Empir Software Eng (2007) 12:359–388
DOI 10.1007/s10664-006-9032-2

Quantifying identifier quality: an analysis of trends

Dawn Lawrie · Henry Feild · David Binkley

Published online: 21 December 2006
© Springer Science + Business Media, LLC 2006
Editor: Giuliano Antoniol

Abstract Identifiers, which represent the defined concepts in a program, account for, by some measures, almost three quarters of source code. The makeup of identifiers plays a key role in how well they communicate these defined concepts. An empirical study of identifier quality based on almost 50 million lines of code, covering thirty years, four programming languages, and both open and proprietary source is presented. For the purposes of the study, identifier quality is conservatively defined as the possibility of constructing the identifier out of dictionary words or known abbreviations. Four hypotheses related to identifier quality are considered using linear mixed effect regression models. For example, the first hypothesis is that modern programs include higher quality identifiers than older ones. In this case, the results show that better programming practices are producing higher quality identifies. Results also confirm some commonly held beliefs, such as proprietary code having more acronyms than open source code.

Keywords Software quality characterizations · Program analysis · Source code

1 Introduction

This paper characterizes identifier usage over three decades, four popular programming languages, and 186 programs. The programs include over 48

D. Lawrie (✉) · H. Feild · D. Binkley
Loyola College in Maryland, Baltimore, MD 21210, USA
e-mail: lawrie@cs.loyola.edu

H. Feild
e-mail: hfeild@cs.loyola.edu

D. Binkley
e-mail: binkley@cs.loyola.edu

million lines of code and cover both open and proprietary source. This enables the study of the effect of history, language, maintenance, and development methodology on identifiers.

Motivation for the analysis of identifiers comes from several previous studies. For example, Deißenböck and Pizka observe that "Research on the cognitive processes of language and text understanding shows that it is the semantics inherent to words that determine the comprehension process" (Deißenböck and Pizka, 2005). Thus, they conclude that the importance of identifier names is crucial to program comprehension (Deißenböck and Pizka, 2005). A second motivation comes from the work of Caprile and Tonella, who conclude that "Identifier names are one of the most important sources of information about program entities" (Caprile and Tonella, 2000).

Furthermore, Rilling and Klemola observe that "In computer programs, identifiers represent defined concepts [where] identifier density corresponds to comprehension cost" (Rilling and Klemola, 2003). Knuth noted that descriptive identifiers are a clear indicator of code quality and comprehensibility (Knuth, 2003). As a measure of how much of a program is devoted to identifiers, Deißenböck and Pizka report that in the source for Eclipse (about 2 MLoC) 33% of the tokens and 72% of characters are devoted to identifiers (Deißenböck and Pizka, 2005).

Finally, Antoniol et al., observe that most of the application-domain knowledge that programmers possess when writing code is captured by identifier mnemonics (Antoniol, 2002). Thus, how readily the semantics inherent to identifiers can be extracted is of key importance. They write, "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." Thus, an underlying premise of their work is that programmers use organized methodical meaningful (high quality) identifiers for code items.

Following Takang et al., Brook's theory of program comprehension underpins the theoretical framework of this analysis (Takang et al., 1996). 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. This processes is aided by methodical identifier choices, such as those that follow the correctness and conciseness rules introduced by Deißenböck and Pizka (Deißenböck and Pizka, 2005).

Thus, to comprehend a program, an engineer must map the identifiers to the concepts they represent. Jones notes that a variety of different kinds of character sequences are used in source code identifiers (Jones, 2004). Some are complete words or phrases, some abbreviated forms of words or phrases, while others have no obvious association with any known language. Studies have found that people's performance in processing character sequences can vary between different kinds of sequences (Jones, 2004; Lawrie et al., 2006).

For instance, frequently used character sequences (e.g., dictionary words) are recognized faster and are more readily recalled than rare ones (Jones, 2004). Thus, the more meaningful the identifiers of a program, the easier it is to map them to appropriate concepts. As an example, compare, `pqins()` with `priority_queue_insert()`.

Anquetil and Lethbridge (among others) have observed that there is some controversy over the value of general identifier names (Anquetil and Lethbridge, 1998b). For example, Sneed finds that "in many legacy systems, procedures and data are named arbitrarily $\cdots$ programmers often choose to name procedures after their girlfriends or favorite sportsmen" (Sneed, 1996). 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.

The study presented herein follows Anquetil and Lethbridge in assuming that software engineers are trying to give significant meaningful names (although they may have failed in this attempt) (Anquetil and Lethbridge, 1998b). With the spread of true engineering discipline in the software construction process, this assumption grows increasingly more likely to be satisfied.

Summarizing the above observations, within this paper, the quality of identifiers comes from their ability to help a programmer identify domain level concepts in order to better comprehend or manipulate the code (Jones, 2004; Ratiu and Deissenboeck, 2006). In general specifying that exact set of identifiers that bring to mind the correct concepts for a given programmer is highly programmer dependent. As this varies significantly from programmer to programmer, a conservative definition of identifier quality is used in the study. Following previous work (Jones, 2004; Lawrie et al., 2006), identifier quality is assumed to be tied to the use of natural language words, identifiable coherent abbreviations, and common library identifiers.

Using the definition above to define quality, the four main contributions of this study are the investigations of the following four hypotheses:

Hypothesis 1: Historical
H0: Modern programs contain the same quality identifiers as older programs.
Ha: Modern programs contain higher quality identifiers.

Hypothesis 2: Longitudinal
H0: Identifier quality for a given program remains constant as the program ages.
Ha: Identifier quality for a given program improves with age.

Hypothesis 3: Development model
H0: Open and proprietary source include the same identifier quality.
Ha: Open and proprietary source include different identifier quality.

Hypothesis 4: Programming language
H0: Programming language choice has no effect on identifier quality.
Ha: Programming language choice effects identifier quality.

As highlighted by the variety of the work cited above, identifiers play a key role in code quality (e.g., Knuth's observation that descriptive identifiers are a clear indicator of code quality and comprehensibility; Knuth, 2003). This study is based upon these results. For example, that of Jones regarding the impact that different kinds of identifier character sequences on the cognitive resources needed during program comprehension (Jones, 2004). In particular, the positive impact on recall that attachment to meaning in long term memory (i.e., dictionary words and well known abbreviations) has.

In fairness, quality programs can have poor identifiers (even though it is expected that normally high quality identifiers will accompany high quality programs). However, the study takes as an underlying assumption that programmers are trying to write programs with useful identifier names. They are not, for example, trying to mislead by choosing identifiers with no correlation to domain level concepts. Thus, for example, it is assumed that the meaning of an identifier that is made up of two dictionary words is somehow related to the meaning of the two words.

The rest of this paper first presents some necessary background material in Section 2. Investigation of the four hypotheses is done in Sections 3 and 4. Related work is then considered in Section 5. Finally, the paper concludes with a discussion of some topics for future investigation and a summary in Sections 6 and 7.

2 Background

This section describes how identifiers are analyzed, the tools that carry out the analysis, the subject programs studied, and the statistical tests employed in the analysis.

2.1 Identifier Analysis

Past studies of identifier quality typically begin by dividing identifiers into their constituent parts (Caprile and Tonella, 1999; Deißenböck and Pizka, 2005; Caprile and Tonella, 2000; Rilling and Klemola, 2003; Antoniol, 2002; Jones, 2004; Lawrie et al., 2006). Herein, these parts are called *words*. Two kinds of words are considered: *hard words* and *soft words*. Hard words are marked out by the programmer through the use of *word markers* (Caprile and Tonella, 1999). Herein the two most popular word markers, camel-casing and underscores, are used. The strings of characters between word markers and the endpoints of an identifier are referred to as *hard words*. For example, the identifiers sponge_bob and spongeBob both contain the well separated hard words sponge and bob.

Hard words are made up of one or more *soft words*. Those hard words that identify a single concept are considered a single soft word. Those that include multiple concepts are composed of multiple soft words. In this case the programmer could have included word markers to separate these soft words.

For example the identifier hashtable_entry consists of one word 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.

As introduced in Section 1, a rather conservative measure of quality is used in which the soft words that make up quality identifiers are assumed to be made up of words from one of three lists: a list of *dictionary* words, a list of well known *abbreviations*, and finally, borrowing an idea from Information Retrieval, a *stop-list*. In the sequel, when a hard or soft word are on one of these three lists, it is referred to as being "on a list."

For the list of dictionary words, the study uses the publicly available dictionaries that accompany the Linux spell checker ispell (Version 3.1.20). The source studied included English, French, German, and Italian; thus these four dictionaries were used. The list of abbreviations includes domain abbreviations (e.g., alt for altitude) and programming abbreviations (e.g., txt for text and msg for message). This list of only about 200 common abbreviations, clearly does not contain all abbreviation used in the analyzed code. The list was kept short in order to ensure that all entries were clearly abbreviations. As described in Section 6, one area of future work will investigate automatically recognizing abbreviations and their expansion into dictionary words using machine translation techniques. Finally, a stop-list is used to omit words that are not thought to bring useful information. In English, words such as *the* are typically eliminated. For identifiers, the stop-list contains three types of words: predefined type names (e.g., int), language or environment global variables (e.g., errno), and standard library names (e.g., printf). All three lists were held constant during the analysis.

The analysis of an identifier begins by splitting the identifier into hard words. Those hard words not found on a list are assumed to by composed of multiple soft words. In this case, a *splitting algorithm* is used in an attempt to identify the constituent soft words. Split hard words are shown using '-' as a word marker. For example, the splitter divides the single hard-word identifier zeroindeg into the three soft words zero-in-deg (i.e., zero, in, and deg). (One related future question is the cognitive load placed on a programmer when they must mentally split identifiers such as zeroindeg. If this cost is high enough, then tools that can correctly split up hard words are of value to programmers reading code, especially unfamiliar code.)

A greedy algorithm is used to identify soft words. It looks for the longest prefix and the longest suffix that are "on a list" (i.e., in the dictionary, on the list of abbreviations, or on the stop list). Starting with each (non-list) hard word, the algorithm conducts two searches and keeps the better result. The first search finds the longest on-a-list prefix of the current word. It then recursively calls itself on the remaining portion of the original hard word (this recursive call considers both prefixes and suffixes). The second search is the same as the first except it searches for the longest suffix. The results of the two searches are compared and the one producing the higher ratio of *on a list* to *total* soft words is chosen. If neither the prefix nor the suffix search generates any on-a-list words, the process is repeated with the first character removed from the identifier. Removed characters are gathered together and presented on the result separated by the '-' (artificial) word marker.

2.2 Extraction Tool

The first stage of the extraction tool is a simple lex-based scanner (essentially implementing an extreme Island Grammar (Moonen, 2001)). It outputs a list of all program identifiers. This list is then sent to a Perl script that splits each identifier into first hard words and then soft words using the greedy algorithm. The output for each identifier includes the following "demographic" information: program name, dominant programming language, source file, project start year, and project release year. The data also includes the original identifier, the identifier after its division into soft words, the number of occurrences of the identifier in the file, and the number of hard words, soft words, soft words found in the dictionary, soft words found on the abbreviation list, soft words found on the stop list, soft words that are single letters, and finally soft words placed in an "other" category. Single letter soft words are separated out because there are a sufficient number of them to generate misleading results.

At present, the "other" category catches program specific abbreviations, acronyms, and other unknown abbreviations. For example, fixsgrel ("FIX a Simple GRid and ELement") is a program-specific abbreviation from I3, while bc ("Bar Code") is a program-specific acronym from barcode. Examples of other unknown abbreviations include circ abbreviating circle and geom abbreviating geometry. To improve the performance of the greedy algorithm, such abbreviations can be added to the list of well known abbreviations. However, during the data collection the list of abbreviations was frozen to prevent any bias that would favor one program over another; thus, abbreviations, such as geom, discovered during the analysis where not added to the list of standard abbreviations. Other, less recognizable, unknown abbreviations such as gusism and cfzon are also placed in the "other" category. Finally, this category includes soft words that contain digits (e.g., p2).

For example, Fig. 1 shows a simple program and an excerpt from the output of the greedy algorithm. Excluded from each line of the table is the demographic information: program name (binsearch-example), dominant programming language (C), source file, (binsearch-snippet.c), project start year (2006), and project release year (2006). The example includes two identifiers with more than one soft word. One of these, binsearch, includes only one hard word. In the column showing the split identifiers, a "-" is used as a soft word separator.

2.3 Subject Programs

The analysis includes 78 unique programs. Given that a longitudinal study is considered, multiple versions of some programs were obtained. Up to 70

```
binsearch(scores[ ], target_value, left, right)
{
   while (right >= left)
   {
      // excerpt – not all code shown
   }
   return -1;
}
```

Identifier					Soft word breakdown				
Original Original	Split Split	Occur-rences	Hard words	Soft words	dictio-nary	abbre-viation	stop list	single letter	other
binsearch	bin-search	1	1	2	1	1	0	0	0
left	left	2	1	1	1	0	0	0	0
right	right	2	1	1	1	0	0	0	0
scores	scores	1	1	1	1	0	0	0	0
target_value	target-value	1	2	2	2	0	0	0	0

Fig. 1 Greedy algorithm output example—a subset of the output generated from the code above is shown. In the table, the demographic data is not shown

versions of a program were incorporated into the results. Thus, a total of 186 programs, including multiple versions, are analyzed for this study. All but 12 were open source programs, but there exists sufficient proprietary code to obtain statistically significant results. The 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 lines, 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 the dictionary of the programmer's native language or, if multiple languages were evident in the code, the union of the respective dictionaries.

Table 1 shows the C, C++, and Java subject programs studied. The two Fortran programs are not shown in the figure. They are PLM compilers from 1975 and 1981 and include 9,704 and 11,478 LoC, respectively. Difficulty in finding non-trivial Fortran codes made it a challenge to obtain statistically significant results for Fortran.

Table 1 includes 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 (David, 2005), 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 1 Subject Programs (proprietary programs are named I#

	wc				sloc	Year	
Program	C	C++	Java	Total	Total	Start	Release
a2ps-4.12	53,967	0	238	54,205	36,335	1988	1999
acct-6.3.2	7,216	467	0	7,683	5,064	1993	1998
apache_1.3.29	101,456	59	0	101,515	68,756	1995	1999
cinelerra-2.0	1,044,996	106,357	0	1,151,353	820,980	1996	2004
cvs-1.11.1p1	90,987	459	0	91,446	62,644	1989	2001
eclipse-3.2m4	0	5,461	3,082,084	3,087,545	1,800,559	2001	2005
ed	10,173	0	0	10,173	6,805	1992	1994
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
EPWIC-1	8,631	0	0	8,631	5,245	1988	1997
flex-2.4.7	20,156	0	0	20,156	14,455	1990	1994
gcc-2.95	834,738	6,895	0	841,633	593,286	1987	1999
genesis-3.0	0	0	18,977	18,977	11,632	2004	2005
ghostscript-7.07	302,336	2,872	0	305,208	225,459	1988	2003
I2	155,996	0	0	155,996	76,105	2001	2003
I3	27,480	0	0	27,480	19,900	2001	2003
I4.2	2,109,050	398,463	502,965	3,010,478	1,704,823	1993	2004
I9	812,830	668	0	813,498	512,661	1998	2002
I12	11,524	0	0	11,524	5,978	1994	1995
ijpeg	24,822	0	0	24,822	15,238	1991	1994
jaccounting	0	0	11,676	11,676	6,526	2003	2005
james-2.2.0	0	0	83,716	83,716	43,266	2000	2004
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
mysql-5.0.17	570,950	711,902	10,418	1,293,270	940,715	1996	2005
named	120,009	0	0	120,009	82,804	1986	1999

Table 1 continued

	wc				sloc	Year	
Program	C	C++	Java	Total	Total	Start	Release
ntp-4.0.95	83,974	0	0	83,974	56,915	1989	1999
quake3-1.32b	353,806	57,431	0	411,237	281,432	1999	2005
samba-3.0.0	353,638	0	0	353,638	227,067	1993	2003
uupc	1,427	0	0	1,427	949	1987	1987
VE-SDK-1.1	0	0	9,424	9,424	5,972	2002	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		

Other programs studied

asterisk_1.2.1	barcode-0.90 – 0.98 (9)	byacc.1.9	bc	CADP
compress	cook	cpm68k1-v1.1 – 1.3 (4)	ctags-5.0	diffutils-2.7
eclipse-2.1	ed	findutils-4.1	flex-2.5.4	frasr193
gnubg-0.0	gnuchess-4.0.pl80	gnugo-1.2 – 3.5.8 (70)	go	gs5.50
httpd-2.0.48	I4.1 – .2 (2)	I6.1 – .6 (6)	indent-1.10.0	javabb_073
jddac-0.5	jakarta-jmeter-2.1	jakarta-tomcat-3.0 – 5.5 (5)	jikes-1.22	kawa-1.8beta
li	mozilla-1.0 – 1.6 (6)	LEDA-2.1.1 – 3.4.1 (11)	pacifi3d0.3	plm80s
sendmail-8.7.5	snns	spice3f4	termutils	tile-forth-2.1
time-1.7	userv-0.95.0	vaxplm	wdiff-0.5	which-2.9
wu-ftpd-2.5.0				

For a sample some details on the size and era are included. In the list, multiple versions are summarized “first–last (number of versions).”

2.4 Statistical Tests

Several statistical techniques are used in the interpretation of the data gathered during the study. This section introduces these techniques. When a simple linear association is of interest, Pearson's linear regression model, which measures linear correlations between variables, is built. For the effect of explanatory variables X, Y, and Z on response variable A, the resulting model coefficients, m_i, belong to the linear equation

$$A = m_1 X + m_2 Y + m_3 Z + b\,.$$

Each coefficient has an associated p-value. A p-value less than 0.05 represents a significant explanatory variable. In addition the model itself has a p-value that tests whether the coefficients of *all* the explanatory variables are zero (i.e., the explanatory variables have no effect on the response variable). For models of a single explanatory variable, the p-value of the model is the same as that of the variable and is not repeated in the discussion.

For more complex models, linear mixed-effects regression models (Verbeke and Molenberghs, 2001) are used to analyze the data. Such models allow the examination of important effects associated with the response variables. While the construction is more involved (Verbeke and Molenberghs, 2001), in summary an initial model that includes all explanatory variables of interest and their associated interaction terms is created. The interaction terms model when the effect on the response variable of one explanatory variable differs based on another explanatory variable. For example, if the choice of programming language interacts with start year in a model for the percentage of soft words in the dictionary (the response variable), then the effect of programming language on the percentage depends on the start year (i.e., the effect is different for different project starting years).

Backward elimination of statistically non-significant terms ($p > 0.05$) yields the final model. Note that some non-significant variables ($p > 0.05$) are retained to preserve a hierarchical well-formulated model (Morrell et al., 1997). This occurs, for example, when an interaction is significant. In which case the two interacting explanatory variables are retained in the model even if one is not statistically significant.

In these models, Tukey's highly significant difference method for multiple comparisons is used. 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 made to the p-values to account for this multiple testing. 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. As with the linear regressions, the model itself has a p-value that tests whether the coefficients of *all* the explanatory variables are zero.

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 of 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.

3 Basic Identifier Statistics

This section presents some basic summary statistics from the 186 programs studied. This data, which is summarized in Table 2, includes some demographics information (e.g., dominant programming language, and the start and *release year* of the program) and summary statistics regarding the identifiers. These include the number of unique identifiers, the total number of identifier instances, and the number of hard words and soft words. The hard word and soft word columns from Table 2 show the total number of words found in all the identifiers.

The top 14 lines of the table include a representative sample of the programs. The bottom seven rows summarize the data over all programs (not just that of the representative programs in the top of the table). Summaries include two orthogonal groupings (open source versus proprietary and by programming language) and all the data taken collectively.

The programming language summary shows that the 186 programs include 26 MLoC of C, 15 MLoC of C++, 7 MLoC of Java, and 21 KLoC of Fortran. The total of almost 50 MLoC includes almost 3 million unique identifiers and over 55 million identifier instances; thus, the typical identifier appears about 19 times. Interestingly, in mozilla-1.0, 2,093 of 8,000 identifiers appear just once. This occurs primarily when a function is called once or passed once as an actual parameter, and the declaration is part of a standard library (e.g., in the case of C in stdio.h). The 3 million unique identifiers included almost 8 million hard words, or an average of 2.7 hard words per identifier and just over 9 million soft words, or an average of 3.2 soft words per identifier.

To provide some sense of the total 'vocabulary' needed by someone considering these programs, the number of unique hard words and unique soft words for all programs are 131,576 and 35,432. It is interesting to compare these numbers with those of other studies and used in typical person's natural language vocabulary. For example, in a related study Deißenböck and Pizka found that Version 3 of Mozilla includes 7,233 unique hard words (Deißenböck and Pizka, 2005). They go on to remark that this number seems high as speakers of English as second language need a vocabulary size only around 5,000 words to understand academic texts.

Table 3 presents information on the number and kinds of soft words found in the identifiers. The table includes the number of soft words and their division into the five categories of soft words studied. The last seven rows repeat the breakdowns from Table 2. For example, over all programs almost two in three (65%) of the soft words were found in the dictionary. While patterns in this data are the topic of the next section, the dominance of dictionary words is

Table 2 Basic counts for 14 selected programs (Proprietary programs are named I#)

Program	Dominant language	Start year	Release year	Unique ids	Id instances	Hard words	Soft words
cinelerra-2.0	C	1996	2004	84,612	1,833,424	209,059	261,793
cpm68k1-v1.1	C	1974	1983	4,167	79,660	4,560	8,193
eclipse-3.2m4	Java	2001	2005	167,662	3,893,272	554,068	612,632
gcc-2.95	C	1987	1999	44,941	897,728	110,060	146,474
I1	C	1987	1997	30,092	482,228	48,125	82,307
I4.2	C	1993	2004	113,662	2,694,901	328,079	422,364
I6.6	C	2000	2002	10,791	104,290	29,207	34,549
jakarta-tomcat-5.5.11	Java	1999	2005	19,202	351,487	48,537	54,471
mozilla-1.6	C++	1998	2004	189,916	3,649,329	563,448	659,396
mysql-5.0.17	C++	1996	2005	50,383	1,023,362	132,249	163,363
plm80s	Fortran	1975	1977	581	22,314	581	886
quake3-1.32b	C	1999	2005	31,114	542,664	75,474	94,144
sendmail-8.7.5	C	1983	1996	2,877	62,075	4,492	6,828
spice3f4	C	1985	1993	12,388	452,423	24,599	34,882
Totals for (over all code not just that shown)	Instances per id	Hard words per id	Soft words per id	Unique ids	Id instances	Hard words	Soft words
Open source	19.2	2.7	3.2	2,504,937	48,098,029	6,817,779	8,040,625
Proprietary	19.6	2.7	3.5	385,792	7,543,663	1,055,329	1,331,327
C	18.6	2.5	3.1	1,566,289	2,9074,119	3,956,372	4,821,045
C++	19.3	2.9	3.5	965,402	18,836,801	2,835,896	3,341,987
Java	22.1	3.0	3.4	356,225	7,885,428	1,076,709	1,203,537
Fortran	18.0	1.4	1.8	2,238	40,273	3,141	3,993
All	19.3	2.7	3.2	2,890,153	55,638,621	7,872,119	9.370,562

Table 3 Soft word counts for selected programs (proprietary source programs are named I#)

Program	Soft words	Percent soft words in Dictionary (%)	Abbreviations (%)	Stop-list (%)	Single letters (%)	Other (%)
a2ps-4.12	9,437	66	13	3	10	8
apache_1.3.29	22,971	63	12	1	12	12
barcode-0.90	377	71	13	0	11	5
cinelerra-2.0	261,793	57	9	1	13	20
cpm68k1-v1.1	8,193	40	17	2	18	23
cvs-1.11.1p1	14,344	64	14	2	11	9
eclipse-3.2m4	612,632	80	4	1	9	6
empire_server	11,375	57	10	1	18	14
gcc-2.95	146,474	61	12	2	15	10
gs5.50	73,666	55	10	1	15	18
httpd-2.0.48	53,385	61	12	1	11	15
I1	82,307	33	11	0	31	25
I4.2	331,875	57	9	1	12	21
I6.6	34,549	48	15	1	12	24
I7	7,031	43	13	3	17	24
jakarta-tomcat-5.5.11	54,471	73	8	1	7	12
LEDA-2.1.1	5,276	70	8	0	16	6
minux-2.0	51,428	52	14	1	17	16
mozilla-1.6	659,396	65	8	1	12	14
mysql-5.0.17	93,852	60	12	2	11	15
plm80s	886	38	14	0	18	29
quake3-1.32b	94,144	64	10	1	13	13
samba-3.0.0	81,496	58	12	2	12	15
sendmail-8.7.5	6,828	56	12	2	16	15
spice3f4	34,882	40	12	1	19	28
uupc	262	59	19	1	13	8
VE-SDK-1.1	3,043	66	1	1	22	10
Totals for all code not just that shown above						
Open source	8,040,625	67	8	1	12	13
Proprietary	1,331,327	53	11	1	14	21
C	4,821,045	60	10	1	13	16
C++	3,341,987	64	9	1	13	14
Java	1,203,537	79	5	1	8	7
Fortran	3,993	53	9	0	18	20
All	9,370,562	65	9	1	12	14

encouraging for those working on identifier-based program comprehension tools.

4 The Study

The study considers the four hypotheses from the introduction. Before this, however, two preliminary statistical tests are described as they simplify later analysis and threats to validity are discussed to provide context. First, the data includes two LoC measures. It turns out that these two provide essentially the same explanatory information. Pearson's linear regression predicting sloc using wc reports $R^2 = 0.992$, indicating that wc explains over 99% of the variation in sloc. The resulting model is

$$\mathsf{sloc} = 0.682 * \mathsf{wc}$$

where the p-value for explanatory variable wc is < 0.0001. Thus, in the following, line counts, as counted by wc, are used.

Second, with a few exceptions, there is a lack of interaction in the models. This simplifies model presentation as the effect of each explanatory variable can be considered separately. The exceptions occur in expected places. For example, in several of the models there is a statistically significant interaction between project *start year* and the programming language Java. As Java was introduced in 1995, this interaction is unsurprising.

In 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 (Sjøberg et al., 1993). 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 the numbers of identifiers, hard words and various kinds of soft words can be accurately counted, 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 (Sjøberg et al., 1993) 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. With two exceptions the variables used in the study can be measured with high accuracy.

Thus, baring these exceptions threats to construct validity are not expected to be a serious concern. The exceptions include the construction of the list of abbreviations and the degree to which identifier quality corresponds to identifiers being constructed from dictionary words and well known abbreviations. 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.

4.1 Is Quality Improving with Time

As computer science has matured as a discipline, the engineers that write programs have received better training. In addition, programming languages have become less stringent in their requirements for identifiers. For example, in some older languages, the length of an identifier is restricted to eight characters. While causality is not testable, the first hypothesis (repeated below) considers whether this maturity is accompanied by an improvement in the quality of identifiers.

Hypothesis 1: Historical

H0: Modern programs contain the same quality identifiers as older programs.

Ha: Modern programs contain higher quality identifiers.

Hypothesis 1 is divided into two parts and tested separately. The first relates to the need for splitting the hard words of the identifiers and the second to the quality of the identified (post splitting) soft words. To help understand the motivation for considering these two, note that there is a comprehension cost to an engineer who must split an identifier into its constituent soft words without the aid of word markers, even given the use of well known vocabulary. For example, consider reading "nowisthetimeforallgoodpeople-tocometotheaidoftheircountry" in the absence of word markers (spaces and punctuation in natural language prose). Once identified, there is also a comprehension cost if these soft words are not well known to the engineer. Thus, Hypothesis 1 is broken into

Hypothesis 1a

H0: Modern identifiers require the same number of hard word splits as older programs.

Ha: Modern identifiers require fewer hard word splits than older programs.

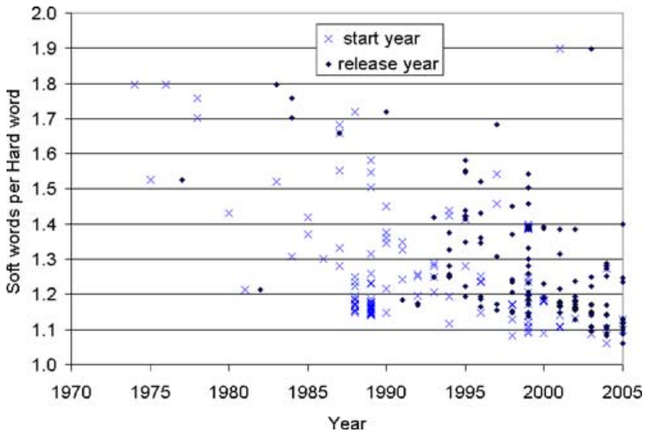

Fig. 2 Ratio of soft words per hard word over time

Hypothesis 1b

H0: Modern programs include the same number of dictionary soft words as older programs.

Ha: Modern programs include more dictionary soft words than older programs.

The investigation of Hypothesis 1a begins with Fig. 2, which compares the ratio of soft words to hard words against the *start year* and *release year* of each program. This ratio provides an indication of how much splitting is required. If the use of word markers is increasing, then less splitting should be required. In Fig. 2, this trend appears to be evident in the data as a trend towards the lower right.

This hypothesis was tested using backward elimination (Morrell et al., 1997) to construct models beginning with the explanatory variables *start-year*, *release-year*, *loc*, *open-source*, and *programming language*. The elimination yields a final model that explains just over half of the variation in the ratio of soft words to hard words ($R^2 = 0.55$, model $p = 0.0002$). Given the number of other factors that might effect this ratio (programmer style, programmer *age*, coding rules, etc.), the explanatory power of the final model is quite good. This final model includes the quantitative variables, *start year* and *release year*, and the qualitative variable *programming language*, whose significance is discussed in Section 4.4. There is, unsurprisingly, an interaction between *start year* and *release year*.

The coefficients of *start year* and *release year* are both negative with *release year* bringing slightly more explanatory power to the model. Thus, programmers are including more word markers, making division easier, and consequently lowering comprehension cost. The exact trend is difficult to describe because of the interaction between *start year* and *release year*. However, as both have negative coefficients, the general trend is for modern programs to

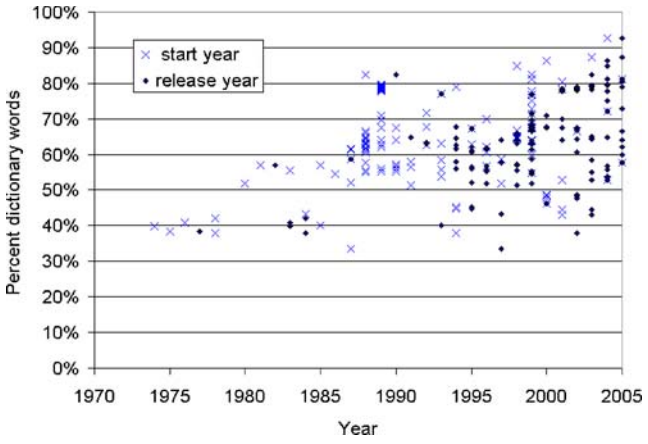

Fig. 3 Percentage of dictionary soft words over time

require less splitting. Finally, multiple comparisons for the different *programming languages* finds two groups that reflect language age. Java and C++ are in one group and C++, C, and Fortran in the other. Thus, Hypothesis 1a is rejected and the alternative hypothesis *modern identifiers require fewer splits* is accepted.

Moving on to Hypothesis 1b, Fig. 3 shows that the decrease in the need to split identifiers is accompanied by an increase in the use of dictionary soft words in identifiers (the general trend towards the upper right). However, this trend is not universal as some programs still include only 40 to 50% dictionary soft words.

Hypothesis 1b is tested using a linear model that includes the same explanatory variables as 1a. Backward elimination yields a model with two differences: first, the interaction between *start year* and *release year* is not significant, and second the explanatory variable *open-source* is significant. The resulting model explains just under two thirds of the variation ($R^2 = 0.61$, model $p < 0.0001$). The sign of the coefficients for *start year* and *release year* are different indicating that program age matters. Thus, the model was re-run with *age* = *release-year* − *start-year* replacing *start-year*. This linear substitution does not change the explanatory power of the model, but makes the results easier to interpret. The lack of interaction allows the effect of *release year* and *age* to be considered separately. In the model, the percentage of dictionary soft words is higher for programs released later (it increases by 1.4% per year). It also increases by 0.4% for every year older that a program is. Finally, the influence of *programming language*, which is again significant, is discussed in Section 4.4, and the influence of *open-source*, which is also significant, is discussed in Section 4.3. Thus, the null hypothesis for 1b can be rejected as well. The alternative hypothesis *modern programs include more dictionary soft words* is accepted.

The above models consider all 186 programs studied. Several of the programs included multiple versions. To factor out any weighting effect, the backwards elimination was repeated using only one version of each program. A systematic method for selecting these versions was needed, so the most recent version was chosen. This choice lowers the influence of *start year*.

In general, the resulting models tell the same story. First, considering hypothesis 1a (the ratio of hard words to soft words), the second model is considerably simpler and includes only the variables *release year* and *programming language*. With an R^2 of 0.42 (model $p < 0.0001$), just under half of the variation in the ratio of soft words to hard words is explained by the model. This is lower than the 0.55 for the model that includes all programs, mostly because fewer data points are included. As before, the coefficient of *release year* is negative and thus the ratio is falling.

For the percentage of soft words in the dictionary, backward elimination removed (in order) *LoC* and *release year*. The resulting model is a good fit, having an R^2 value of 0.54 (model $p < 0.0001$), which, similar to the ratio of hard word to soft words, is lower than the model for all programs. As an interesting aside, *LoC* is not significant; thus, program size has nothing to do with the quality of its identifiers.

The coefficient of the explanatory variable *start year*, 0.48%, indicates that more recent projects include a higher percentage of dictionary words in their identifiers (at a growth rate of almost half a percent per year). A minor result from a related model introduced in Section 4.3 is the coefficient of −0.34% for *release year* indicating that the use of abbreviations is declining with time. The explanatory variables *open source* and *programming language* are again significant. Their contribution to the model is discussed in Sections 4.3 and 4.4.

In summary, for each Hypotheses 1a and 1b, two models were considered. One included all 186 programs studied and the other the latest version of each program. In all four models the *start year*, *release year*, or both are significant explanatory variables. Therefore time is important to identifier quality. The two models for the ratio of soft words to hard words both show a reduction in the need for hard word splitting in more modern programs. The two models for the percentage of dictionary soft words show an increase in this percentage; therefore, Hypothesis 1 is rejected. The alternative hypothesis *modern programs contain higher quality identifiers* is accepted.

4.2 Longitudinal Effects

This section addresses the quality of identifiers over multiple versions of the same program. It examines

Hypothesis 2: Longitudinal

H0: Identifier quality for a given program remains constant as the program ages.

Ha: Identifier quality for a given program improves with age.

The subject programs include four or more versions of six programs. Figure 4 shows each of the six and the percentage of dictionary soft words

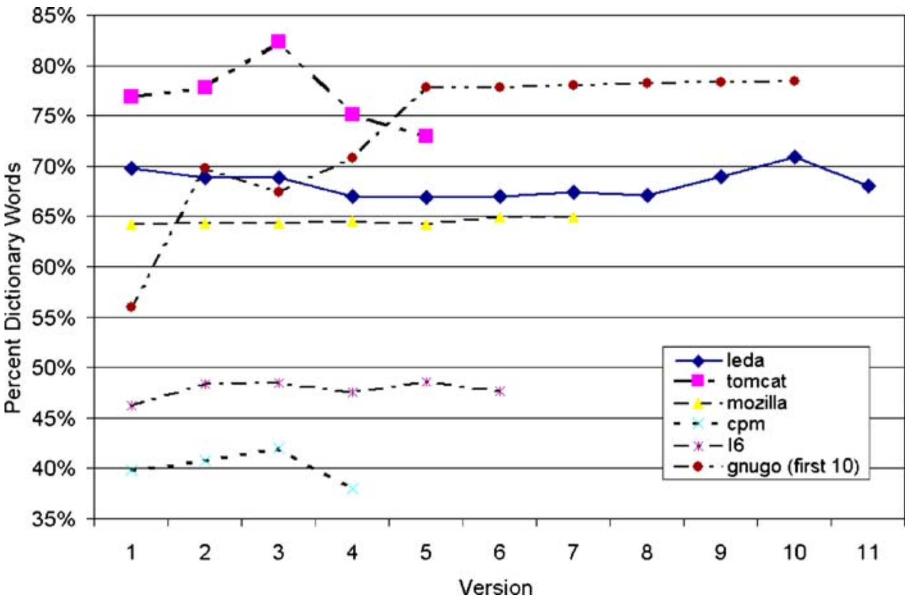

Fig. 4 Percentage of soft words in dictionary for six multi-version programs

in each version except for gnugo which has 70 versions. The data for gnugo flattens out after the first four and five versions; thus, only the first ten are shown in the figure.

Of the six multi-version programs, most cover rather stable periods in the program's development, at least based on size. For example, all six versions of mozilla differ in size by less than 5% with the first and the last version being only 1% different. Two programs, tomcat and leda, show significant growth over the versions available: tomcat triples in size and leda increases by a factor of six.

Despite these differences, as can be seen in Fig. 4, there is very little change in the percentage of dictionary soft words. Most of the programs produce essentially flat lines, indicating that identifier quality takes hold early in a program's development. Programs gnugo, mozilla, and l6 show somewhat of an upward trend, while tomcat and cpm, which are initially the top and the

Table 4 Linear regression models for the longitudinal hypothesis

Program	Versions	R^2	p-value
gnugo-10	11	0.70	0.003*
mozilla	7	0.66	0.027*
tomcat	5	0.22	0.42
gnugo-all	70	0.23	< 0.0001*
l6	6	0.16	0.43
cpm	4	0.11	0.66
leda	11	0.0003	0.96

*Significant p-values

bottom lines (shown in bold in Fig. 4), after reaching a peek, show a downward trend.

To test Hypothesis 2, linear regression models that predict the percentage of dictionary soft words were built for each of the six programs. Table 4 summarizes key statistics of these models. From the R^2 values, only the versions of mozilla and the first ten versions of gnugo show significant improvement: the slopes, 2.04% for gnugo-10 and 0.12% for mozilla, are both positive (though the coefficient for mozilla is rather small). Further evidence of the fall off comes from looking at the first 5, 10, 15, and 20 versions of gnugo. These have coefficient values of 4.5, 2.4, 1.1, and 7% respectively. Each is statistically significant. The remaining five models have considerably less explanatory power and in all but one of the five, explanatory variable *version* does not bring significant information to the model ($p > 0.05$).

Relating these models back to Hypothesis 2, there is rather limited statistical evidence that "young" programs show mild improvement as they age. This comes from versions 1.0 to 1.6 of mozilla and the effect of including an increasing number of versions of gnugo. However the other multi-version programs did not show this behavior; thus, is appears that while identifier quality improves with version, this improvement is not universal and plateaus quickly. One implication of this result is that the improvement in quality over time noted in Section 4.1 comes primarily from new program starts and less from improvements in existing programs.

4.3 Does Open Source Differ from Proprietary

This section examines the similarities and differences between open source and proprietary source programs.

Hypothesis 3: Development Model

H0: Open and proprietary source include the same identifier quality.
Ha: Open and proprietary source include different identifier quality.

In short they are different, as is hinted at by the data in Tables 2 and 3. To more formally test the hypothesis, Table 5 shows key statistics from five simple models that predict the percentage of soft words in the dictionary, abbreviations, stop list, single-letter words, and "other" based solely on whether the source is open or proprietary. All but the model for stop-list words show statistically significant results. However, excluding the dictionary and "other" categories, none of the R^2 values suggest much explanatory power. The trend shown in the table is that open source code includes about 20%

Table 5 Simple model coefficients comparing open and proprietary source

Soft words in	R^2	Coefficient (%)	p-value
Dictionary	0.23	20.20	< 0.0001
Abbreviations	0.11	−4.60	< 0.0001
Stop-list	0.0003	−0.05	0.79
Single-letter	0.052	−3.70	0.002
"Other"	0.35	−11.90	< 0.0001

Table 6 Effect of the explanatory variables *programming language* and *open source* on the percentage of dictionary soft words

Using all version of each program			Using one version of each program		
Category	Mean (%)	Tukey's comparison	Category	Mean (%)	Tukey's comparison
Open source	69	A	Open source	63	A
Proprietary	48	B	Proprietary	49	B
Java	80	A	Java	81	A
C	66	B	C	58	A
C++	65	B C	C++	67	AB
Fortran	48	C	Fortran	48	B

more dictionary words and 5% fewer abbreviations, 4% fewer single letter words, and 12% fewer "other" soft words.

To further investigate this hypothesis, the models for the percentage of soft words in the dictionary (introduced in Section 4.1), and new models for the percentage of soft word abbreviations and "other" soft words are compared. The model for dictionary words supports the observation that the percentage of soft words found in the dictionary is higher for open-source code. As shown on the upper left of Table 6, for the model built using all 186 programs, open source programs have a significantly higher percentage of dictionary soft words (69 versus 48%). This is also true in the model built with one version of each program shown on the right of Table 6, although the gap, while still statistically significant, is slightly smaller (63 versus 49%). As Tukey's groupings for both models indicate, open source programs include a higher percentage of dictionary words in their identifiers.

If the percentage of dictionary words is higher, then the percentages elsewhere must decrease. From the data in Table 3, it appears that "elsewhere" is abbreviations and "other." A model was constructed for each of these (each has a model p-value < 0.0001). For abbreviations, the model has an R^2 value of 0.53; thus, just over half of the variation in the percentage of abbreviations is explained by the model. Backward elimination, which removes the variables *start year* and *LoC*, shows that open-source code includes significantly fewer abbreviations (9 versus 13.3%). Backward elimination of the percentage of "other" soft words eliminates *LoC* and *start year*. The resulting model, with an R^2 of 0.48, shows that open source includes 12.4% fewer soft words in the "other" category.

In summary, both the simple and the backward elimination models refute Hypothesis 3. The data support the observation that open source includes a higher percentage of dictionary words, while proprietary source code includes more abbreviations (some known abbreviations, but mostly unknown abbreviations placed in the "other" category). Inspection of the source reveals that many of the soft words in "other" are domain and often company specific acronyms and abbreviations. The implication for tool designers is that tools targeting proprietary source code need to accept a dictionary of locally defined abbreviations or be prepared to learn local abbreviations. From a comprehension standpoint, this means that proprietary source comes with an increased start up cost for beginners as they must learn the local vocabulary. By its very nature, having contribution from a wide range of programmers, open source can support fewer "company" or even project specific abbreviations.

4.4 Language Effects

This section considers the influence of programming language on identifier quality.

Hypothesis 4: Programming Language

H0: Programming language choice has no effect on identifier quality.
Ha: Programming language choice effects identifier quality.

Similar to Section 4.3, five simple models for the five soft word categories were constructed using *programming language* as the sole explanatory variable to test the hypothesis. Key outputs from the models are shown in Table 7. While none of these models explain a significant amount of the variation in the percentages (the highest being 11%), Tukey's multiple comparison separates out Java in four of the five models.

Further data in support of singling out Java comes from the backward elimination models built in Sections 4.1 and 4.3. Multiple comparisons in both models of Section 4.1 for Hypothesis 1a place the different programming languages in two groups: Java and C++ are in the better group and C++, C, and Fortran in the weaker group. Turning to the percentage of dictionary words (Hypothesis 1b), when considering all programs, three groups emerge: Java, C and C++, and C++ and Fortran; thus, Java has a higher percentage of dictionary words where it brings an increase of 14% over C, which is used as

Table 7 Simple model coefficients for *programming language*

Percent soft words in	R^2	Tukey's comparison		
Dictionary	0.110	Java	A	
		C		B
		C++		B
		Fortran		B
Abbreviations	0.084	Java	A	
		C++	A	B
		C		B
		Fortran		B
Stop-list	0.020	Java	A	
		C++	A	
		C	A	
		Fortran	A	
Single-letter	0.081	Java	A	
		C		B
		C++		B
		Fortran		B
"Other"	0.094	Java	A	
		C		B
		C++		B
		Fortran		B

the baseline. This is shown in the left of Table 6. Using one version of each program (the right of Table 6), the difference in percentage is similar, but the difference is not statistically significant. Finally, from the abbreviations model from Section 4.3, Java, C++, and Fortran have statistically fewer abbreviations than Fortran and C (Fortran, being in both groups cannot be shown statistically differentiated from the others).

In conclusion, there are mixed results for Hypothesis 4. In general, *programming language* does not have a large influence because the explanatory power of the models is not very great, so the null hypothesis cannot be universally rejected. However, in some of the models, the newest language, Java, does differentiated itself. It has a lower need for splitting and the highest percentage of dictionary words. Perhaps this is because of the early encouragement of style guidelines for Java. It will be interesting to see if future languages continue this trend.

5 Related Work

This section considers a collection of projects related to the study of identifier names. In general, this paper considers the make up of all kinds of identifiers. It does not look for patterns within certain kinds of identifiers (e.g., function names) nor does it attempt to tie these identifiers to concepts within the program (Ratiu and Deissenboeck, 2006) (a topic of future work). Thus, the primary outcome of this work is a better understanding of identifiers as they appear in existing code. Ideas from the following related research projects suggest ways of improving this understanding and ways to improve other techniques based on this understanding.

First, Anquetil and Lethbridge consider extracting information from type names in a large Pascal application (Anquetil and Lethbridge, 1998b). According to them, 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 this work is the notion of a hard-word-based conceptual similarity metric for record types (a similar notion would apply to structure or class types). The approach builds a list of separate words from each record's field definitions. For example, the hard word based conceptual similarity between these two identifier lists: "cpreport_table_id, cpreport_string_1" and "hm_table_id, hm_string_1" amounts to looking for common words in the following two hard word lists "cpreport, table, id, cpreport, string, 1" and "hm, table, id, hm, string, 1." The technique assumed 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 approach is expected to work.

Caprile and Tonella analyze function identifiers by considering their lexical, syntactical, and semantical structure (Caprile and Tonella, 1999). They later present an approach for restructuring function names aimed at improving their meaningfulness (Caprile and Tonella, 2000). Both of these results support the importance of dictionary words in the quality of identifiers. This later work is related to Anquetil and Lethbridge's earlier work on the analysis of file names (Anquetil and Lethbridge, 1998a). The analysis involves breaking identifiers into hard words—to which some meaning may be associated in the context of the program. 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 itself. For example, the syntax of an indirect action, where the verb is implicit, is different from the syntax of a direct action.

Two quality measures for the resulting grammar are considered. First, the *coverage* of the grammar is the ratio of strings of the language for which at least one syntactic derivation can be obtained to the set of all string in the language. Second, *grammar ambiguity* measures the possibility to produce 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. In this study of ten C programs, the average number of (hard) words per identifier varies between 2.04 for program bc to 3.36 for program cache. For the C programs studied in Section 4, the number of hard words per identifier ranges from 1.0 for gnugo-1.2 to 4.0 for l2. Given the larger code base, this wider range is expected and is consistent with the earlier study.

The third project presents a formal model (and tool) for adequate identifier naming (Deißenböck and Pizka, 2005). The model is based on the observation that "research on the cognitive processes of language and text understanding shows that it is the semantics inherent to words that determine the comprehension process." It includes rules for the *correct* and *concise* naming of identifiers. The rules are defined in terms of the set of all concepts relevant to the program and a mapping from identifiers to concepts, which require a domain expert to construct. Concepts can be ordered. For example, position is more general than absolute_position. A concept lattice computed for the hard words found in a program's identifiers allowed the extraction of meaningful high level concepts.

The next experiment considers the impact of identifier length and meaningfulness on programmer retention. Jones reports on an experiment involving 40 programmers that investigated the impact of limited short term memory capacity on subjects' performance in the comprehension of short code sequences (Jones, 2004). As short term memory only has the capacity to hold information on a few statements at most, identifier names linked to long term memory are important.

Two identifier attributes where studied. First, the amount of short term memory required to hold their spoken form (i.e., the number of syllables) and pre-existence (i.e., was the identifier already established in users long term memory). The identifiers thus belonged to one of four possible sets: a single character whose spoken form contained a single syllable, an English word whose spoken form contained one syllable, three characters not forming a word whose spoken form contained three syllables (e.g., vcq), and an English word whose spoken form contained three syllables.

In terms of correct answers one syllable words received the most correct answers and three unrelated letters the least. In between these two, three syllable words received more correct answers then the single characters. Thus, attachment to meaning in long term memory (i.e., dictionary words and well known abbreviations as studied in Section 4) has a positive impact on recall. These results also seem to suggest that short meaningful identifier names strike the best compromise between meaningfulness and not flooding short term memory.

In the fifth experiment considered, Takang et al. investigate, among other things, the hypothesis that programs that contain full identifier names are more understandable than those with abbreviated names (Takang et al., 1996). There abbreviations were constructed from the first two characters of each word (e.g., '`CalculateNumericScore`' was abbreviated to '`CaNuSc`'). The study involved 89 first and second year computer science students who studied a program for 50 minutes. The source program studied was appropriate for first year students and was thus syntactically and algorithmically trivial. One caveat noted by the authors: "the impact of identifier names on comprehension of small or familiar programs (like the ones in the study) might not be significant enough to be reflected in the test scores." Their hypothesis was supported by the subjective scores but not by the objective test scores.

Furthermore, Takang et al. note that there is some controversy on the value of dictionary word identifiers. 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. were quoted to 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 and 57 lines of code), which may have been too short to bring out differences especially with professional programmers (Lawrie et al., 2006). This observation relates to the memory experiment reported by Jones (Jones, 2004). The need to split hard words, brings up an additional variable: the cognitive load placed on a programmer when they must mentally split identifiers. These factors suggest a larger scale experiment that considers the well separateness of identifiers, and strikes a balance between full word identifiers and abbreviations is needed.

Finally, identifiers play a key role in several applications of information retrieval (IR) techniques to software. For example, the early work of Maarek (Maarek et al., 1991), which used IR to automatically construct software libraries, made heavy use of identifiers. More recently, Marcus et al. used IR to identify semantic similarities between source code documents (Marcus and Maletic, 2001). Based on IR, similar high-level concepts (e.g., abstract data types) are extracted as identified in the code. In similar work, Kawaguchi et al. describe an automatic software categorization algorithm to help find similar software systems in software archives (Kawaguchi et al., 2003). They explore several known approaches including code clone-based similarity metrics, decision trees, and latent semantic analysis. Finally, in a related vein, Marcus et al. address the problem of concept location using latent semantic analysis (Marcus et al., 2004). Two concept locators are presented—one based on user queries and the other on partially automated queries. These projects underscore the importance of identifiers in program comprehension and in particular the inclusion of dictionary words and well know abbreviations.

6 Future Challenges

Code comprehensibility is closely tied to the meaningfulness of the soft words contained in its identifiers. This section discusses two improvements to the algorithm that splits hard words into soft words. The first considers automatically identifying abbreviations, while the second deals with choosing between multiple outputs.

The current hard-word splitting tool uses a fixed list of abbreviations. More sophisticated approaches support automated finding of abbreviations. This would improve the quality of the tool. There are two issues. One is to identify character sequences that are not currently recognized as abbreviations. The second is to associate meaning with each new found abbreviation.

Techniques for machine translation and information retrieval provide guides as to how this can be accomplished. A probabilistic approach is well suited for automatic abbreviation identification. For example, this can be addressed with n-gram language models (McMahon and Smith, 1998). Based on a similar successful approach to finding topics in text (Lawrie, 2003), such models are built from n sequential letters. The relative entropy (Cover and Thomas, 1991) of different language models can be used to find unusually frequent sequences of characters. These are likely to be meaningful sequences of characters for a particular program. For example, the identifier pw_setcmsname is presently split as pw_s-etc-ms-name (having two list words). However, the relative entropy technique should discover that cms, appears frequently in the identifiers of the program and thus identify it as a program specific abbreviation. After doing so, pw_setcmsname would be split correctly as pw_set-cms-name (having three list words).

Once a potential abbreviation is discovered, an associated meaning must be identified. One technique starts with a *wild-card search*. Take, for example, the soft word horiz from the program a2ps. The search would look in the code (and the external documentation) for the string "h.o.r.i.z.," where a "." represents an arbitrary sequence of letters in a single word. In the code, this search uncovers the identifiers seghorizontal and horizontal. Restricting the output to dictionary words leaves horizontal. A manual inspection of the code reveals that horiz is in fact an abbreviation for *horizontal*, which means that this automatic technique has promise. Thus, its dictionary meaning could be associated with the abbreviation horiz. When multiple words are discovered, then the search would assign probabilities based on proximity and frequency to narrow down the list of possible dictionary words.

Finally, using the current method, some hard words have multiple splits that receive the same ratio of "on list" soft words to total soft words. The current tool picks the first of these (essentially a random choice). Machine learning techniques would allow a more informed decision. For example, the splittings of the hard word thenewestone includes the_newest_one and then_ewe_stone, which each have three dictionary words. Two complimentary methods for choosing between these both use a language modeling approach. The simple language model approach would factor word frequencies (from the code, its documentation, or even the associated natural language in general) into the choice. In the case of the hard word thenewestone, the frequency of newest and one are much higher than that of ewe or even then (at least in the code) providing evidence that the_newest_one is the correct division into soft words.

More sophisticated language models could incorporate proximity information into this decision. Consider the example hard word thenewestone. This approach would compare the probability of finding the words "the," "newest," and "one" in close proximity with that of finding the words "then," "ewe," and "stone" in close proximity. Thus, probabilities based on the documentation and frequencies of the word in other identifiers, and even frequencies in general text, would play a role in improving the selection of soft words.

7 Summary

Identifiers are clearly important to comprehending the concepts in a program (Deißenböck and Pizka, 2005; Caprile and Tonella, 2000; Rilling and Klemola, 2003; Knuth, 2003; Antoniol, 2002; Takang et al., 1996; Jones, 2004; Lawrie et al., 2006; Anquetil and Lethbridge, 1998b; Sneed, 1996; Caprile and Tonella, 1999). This is particularly true of comment free *self-documenting* code. For example, the importance of identifier comprehensibility can be seen in the operation of one local company (wishing to remain anonymous) where a senior developer remarked that documentation is light because there are only six developers, each working on all five products. Here, readability and easy comprehension are critical, especially when one works on code written by someone else. It is absolutely necessary for each team member to be able to comprehend the code quickly, without "bugging" the original author. A key factor is meaningful identifiers.

This study shows how identifier usage has improved over the years and how it differs in certain dimensions (e.g., open versus proprietary source or programming language). It is an important step towards understanding the trends that effect the comprehensibility of source code through its identifiers. In particular, for those working on identifier-based program comprehension tools, the dominance of dictionary words is encouraging. Another implication for tool designers is that tools targeting proprietary source code need to accept a dictionary of locally defined abbreviations or be prepared to learn local abbreviations. From a comprehension standpoint, this means that proprietary source comes with an increased start up cost for beginners as they must learn the local vocabulary.

Acknowledgement This work is supported by National Science Foundation grant CCR0305330.

References

Anquetil N, Lethbridge T (1998a) Extracting concepts from file names; a new file clustering criterion. In: 20*th* IEEE International Conference and Software Engineering (ICSE 1998), Kyoto, Japan. IEEE Computer Society Press, Los Alamitos, CA, pp 84–93, April

Anquetil N, Lethbridge T (1998b) 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

Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering traceability links between code and documentation. IEEE Trans Softw Eng 28(10):970–983, October

Caprile B, Tonella P (1999) Nomen est omen: analyzing the language of function identifiers. In: Working conference on reverse engineering, Altanta, GA, October

Caprile B, Tonella P (2000) Restructuring program identifier names. In: Proc. of ICSM'2000, International conference on software maintenance, pp 97–107, San Jose, CA, 11–14 October, 2000

Cover TM, Thomas JA (1991) Elements of information theory. Wiley, New York

David A (2005) Wheeler. SLOC count user's guide, http://www.dwheeler.com/sloccount/sloccount.html

Deißenböck F, Pizka M (2005) Concise and consistent naming. In: Proceedings of the 13th international workshop on program comprehension (IWPC 2005). IEEE Computer Society, St. Louis, MO, May

Jones D (2004) Memory for a short sequence of assignment statements. C Vu 16(6):15–19, December

Kawaguchi S, Garg PK, Matsushita MM, Inoue K (2003) Automatic categorization algorithm for evolvable software archive. In: Proceedings of international workshop on principles of software evolution, Helsinki, Finland, September

Knuth D (2003) Selected papers on computer languages. In: Center for the Study of Language and Information (CSLI Lecture Notes, no. 139), Stanford, CA

Lawrie D (2003) Language models for hierarchical summarization. PhD thesis, University of Massachusetts Amherst

Lawrie D, Morrell C, Feild H, Binkley D (2006) What's in a name? A study of identifiers. In: 14th International conference on program comprehension, Athens, Greece, pp 3–12

Maarek YS, Berry DM, Kaiser GE (1991) An information retrieval approach for automatically constructing software libraries. IEEE Trans Softw Eng 17(8):800–813

Marcus A, Maletic J (2001) Identification of high-level concept clones in source code. In: Proceedings of automated software engineering, San Diego, CA, November

Marcus A, Sergeyev A, Rajlich V, Maletic J (2004) An information retrieval approach to concept location in source code. In: IEEE working conference on reverse engineering, Delft, The Netherlands, November

McMahon JG, Smith JF (1998) A review of statistical language processing techniques. Artif Intell Rev 12(5):347–391

Moonen Leon (2001) Generating robust parsers using island grammars. In: Proceedings of the 8th working conference on reverse engineering, October 2001. IEEE Computer Society Press, Los Alamitos, CA, pp 13–22

Morrell C, Pearson J, Brant L (1997) Linear transformation of linear mixed effects models. Am Stat 51:338–343

Ratiu D, Deissenboeck F (2006) Programs are knowledge bases. In: 14th IEEE International Conference on Program Comprehension, pp 79–83, 14-16 June 2006

Rilling J, Klemola T (2003) Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In: Proceedings of the 11th IEEE international workshop on program comprehension, Portland, OR, May

Sjøberg D, Hannay J, Hansen O, Kampenes V, Karahasanovic A, Liborg N, Rekdal A (1993) A survey of controlled experiments in software engineering. IEEE Trans Softw Eng 19(4)

Sneed H (1996) Object-oriented cobol recycling. In: Proceedings of 3rd IEEE working conference on reverse engineering, Monterey, CA. IEEE Computer Soceity, Los Alamitos, CA, pp 169–178

Takang A, Grubb P, Macredie R (1996) The effects of comments and identifier names on program comprehensibility: an experiential study. J Program Lang 4(3):143–167

Verbeke G, Molenberghs G (2001) Linear mixed models for longitudinal data, 2nd edn. Springer, Berlin Heidelberg New York

Dawn Lawrie received her A.B. from Dartmouth College with High Honors in Computer Science and an M.S. and Ph.D. from the University of Massachusetts, Amherst. After earning her degree, Dr. Lawrie joined the faculty of Loyola College in Maryland in 2003 where she currently an assistant professor. Her research interests include the organization of information and applying information retrieval techniques to software engineering.

Henry Feild is an undergraduate studying at Loyola College in Maryland. A major in Computer Science, Henry has been a research assistant for Loyola College Computer Science faculty for three years. His research interests include natural language processing and information retrieval.

Dave Binkley is a Professor of Computer Science at Loyola College in Maryland. He received his doctorate from the University of Wisconsin in 1991 at which time he joined the faculty at Loyola.

From 1993 to 2000 Dr. Binkley worked as a faculty research at the National Institute of Standards and Technology (NIST) where he become interested in empirical software engineering while studying the factors involved in technology transfer. In 2006 he co-chaired the program at the International Conference on Software maintenance. His current research interests include semantics-based software engineering tools, the application of information retrieval techniques in software engineering, and improved techniques for program slicing.