variation in Type II failures are explained by variation in version. However, in this model (in which the slope is



Table VI. Key Statistics for Longitudinal Study. (Statistically significant values in bold)

Program	Type I Failures			Type II Failures		
	R^2	slope	slope p	R^2	slope	slope p
tomcat	0.07	0.19%	0.67	0.06	-0.02%	0.68
mozilla	0.12	-0.01%	0.45	0.63	-0.04%	0.03
I6	0.00	0.00%	0.97	0.38	-0.05%	0.20
gnugo	0.32	0.06%	< 0.001	0.32	0.03%	< 0.001
cpm	0.07	-0.13%	0.74	0.22	0.11%	0.53
barcode	0.14	-0.14%	0.31	0.41	-0.27%	0.06
leda	0.13	-0.13%	0.31	0.28	-0.14%	0.11

statistically significant with a p -value of 0.03), the slope is negative and quite small: it would take 25 releases for the number of Type II failures to drop by 1%.

Looking at its early development, **gnugo** actually shows a significant correlation. When predicting Type I failures, the max occurs when considering the first 13 versions. Here R^2 is 0.60 and the slope is 0.24% (with a p -value of 0.002). For Type II failures, the max occurs at 10 versions with an R^2 of 0.80 and the slope is 0.76% (p -value of 0.001). However, the slopes of the resulting linear equations (0.76% and 0.25%, respectively) are quite small and as more version are considered the lines flatten out. Thus, there is insufficient evidence to reject the null hypothesis; one cannot conclude that evolution introduces failures.



Programming Model

The final of the three hypotheses addresses a question that often arises when one considers both open and proprietary source is “do the two differ?” The programming culture for open source systems tends to include “a spectrum of processes from undefined and flexible processes to some extent defined and controlled processes among open-source projects” [11]. In contrast the culture in which proprietary program is produced is often quite strict. Including strict style guidelines that encourage uniformity across all of the source code gives the engineers less freedom in coming up with identifiers. This section considers the impact of the programming models used by these two cultures on conciseness and consistency failures:

Programming Model Hypothesis

H_0 : Programming model (open-source versus proprietary) does not affect the number of conciseness and consistency failures.

H_a : Programming model (open-source versus proprietary) affects the number of conciseness and consistency failures.

Linear mixed effect regression models are used to investigate the programming model hypothesis. For Type I and Type II violations, backward elimination started with the explanatory variables *open source* along with the other potential explanatory variables *program size*, *start year*, *release year*, and *programming language*. For Type I failures the resulting model includes only *release year* and *programming language*. As *open source* is absent from the model it does not play a significant role in predicting the percentage of failures. It is interesting to note that when compared directly (using a Mann-Whitney test as the distributions are not normally distributed) more violations occur in open-source code than in proprietary-source



code ($p = 0.0003$). Together these results imply that other explanatory variables are more important in explaining the difference in the percentage of Type I failures.

For Type II failures, the resulting mixed effect regression model (starting from the same set of explanatory variables as with Type I failures) includes the explanatory variables *start year*, *release year*, *programming language*, and *open-source*. However, there exists an interaction between *open source* and *release year*. Thus, *release year* has a different effect when considering open and proprietary source code. In this case, an increase in *release year* brings a reduction of 0.28% in the percentage of Type II failures in proprietary code, while it brings an increase of 0.14% to the percentage of Type II failures in open source code. A possible cause for this is greater use of engineering discipline in the construction of proprietary source, which leads to more concise and consistent naming. At the same time, the growth in size of and number of programmers working on open source projects has the opposite effect.

In summary, for Type I violations there is evidence (from the Mann-Whitney test) that open source code includes more violations; however, the difference is overshadowed by other explanatory variables in the mixed effects model. For Type II violations, the mixed effect model indicates that the percentage of violations is dropping as the field matures, which may be caused by a more predominant engineering discipline. However, for open code, the percentage is growing. Based on these models, in particular the one for Type II violations, the null hypothesis is rejected.



7. Related Work

This section briefly surveys work related to that presented herein. Anquetil and Lethbridge (among others) have observed that there is some controversy over the value of general identifier names [1]. For example, Sneed finds that “in many legacy systems, procedures and data are named arbitrarily ... programmers often choose to name procedures after their girlfriends or favorite sportsmen” [19]. A similar pattern was observed by one of the authors at a previous industrial position in the code of a colleague who was fond of Star Wars. This controversy on the value of dictionary word identifiers was also noted by Takang et al. [20]. For example, Shneiderman and Mayer report that “variable names had a statistical significance on comprehension.” However, their study included only beginning students as participants. On the flip side, Sheppard et al. observe that “variable names did not have a statistical significance on the subject’s performance.” This was based on an experiment that involved 36 professional programmers. In this second experiment, the programs were quite small (they varied between 26 to 57 lines of code), which may have been too short to bring out differences especially with professional programmers. The study presented herein follows Anquetil and Lethbridge in assuming that software engineers are trying to give significant names (although they may have failed in this attempt) [1]. With the spread of true engineering discipline in the software construction process, this assumption grows increasingly more likely to be satisfied.

Anquetil and Lethbridge consider extracting information from type names in a large Pascal application [1]. In their definition two records implement the same *concept* if they have similar field names and types (though they are lax on enforcing type equivalence). Thus, this work provides a framework in which to study a form of concept identification (or at least concept



equivalence) through types. The main outcome of their work is the notion of a word based conceptual similarity metric for Pascal record types (a similar notion would apply to structure or class types). The technique assumes the existence of “word markers” (*i.e.*, only hard words were compared). An improved understanding of splitting identifiers into soft words allows the work of Anquetil and Lethbridge to be extended to programs without well separated identifiers. Furthermore, understanding the number of words (soft or hard) per identifier helps to understand how well the conceptual similarity should be expected to work.

Type names may also play a useful role in identifying concise and consistent identifiers. Type information is an example of the kind of information that a fact extractor (*e.g.*, Columbus [9]) can extract about identifiers. For example, `tree_node` is contained in `visit_tree_node`, and `position` is contained in `absolute_position`. Knowing that `visit_tree_node` is a function and `tree_node` a formal parameter of the function indicates that the two are associated with different concepts and thus not a violation of the synonym rule in the same way that two global integer variables `position` and `absolute_position` are. A related refinement would not consider the identifiers `hash_fn`, `node_hash_fn`, and `ast_hash_fn` a conciseness violation if extracted facts indicated that `hash_fn` was function pointer in an ADT and the other identifiers were functions whose address was passed to this pointer.

Caprile and Tonella analyze function identifiers by considering their lexical, syntactical, and semantical structure [4]. They later present an approach for restructuring function names aimed at improving their meaningfulness [5]. The analysis involves breaking identifiers into well separated words (*i.e.*, hard words). The restructuring involves two steps. First, a lexicon is standardized by using only standard terms for composing words within identifiers. Second,



the arrangement of standard terms into a sequence has to respect a grammar that conveys additional information. For example, the syntax of an indirect action, where the verb is implicit, is different from the syntax of a direct action. They were able to come up with an effective grammar for the restricted domain of function identifiers. Extending this to all identifiers is a non-trivial task, but the resulting grammar would be useful in refining the notion of syntactic consistency and conciseness.

Two quality measures for the resulting grammar are considered. First, the coverage of a grammar is the ratio of strings of the language for which at least one syntactic derivation can be obtained to the set of all strings in the language. Second, grammar ambiguity measures the possibility of producing a given string of the language with more than one syntactic derivation. In addition to grammar coverage and ambiguity, other metrics are also proposed to measure the value of identifiers in a program. For example, the average number of words per identifier and the frequency of use of abbreviations.

Deißenböck and Pizka stress the value of identifiers in source code [6] as they make up a significant amount of the unique information available from the source. For example, Eclipse 3.0M7 has 94,829 different identifiers which is around the same number of words as in Oxford Advanced Learner's Dictionary. They also introduce a tool that enforces the rules for consistent and concise identifiers during program construction. This is done with the aid of an identifier dictionary that holds the concept mapping. They find that the tool improves the productivity of programmers.

Finally, identifiers play a key role in several applications of information retrieval (IR) to software. For example, the early work of Maarek [13], which used IR techniques to



automatically construct software libraries, made heavy use of identifiers. More recently, Marcus et al. used IR techniques to identify semantic similarities between source code documents [14]. Based on IR techniques, similar high-level concepts (*e.g.*, abstract data types) are extracted as identified clusters in the code. In similar work, Kawaguchi et al. describe an automatic software categorization algorithm to help find similar software systems in software archives [10]. They explore several known approaches including code clones-based similarity metric, decision trees, and latent semantic analysis. Finally, in a related vein, Marcus et al. address the problem of concept location using latent semantic analysis [15]. Two concept locators are presented—one based on user queries and the other on partially automated queries. As each of these techniques relies on the information content of the identifiers, well formed identifies, such as those satisfying the rules presented in Section 3, should improve each of these techniques.

8. Future Challenges

This section considers inward looking and outward looking future challenges. The two inward looking challenges consider false positives and dealing with abbreviations. First, the case study from Section 5 suggests that it is possible to improve the identification of violations by exploiting certain grammatical patterns. This is, in essence, the start of a grammar-based technique similar to the function-name grammar of Caprile and Tonella [5]. For example, two common patterns seen in the empirical studies have an identifier that is a noun phrase and another that includes a verb or an adjective with the noun phrase. For example, `tree_node` and `visit_tree_node` is an example of the verb-noun phrase pattern while `bit` and `highest_bit` is an example of the adjective-noun pattern. In the first example, syntactically, `tree_node` is



contained in `visit_tree_node` and is thus a (syntactic) violation. However, these two identifiers are associated with different (related) concepts, and `tree_node` is a concise name for the concept. Thus, no violation exists in the Deißeböck and Pizka sense.

A preliminary study counted the violations that matched either pattern. The verb form accounts for 4.5% of the synonym violations. This is consistent with the percentage identified in the exhaustive case study of Section 5. The adjective form accounts for 2.2%, or about half as many of the violations. Together the two grammar based patterns identified 6.7% of the violations. Assuming that the case study from Section 5 is representative, this represents about one quarter of the false positives.

Second, the current tool does not discover the violation that occurs between `absolute_path` and `abs_path` because `abs` is an abbreviation of `absolute`. Definitions 1 and 2 could be broadened to include such cases as follows: for soft word w , let $A(w)$ denote the set of all dictionary words appearing in the program that map to the same concept as w . In Definitions 1 and 2 replace $w_1 w_2 \cdots sw_1 sw_2 \cdots sw_n \cdots w_m$ with $w_1 w_2 \cdots a_1 a_2 \cdots a_n \cdots w_m$, where $a_i \in A(sw_i)$. As `absolute` is in $A(\text{abs})$ the above violation would be detected as a violation.

The two outward looking challenges consider improving tools by using well-formed identifiers and building new tools using the approach. The ability to automatically detect violations can improve solutions to other problems. For example, most fault detectors use only various structural measures of the code [16]. Given that well-formed identifiers have a positive impact on comprehension [6], a measure based on violations of the rules from Section 3 would provide a new information that should improve the fault prediction models.

A second outward looking idea considers tools support for identifying Type I and Type II



violations that is integrated into an IDE. Support could range from alerting a programmer of a violation introduced by a new identifier to forcing the use of names that do not produce violations, or perhaps suggesting replacement variable names as Deißböck and Pizka do in their prototype tool [6]. In the latter case natural language processing techniques might be employed. For example, when *long_options* is first introduced the programmer can be made aware that it conflicts with existing identifier *options*. This implies that the existing container identifier *options* should be replaced with an identifier associated with the concept “all options that are not long.” A tool like WordNet can then be used to propose antonyms, which leads to the name *short_options* as a replacement for *options*. If multiple replacements were possible then these could be presented to the programmer. Finally, existing program transformations (such as Eclipse’s rename refactoring) would be used to update the source code.

9. Summary

Modern programming languages provide programmers with too much freedom when selecting identifiers. Informally, this is quite clear from the large number of violations considered while conducting this study. There is a real comprehension cost when identifiers are not concise and consistent. In one example, the meaning of a function returning the identifier *found* took considerable time to identify. Replacing the returned identifier with the concise and consistent identifier *found_free_space* removed the violation and made the goal of the function considerably clearer. Formally, several studies have noted the importance of making careful choices for identifier names [12, 6, 18, 5]. The rules presented in this paper provide an easily automated



method for restricting the choices that a programmer has. The resulting well-formed identifiers improve code quality [6].

The empirical study presented in Section 5 shows that violations of the syntactic rules for well-formed identifiers occur in practice. Furthermore, the second study shows that the violations found largely match those found using Deißeböck and Pizka's concept-based approach. Thus, easily automateable tool support based on these definitions is viable. Empirical study was also used in Section 6 to better understand the causes (and non-causes) of violations. The most interesting result from this section is that the percent of identifiers violating the syntactic rules in open and proprietary source differ.

10. Acknowledgments

Chris Morrell and Beth Domholdt provided invaluable statistical assistance. Jeri Hanly and Margaret Daley were extremely helpful in editing the paper. Finally, the anonymous reviewers, in particular Referee 1, provided many thoughtful comments that greatly improved the paper. This work is supported by National Science Foundation grant CCR0305330.

REFERENCES

1. N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada, November 1998.
2. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10), October 2002.



-
3. D. Sjøberg, J. Hannay, O. Hansen, V. Kampenes, A. Karahasanovic, N. Liborg, and A. Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 19(4), 1993.
 4. B. Caprile and P. Tonella. Nomen est omen: analyzing the language of function identifiers. In *Working Conference on Reverse Engineering*, Atlanta, Georgia, USA, October 1999.
 5. B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM*, 2000.
 6. F. Deißböck and M. Pizka. Concise and consistent naming. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005)*, St. Louis, MO, USA, May 2005. IEEE Computer Society.
 7. H. Feild, D. Binkley, and D. Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA 2006)*, Dallas, TX, November 2006.
 8. C. Fellbaum, editor. *WordNet – An Electronic Lexical Database*. MIT press, 1998.
 9. R. Ferenc, Á Beszédés, M. Tarkiainen, and T. Gyimóth. Columbus - reverse engineering tool and schema for c++. In *IEEE International Conference on Software Maintenance (ICSM 2002)*, Montreal, Canada, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.
 10. S. Kawaguchi, P.K. Garg, M M. Matsushita, and K. Inoue. Automatic categorization algorithm for evolvable software archive. In *Proceedings of International Workshop on Principles of Software Evolution*, Helsinki, Finland, September 2003.
 11. G. Koru and J. Tian. Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. *IEEE Transactions on Software Engineering*, 33(8), August 2007.
 12. D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *14th International Conference on Program Comprehension*, 2006.
 13. Y.S. Maarek, D.M. Berry, and G.E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8), 1991.
 14. A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *Proceedings of Automated Software Engineering*, San Diego, CA, November 2001.
 15. A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *IEEE Working Conference on Reverse Engineering*, Delft, The Netherlands, November
-



-
- 2004.
16. T. Menzies, J. Greenwald, and A. Fransk. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1), January 2007.
 17. C. Morrell, J. Pearson, and L. Brant. Linear transformation of linear mixed effects models. *The American Statistician*, 51, 1997.
 18. J. Rilling and T. Klemola. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, Portland, Oregon, USA, May 2003.
 19. H. Sneed. Object-oriented cobol recycling. In *3rd Working Conference on Reverse Engineering*. IEEE Computer Society., 1996.
 20. A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: an experiential study. *Journal of Program Languages*, 4(3), 1996.
 21. G. Verbeke and G. Molenberghs. *Linear mixed models for longitudinal data*. Springer-Verlag, New York, second edition, 2001.
 22. Webster. *Collegiate Dictionary, 11th Edition*. Merriam-Webster, 2003.
 23. David A. Wheeler. SLOC count user's guide, 2005. <http://www.dwheeler.com/sloccount/sloccount.html>.