Concept Vocabularies in Programmer Sociolects (Work in Progress)

J. E. Rice[1], B. Ellert[2], I. Genee[3], F. Täiani[4], and P. Rayson[5]

[1] Dept. of Math and Computer Science, University of Lethbridge, Canada j.rice@uleth.ca
[2] School of Computing Science, Simon Fraser University, Canada bellert@sfu.ca
[3] Dept. of Modern Languages, University of Lethbridge, Canada inge.genee@uleth.ca
[4] IRISA, Université de Rennes 1, France francois.taiani@irisa.fr
[5] School of Computing and Communications, UCREL, Lancaster University, UK p.rayson@lancaster.ac.uk

Abstract. The code a programmer writes plays a key role in communicating the intent and purpose of that code. However little is known about how this process is influenced by sociological factors. Does a programmer's background, experience, or even gender affect how they write computer programs? Understanding this may offer valuable information to software developers and educators. In this initial phase of research we focus on experience and writing, while upcoming phases will incorporate reading and comprehension. In this paper we discuss an early experiment looking at how years of programming experience might influence identifier formation.

1 Introduction

Large software engineering projects require many individuals to collaborate on the same code, and ideally everyone involved should have an equally clear understanding of the code. Unfortunately this can be problematic [1]. Both natural language communication, e.g. during the design phase or in accompanying documentation, and artificial language communication, e.g. communication through programming languages, must be considered, particularly as project complexity increases. Increased project complexity also means that up-to-date documentation may not be available, and so code may be the primary communication tool. Clearly it is important to know what factors might impact a programmer's coding choices. The question of how people use artificial (programming) languages is very broad, and so we have begun with a small examination of how people choose identifiers as they are writing programs, particularly in the choice of identifier names when compared to the known vocabulary of words surrounding a particular project. Subsequent phases will broaden this investigation to include broader aspects of language use, and as well examine how varying choices can impact comprehension.

2 Background

2.1 Variation in Programming

Although a programmer is not permitted to break the syntax rules when developing a program, he or she is still allowed significant variability including the choice of several equivalent operators (or combinations) for carrying out computations; the method or approach for breaking down the problem; and the choice of names for identifiers. We are interested in all of these variations; however at this early stage of the research we have restricted ourselves to examining identifier naming conventions.

Unlike categorical approaches to natural language studies, the comparatively new field of sociolinguistics examines linguistic variability. Sociolinguistic studies have shown that sociological variables such as gender, age, socioeconomic status, and register (the relative formality of the situation) systematically correlate with linguistic variables, e.g. [2]. Sociolinguists examine what circumstances affect the production of different and equally viable methods of expression by speakers belonging to different social groups. When such variability becomes ingrained in a specific speech community, this gives rise to various "lects" such as dialects, ethnolects, sociolects, genderlects, and idiolects.

When looking at linguistic variables, computer code provides much less room for variation than natural language. Possibly because of this identifier names act as a method of annotation for the coder. For the parser and compiler, the choice of name is irrelevant as long as it is unique. Where the name choice matters is in the documentary structure to help explain the code to the reader.

2.2 Connections with Quality & Comprehension

As a project matures it generally becomes more complex as varying authors contribute more concepts and possibly more styles of expression [3]. By investigating how styles vary between people, it may be possible to introduce standards to reduce this variation, thus improving comprehensibility. For instance [4] has suggested that using descriptive and consistent identifier names can improve comprehension.

Overall program quality is harder to accurately define than is comprehensibility. Although both have levels of subjectivity, comprehensibility can be entirely defined based on perception. If the majority agrees that a program is comprehensible then it is comprehensible. On the other hand, program quality has a certain amount of objective fact that is difficult to pinpoint. One way to approximate this is by counting the number of software bugs. High "program quality" (low bug count) has been correlated with high "identifier name quality" (use of dictionary-based words, consistent capitalization, concise and consistent naming, appropriate lengths of names, and no type encoding) [5].

These findings can also be used to help improve code as it is being written. E.g. tools have been developed to assist in choosing identifier names based on a set of consistent identifier naming conventions. As reported in [6] programmers were tasked with coming up with names without assistance that were subsequently checked against the recommendations. The participants were rarely able to come up with acceptable names without assistance, where "acceptable" considered important factors such as consistency.

2.3 Defining the Word

Current studies in identifier naming conventions focus on linking them to their sources in natural language, e.g. [7]. These previous studies do not, however, use cues such as camelcase or underscores to break up the identifier names. This should be done to avoid problems that come about when coders write something as "one word" when it should be "two words" (e.g. dataset). Of course a large problem lies in determining an accurate definition of what constitutes a word.

Linguistic theory avoids the notion of a word being just a string of characters, the boundaries of which are defined by whitespace and punctuation [8]. One reason for this avoidance is that this definition breaks down when looking at languages with different conventions, and even within English. In addition trying to force identifier names to fit arbitrary English spacing conventions results in the loss of interesting distinctions. This problem was also addressed in [9]. Instead, providing a comparison between identifiers in programs and those used in the libraries referenced by those programs allows us to determine what is common and standard. In this way domain-specific abbreviations that cannot easily be linked to natural language correlates are not viewed as problem cases or bad practice, but rather an integral part of how programmers choose to name their identifiers. Just as the speech community shapes natural language, the programming community shapes artificial language.

3 Methodology

The goal of this study was to investigate what words individuals chose as identifiers. We hoped to discern patterns in these choices that could be linked to sociological variables. This might tell us something about how different groups used this aspect of programming languages.

Since the scope of this study was narrowed to identifier names, the approach from [10] provided a nice basis for our methodology, which is centered around shifting the paradigm of studying identifier names away from its reliance on the natural languages on which they are based. By treating this as a mode of communication separate from *both* computer code and natural language we hope to obtain a clearer picture of an individual's naming "style" and prevent losing distinctions between extremely similar identifiers.

3.1 Extracting and Parsing Identifiers

Figure 1 (a) presents the pseudocode for extracting identifier names from C++ code. Lines 1–5 reduce the code to identifiers, reserved keywords, and whitespace. Line 6 removes all duplicates and whitespace. Line 7 leaves just the identifiers, and lastly, the identifiers are output in the format required by the parsing script.

(a)

```
1 remove comments
2 remove string literals
3 remove character literals
4 remove preprocessor directives
5 remove non-alphanumeric characters
6 put tokens into set
7 remove reserved keywords
8 output identifiers
```

(b)

```
1 put library concepts into set
2 put sample concepts into set
3 take set difference
4 output set sizes
```

Fig. 1. (a) Extracting identifier names. (b) Comparing concept sets.

This output is then fed into the scripts utilized in [10]. Class names are parsed into "concepts" (or words), where the parsing is performed solely on the assumption of camelcasing and underscoring conventions. That is, camelcasing and/or underscoring is assumed to indicate that there are multiple concepts of interest within the identifier, and so it is broken up accordingly. By letting the programmer's demarcation of identifiers determine the concepts instead of using external sources, a more *descriptive* rather than *prescriptive* analysis can be taken. We believe that how a coder chooses to break up an identifier speaks to how the concepts are viewed. As the concepts are extracted from the list of identifiers, a cumulative sum representing the number of unique identifiers each concept occurs in is recorded. This basic process can be used to evaluate a coder's choices in identifier naming by seeing how many concepts are drawn from source libraries.

3.2 Comparing Concept Vocabularies

To see how each sample compares to each other, a "standard" must be defined as a base point. By using the domain-specific libraries that all the samples from within a particular group reference a common ground can be found. The same procedure for extracting identifiers and parsing them into concepts was run on the library code to form a basis for a standard vocabulary. Figure 1 (b) presents the pseudocode for comparing an individual's concepts to the standard. Lines 1 and 2 import the concept lists to be compared into sets. Lines 3 and 4 output how many concepts are in the sample that are not in the library. A simple set difference removes concepts from the sample that are in the library and the number remaining gives this result.

4 Results

From a fourth year image processing class at the University of Lethbridge, 40 samples of C++ code were drawn from 14 students, listed in Table 1. Each contributed three files, except for participant 13 who only contributed one. All were working on the same set of assignments using

the library Netpbm[1].The Netpbm source files were concatenated to provide one standard source to draw concepts from. While giving consent, each student undertook a short sociological survey asking for name, gender, first spoken language (NL1), first programming language (AL1), years of programming experience, whether the assignment was group work or not, and whether the code was planned to be reused or not.

Table 1. Participants.

PID	Gender	NL1	AL1	Experience	Reuse
1	M	English	BASIC	5 years	Y
2	F	English	C++	2 years	N
3	M	Chinese	C	10 years	Y
4	M	English	TI-BASIC	5 years	N
5	M	Hungarian	Pascal	7 years	N
6	M	English	C++	2 years	N
7	M	English	C++	2.5 years	N
8	M	English	Java	4 years	Y
9	M	English	C++	5 years	N
10	M	English	C	4 years	Y
11	M	English	ACS	5 years	N
12	M	English	C++	1.5 years	N
13	M	English	C++	4 years	Y
14	M	English	Java	7 years	N

For each sample, the percentage of library-external concepts was calculated as shown in Table 2. As expected, the majority of the concepts used were also used in the library. Figure 2 shows a plot of programming experience against percent of out-of-library concepts. The size of each bubble represents the total number of concepts. Samples from the same participant are grouped by colour, so each circle represents one participant file. A linear regression produces an r value of -0.126256 and an R-squared value of 0.015941. The negative r value means that any potential correlation is negative. This means that as programming experience increases, less out-of-library concepts are used, which is what should be expected. The low R-squared value

Table 2. Samples.

PID	File	Concepts	External	Percent	Experience
12	1	47	4	8.51%	1.5 years
12	2	62	6	9.68%	1.5 years
12	3	46	6	13.04%	1.5 years
2	1	44	4	9.09%	2 years
2	2	43	5	11.63%	2 years
2	3	60	10	16.67%	2 years
6	1	51	5	9.80%	2 years
6	2	48	7	14.58%	2 years
6	3	63	10	15.87%	2 years
7	1	50	4	8.00%	2.5 years
7	2	48	4	8.33%	2.5 years
7	3	42	4	9.52%	2.5 years
8	1	58	4	6.90%	4 years
8	2	46	4	8.70%	4 years
8	3	52	13	25.00%	4 years
10	1	62	5	8.06%	4 years
10	2	54	5	9.26%	4 years
10	3	71	11	15.49%	4 years
13	1	50	5	10.00%	4 years
1	1	68	8	11.76%	5 years

PID	File	Concepts	External	Percent	Experience
1	2	56	8	14.29%	5 years
1	3	51	9	17.65%	5 years
4	1	53	3	5.66%	5 years
4	2	48	6	12.50%	5 years
4	3	60	9	15.00%	5 years
9	1	45	4	8.89%	5 years
9	2	55	6	10.91%	5 years
9	3	45	5	11.11%	5 years
11	1	49	2	4.08%	5 years
11	2	50	5	10.00%	5 years
11	3	50	6	12.00%	5 years
5	1	70	12	17.14%	7 years
5	2	70	15	21.43%	7 years
5	3	79	17	21.52%	7 years
14	1	62	5	8.06%	7 years
14	2	21	3	14.29%	7 years
14	3	31	5	16.13%	7 years
3	1	40	1	2.50%	10 years
3	2	39	1	2.56%	10 years
3	3	38	2	5.26%	10 years

[1] available from ftp://ftp.wustl.edu/graphics/graphics/packages/NetPBM

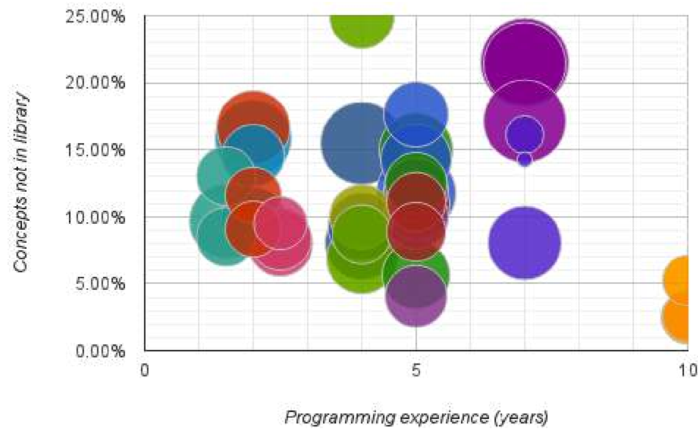

Fig. 2. Programming experience vs concept use.

means that less than 2% of the variation is explained by a linear regression. In other words, the data may as well have been random and there is no reason to say it is correlated.

5 Discussion

Although the current results are quantitatively inconclusive, there are various directions of thought that can be pursued. We are working on creating the groundwork for a basic paradigm of viewing identifier naming conventions. It is important to make a distinction between an individual's concept vocabulary and the natural language vocabulary on which it is based. Making this distinction allows questions such as whether the concept vocabularly is influenced in a similar way as the natural language vocabulary, for instance by one's mother tongue.

Metrics We are currently using multiple samples from each individual; however it may be better to concatenate each individual's code into one large sample beforehand as was done for the library. This would allow grouping of the concepts and prevent potential bias. Additionally, the number of times a concept appears in a sample has not been utilized (we use a cumulative sum representing the number of unique identifiers containing each concept). This statistic could lead to some interesting results by giving weight to each sample's concept use instead of just doing a simple count. This might give insight into a programmer's thought processes, as this could reflect the (perceived) importance of particular concepts. Finally, instead of computing a set difference versus the standard, each individual could be compared to each other or to the entire population to generate another metric for similarity. This, combined with interviews, could determine whether our existing metric could be used to predict how easily another individual (or group) might understand code from a different individual (or group).

Qualitative Analysis It might also be useful to qualitatively examine the samples, given that we have a relatively small sample set at this point. Examination by hand could lead to insights as to how identifiers are being created, and analysis as simple as outputing the concepts in the set created after taking the set difference would allow us to identify what specific out-of-library concepts individuals are using. Following this, checking whether there are any trends between the samples may lead to some interesting results. By grouping individuals together based on their identifier naming style, it may be possible to develop teaching approaches for more directed instruction. Another possible approach could be to look at the specific in-library concepts. For example, we could look at the top ten library concepts reused and see if this correlates to any sociological variables.

Data Collection More data is required before we can draw any conclusions from quantitative analysis approaches. Student data collection has proven to be more difficult than anticipated;

however using online repositories (e.g. open source) leads to the problem of determining the sociological data. We currently have restricted our samples to code from projects that are related, i.e. work based on a class assignment, and this was done in order to ensure that the same libraries were used in all the samples in order that we could use those libraries as a standard. However it might be interesting to lift this restriction, as long as some basis for a standard identifier naming source could be identified.

The initial aim of this study was to investigate the effects of gender on coding, in the hope that this may lead to a better understanding of the underrepresentation of women in the field. Unfortunately the underrepresentation prevented the collection of enough data from female participants to draw any statistically significant conclusions, and this is a problem we are still struggling with.

6 Conclusions

This paper presents our initial work in examining identifier naming styles, with a goal towards linking this to sociological variables. We believe that understanding how groups within the programming society use artificial language to communicate will provide interesting and enlightening information with applications in software engineering, software quality, and computer science education. In other fields researchers are "... committed to examining the way language contributes to social reproduction and social change" [11], and have shown that people "... actively choose ways of framing to accomplish specific ends within particular interaction. These choices are drawn, in part, from sociocultural norms..." [12]. Given the reliance that the world now has on computer programs, we hope that this study may offer a step towards similar research focusing on artificial languages. Our preliminary study was quantitatively inconclusive; however we have offered a methodology to continue with this work, and several avenues for other directions that we hope to pursue.

References

1. R. N. Charette. Why software fails [software failure]. *IEEE Spectr.*, 42(9):42–49, September 2005.
2. W. Labov. *Principles of Linguistic Change, Cognitive and Cultural Factors.* Language in Society. Wiley, 2011.
3. A. Mohan, N. Gold, and P. Layzell. An initial approach to assessing program comprehensibility using spatial complexity, number of concepts and typographical style. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 246–255, 2004.
4. S. Blinman and A. Cockburn. Program comprehension: investigating the effects of naming style and documentation. In *Proceedings of the Sixth Australasian User Interface Conference (Vol. 40)*, AUIC '05, pages 73–78, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
5. S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CMSR)*, pages 156–165, Washington, DC, USA, 2010. IEEE Computer Society.
6. P. A. Relf. Tool assisted identifier naming for improved software readability: an empirical study. In *Proceedings of the 2005 International Symposium on Empirical Software Engineering*, November 2005.
7. Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. Amap: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, page 79–88, New York, NY, USA, 05/2008 2008. ACM, ACM.
8. Dee Gardner. Validating the construct of word in applied corpus-based vocabulary research: A critical survey. *Applied Linguistics*, 28(2):241–265, 2007.
9. D. P. Delorey, C. D. Knutson, and M. Davies. Mining programming language vocabularies from source code. In *21st Annual Psychology of Programming Interest Group Conference - PPIG*, 2009.
10. François Taiani, Jackie Rice, and Paul Rayson. What is middleware made of?: exploring abstractions, concepts, and class names in modern middleware. In *Proceedings of the 11th International Workshop on Adaptive and Reflective Middleware*, ARM '12, pages 6:1–6:6, New York, NY, USA, 2012. ACM.
11. Mary M. Talbott. *Language and Gender.* Polity Press, 1998.
12. S. Kendall and D. Tannen. Gender and language in the workplace. In R. Wodak, editor, *Gender and Discourse.* Sage, London, 1997.