Automatic Derivation of Concepts based on the Analysis of Source Code Identifiers

Latifa Guerrouj
DGIGL - SOCCER Lab, Ptidej Team
Ecole Polytechnique de Montréal, Québec, Canada
Email: latifa.guerrouj@polymtl.ca

***Abstract*—The existing software engineering literature has empirically shown that a proper choice of identifiers influences software understandability and maintainability. Indeed, identifiers are developers' main up-to-date source of information and guide their cognitive processes during program understanding when the high-level documentation is scarce or outdated and when the source code is not sufficiently commented.**

Deriving domain terms from identifiers using high-level and domain concepts is not an easy task when naming conventions (e.g., Camel Case) are not used or strictly followed and-or when these words have been abbreviated or otherwise transformed. Our thesis is to develop an approach that overcomes the shortcomings of the existing approaches and maps identifiers to domain concepts even in the absence of naming conventions and-or the presence of abbreviations.

Our approach uses a thesaurus of words and abbreviations to map terms or transformed words composing identifiers to dictionary words. It relies on an oracle that we manually build for the validation of our results. To evaluate our technique, we apply it to derive concepts from identifiers of different systems and open source projects. We also enrich it by the use of domain knowledge and context-aware dictionaries to analyze how sensitive are its performances to the use of contextual information and specialized knowledge.

***Keywords*-Identifier Splitting, Program Comprehension, Linguistic Analysis, Software Quality.**

I. Research Context: Program Comprehension and Software Quality

The software engineering literature reports evidence on the usefulness of source code identifiers to enhance software quality, program comprehension and program understandability. Many researchers [1], [2], [3], [4], [5] have shown the crucial role of identifers in these areas. Indeed, some research works have studied the usefulness of identifiers to recover traceability links [6], [7], measure conceptual cohesion and coupling [8], [9], and, in general, as an asset that can highly affect source code understandability and maintainability [10], [11], [12].

Researchers have also studied the quality of source code comments and the use of comments and identifiers by developers during their understanding and maintenance activities [12], [13], [14]. They all concluded that identifiers can be useful if carefully chosen to reflect the semantics and the role of the named entities they are intended to label Stemming from Deißenböck and Pizka observation on the relevance of terms in identifiers to drive program comprehension, almost all previous works attempted to map identifiers to concepts by splitting them into component terms and words to guide the cognitive process using these identifier fragments.

To the best of our knowledge, two families of approaches exist to segment compound identifiers: the simplest one assumes the use of the Camel Case naming convention or the presence of an explicit separator. The more complex strategy is implemented by the *Samurai* tool and it relies on a lexicon and uses greedy algorithms to identify component words [5].

However, the above mentioned approaches have limitations. First, they are not always able to associate identifier substrings to words or terms, *e.g.*, domain-specific terms or English words, which could be useful to understand the extent to which the source code terms reflect terms in high-level artifacts [15]. Second, they do not deal with word transformations, *e.g.*, abbreviation of *pointer* into *pntr*.

II. Question

The research question of our thesis can be stated as follows:

How to build a contextual approach able to automatically derive domain terms and thus concepts based on the analysis of source code identifiers even in the absence of naming conventions and/or the presence of truncated/abbreviated words?

III. Answer

To answer our question, we propose an approach that assumes that programmers create identifiers by applying a set of transformation rules on terms/words. For example, to create an identifier for a variable that counts the number of defects in a program, the two words *number* and *defects* can be concatenated with or without an underscore, *e.g.*, *error_defect* or *defectnumber* or following the Camel Case, *e.g.*, *defectNumber*. Contractions of one or both words are also possible leading to identifiers such as *defectNbr*, *nbr_defect*, *nbrOfDefect*, or *dfct_nbr*.

The approach uses the string-edit distance between terms and words to quantify how close are words, representing concepts, to such terms and, thus, provide a measure of the likelihood that the terms refer to some words. To deal with abbreviations, our technique uses a thesaurus of words and abbreviations and applies word transformation rules in the context of a hill-climbing search. But, a thesaurus is not enough when several possible mappings are available. It is why, we propose the use of n-gram language models as a semantic concept predictor. N-Grams are traditionally used in large vocabulary speech recognition systems to provide the recognizer with an a-priori likelihood P(W) of a given word sequence W.

In a nutshell, our approach relies on input dictionaries and a distance function to segment (if necessary) simple and composed identifiers and associate the resulting terms with words in the dictionaries, even if the terms are truncated/abbreviated. Dictionaries may include English words and–or technical words, *e.g.*, *microprocessor* and *database* (in the computer domain), or known acronyms, *e.g.*, *afaik* (in the Internet jargon).

IV. Methodology

To answer our research question, we propose and follow a methodology where the tree main phases are detailed below. Details about the accomplishment of our previous work and how our technique components have been developed are reported in [16].

1. **Building a thesaurus:** To map terms or transformed words composing identifiers to dictionary words, we build a thesaurus of words and abbreviations. One possibility to build such a thesaurus would be to merge different specific or generic dictionaries, such as those of spell checkers, *e.g.*, i-spell which contains about 35,000 words, or of upper ontologies, *e.g.*, WordNet, which contains about 90,000 entries. Yet, to reduce the computation time, we build smaller dictionaries, *e.g.*, dictionaries containing the most frequently-used English words only as well as specialized dictionaries containing acronyms and known abbreviations.
2. **Building an oracle:** To validate our approach, we need an oracle. This means that for each identifier, we will have a list of terms obtained after splitting it and, wherever needed, expending contracted words. The oracle is produced as follows: (i) a splitting of each sampled identifier, and expanded abbreviations is produced independently (ii) for all cases where there is a different splitting/expansion, a discussion meeting is held and a consensus is reached. The manual analysis is performed relying on various sources of information of the projects, ranging from source code comments to user manuals.
3. **Validation:** To evaluate our approach, we apply it to derive concepts from identifiers of different systems and open source projects. In fact, we first apply our technique to JHotDraw and Lynx using an English dictionary composed of 3,000 words. Then, we apply it to a set of 1,026 C identifiers randomly extracted from a corpus of 340 open source programs using a single English dictionary of 2,600 words and also various dictionaries: a dictionary of about 2,800 words, the previous dictionary augmented with domain knowledge, i.e., about 700 domain terms, acronyms, and well-known abbreviations, a full English dictionary of 175,000 words, contextual dictionaries, i.e., dictionaries built using terms from the same function, file, or program, and contextual dictionaries augmented with domain knowledge. Our empirical study compares the three approaches on their splitting correctness (with respect to the oracle), and on their precision, recall, and FMeasure.

V. On-going Work and Forecast Completion

At this time of our thesis, we achieved four phases of our project: we developed the key components of our approach, and achieved the tree phases of our methodology. Our on-going work consists on addressing the research questions that have not been addressed yet, expanding the evaluation of our approach to others systems and open source projects. We are also studying the possibility of building a general oracle for identifer splitting. We plan to improve the current performances of our technique in term of execution time and focus on the effect of contextual information and specialized knowledge on it to analyze how sensitive are its performances to the use of such information. Finally, we wish to conduct more experimental studies for the validation and the generalization of our results.

VI. Related Work

The crucial role of identifiers in program understanding, traceability recovery, feature and concept location motivates the large body of relevant work. In the following, we focus on the most relevant contributions to identifier splitting.

Takang *et al.* [10] empirically showed that commented programs are more understandable than non-commented programs and programs containing full-word identifiers are more understandable than those with abbreviated identifiers. Similar results have also been achieved by Lawrie *et al.* [11], [12], who suggest that the identification of words composing identifiers, and, thus, of the domain concepts associated with them could contribute to a better comprehension.

Binkley *et al.* [18] found that the Camel Case convention led to better understanding than underscores and, when subjects are properly trained, that subjects performed faster with identifiers built using the Camel Case convention rather than with underscores.

Other work [2], [19] have investigated the information carried by the terms composing identifiers, their syntactic structure and quality. An in-depth analysis of the internal structure of identifier was conducted by Caprile and Tonella [1]. They reported that identifiers are chosen to convey

Table I
MAIN RESEARCH QUESTIONS TO ADDRESS DURING OUR PROJECT

Research Questions	Answers
What is the percentage of identifiers correctly split by the proposed approach?	Overall, about 96% of JHotDraw identifiers and 93% of Lynx identifiers were correctly segmented by our approach with zero distance [16].
How does the proposed approach perform compared with the Camel Case splitter when applied to JHotDraw and Lynx?	Fisher's exact test indicates that the proposed approach performs better than Camel Case splitter on both systems and significantly better on Lynx [16].
What percentage of identifiers containing word abbreviations is the approach able to map to dictionary words when applied to JHotDraw and Lynx?	44% and 70% of JHotDraw and Lynx identifiers containing abbreviations are correctly split by our technique into component words [16].
How does our approach compare with alternative approaches, Camel Case splitting and Samurai, when C identifiers must be split?	With the simple English dictionary, our approach performs worse than the alternative approaches. However, our technique outperforms other approaches when the simple English dictionary is augmented with domain knowledge or, with even better results, when a program-level contextual dictionary augmented with domain knowledge is used [17].
How sensitive are the performances of approach to the use of contextual information and specialized knowledge in different dictionaries?	Two factors contribute to the increase of performance of our approach: augmenting the dictionary with domain knowledge, using a program-level contextual dictionary, or augmenting a program-level dictionary with domain knowledge to obtain the best performances [17].
What percentage of identifiers containing word abbreviations is our approach able to map to dictionary words when applied to a set of 1,026 C identifiers randomly extracted from a corpus of 340 open source programs?	Out of the 73 abbreviations that our technique could potentially map to dictionary words, our approach produces a correct mapping for 35 of them, achieving an accuracy of 48%. Although this percentage does not look very high, to the best of our knowledge, our approach is the first and only approach able to deal with abbreviations [17].
How could we enhance the accuracy of our results in term of finding the appropriate candidate (concept) represented by a given identifer?	To increase the accuracy of our results in term of choosing the domain terms that correspond to a given identifier, we plan to develop a new word transformation rules that mimic the developers cognitive processes when building identifiers. We also want to develop a variant of our algorithm in which these transformation rules will be applied according to a determined priority instead of being randomly-chosen.
How could we overcome the non-determinism presented by our approach in the way word transformation rules are applied and in the way in which the candidate words to be transformed are selected?	This fact suggests the need for improving the heuristic to select the candidate word to be used in splitting an identifier.
How could we improve the current performances of our approach in term of execution time?	The string-edit distance used by our technique has a cubic complexity in the number of characters in the identifier (say M), words in the dictionary (say T), and maximum number of characters composing dictionary words (say N). For each word in the dictionary, we must compute as many distances as there are cells to fill the distance matrix, with a complexity of $\mathcal{O}(M \times N)$. Since there are T dictionary words, the overall complexity is $\mathcal{O}(T \times M \times N)$. A remarkable increase of performance can be achieved by saving the first edit-distance computation and, in the context of the hill climbing, recomputing only cells where the distance improves.

relevant information about the role and properties of the program entities that they label.

Guidelines for the production of high-quality identifiers have been provided by Deißenböck *et al.* [3]. Methods related to identifier refactoring were proposed by Caprile and Tonella [2] and Demeyer *et al.* [20].

Some studies [6], [7] reported the use of identifiers to recover traceability links. Also, textual similarity between methods within a class, or among methods belonging to different classes, has been used to define new measures of cohesion and coupling such as the Conceptual Cohesion of Classes proposed by Marcus *et al.* [21], which bring information complementary to structural cohesion and coupling measures.

Many commonalities can be found with previous work using the Camel Case convention to split identifiers and, in particular, with the work of Enslen *et al.* [5]. We share with these previous works the goal of automatically splitting identifiers into component terms. However, we do assume the use of neither Camel Case conventions nor a set of known prefixes or suffixes. In addition, our approach automatically generates a thesaurus of abbreviations using transformation rules attempting to mimic the developers' cognitive processes when building identifiers.

VII. CONCLUSION

Our initial work gave birth to TIDIER: An automatic tool that derives domain terms based on the analysis of source code identifiers. TIDIER, by mapping identifiers to concepts, provides hints that could help developers comprehend programs during their understanding and maintenance activities.

The power of DTW in showing how close a given identifier term is to a dictionary word reveals how well the concepts associated to the dictionary words are conveyed by the identifier. Reported results show that with program-level dictionaries augmented with domain knowledge, i.e., common acronyms, abbreviations, and C library functions, our approach significantly outperforms the previous approaches.

Future works will be devoted to the development of new word transformation rules that mimic the cognitive processes of developers when composing identifiers with abbreviated forms, and to the introduction of enhanced heuristics for term selection and word transformations. we also plan to improve the string-edit distance guiding our approach, to speed up the algorithm by reducing the search space, and to use semantic information. In fact, one of the main limitations of our technique is its pure lexical-level matching and its cubic complexity. Finally, we plan to combine our approach with Samurai by Enslen et *et al.* [5] to benefit from their respective advantages while reducing their limitations.

Acknowledgments

I am deeply grateful to my supervisors for their guidance and support.

References

[1] B. Caprile and P. Tonella, "Nomen est omen: Analyzing the language of function identifiers," in *Proc. of the Working Conference on Reverse Engineering (WCRE)*, Atlanta Georgia USA, October 1999, pp. 112–122.

[2] ——, "Restructuring program identifier names," in *Proc. of the International Conference on Software Maintenance (ICSM)*, 2000, pp. 97–107.

[3] F. Deissenbock and M. Pizka, "Concise and consistent naming," in *Proc. of the International Workshop on Program Comprehension (IWPC)*, May 2005.

[4] D. Lawrie, H. Feild, and D. Binkley, "Syntactic identifier conciseness and consistency," in *Sixth IEEE International Workshop on Source Code Analysis and Manipulation Philadelphia Pennsylvania USA*, Sept 27-29 2006, pp. 139–148.

[5] E. Enslen, E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009, Vancouver, BC, Canada, May 16-17, 2009*, 2009, pp. 71–80.

[6] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 970–983, Oct 2002.

[7] J. I. Maletic, G. Antoniol, J. Cleland-Huang, and J. H. Hayes, "3rd international workshop on traceability in emerging forms of software engineering (TEFSE2005)." in *ASE*, 2005, p. 462.

[8] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[9] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of 22nd IEEE International Conference on Software Maintenance*. Philadelphia Pennsylvania USA: IEEE CS Press, 2006, pp. 469 – 478.

[10] A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experiential study," *Journal of Program Languages*, vol. 4, no. 3, pp. 143–167, 1996.

[11] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007.

[12] ——, "What's in a name? a study of identifiers," in *Proceedings of 14th IEEE International Conference on Program Comprehension*. Athens, Greece: IEEE CS Press, 2006, pp. 3–12.

[13] B. Fluri, M. Wursch, and H. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007, pp. 70–79.

[14] Z. M. Jiang and A. E. Hassan, "Examining the evolution of code comments in postgresql," in *Proceedings of the 2006 International Workshop on Mining Software Repositories MSR 2006*, 2006, pp. 179–180.

[15] A. D. Lucia, M. Di Penta, R. Oliveto, and F. Zurolo, "Improving comprehensibility of source code via traceability information: a controlled experiment," in *Proceedings of 14th IEEE International Conference on Program Comprehension*. Athens Greece: IEEE CS Press, 2006, pp. 317–326.

[16] N. Madani, L. Guerrouj, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "Recognizing words from source code identifiers using speech recognition techniques," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010), March 15-18 2010, Madrid, Spain*. IEEE CS Press, 2010.

[17] L. Guerrouj, D. P. Massimiliano, A. Giuliano, and Y.-G. Guéhéneuc, "Tidier: An identifier splitting approach using speech recognition techniques," *Journal of Software Maintenance and Evolution: Research and Practice*, May 2010 (submitted for publication).

[18] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under_score," in *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*. IEEE Computer Society, 2009, pp. 158–167.

[19] E. Merlo, I. McAdam, and R. D. Mori, "Feed-forward and recurrent neural networks for source code informal information analysis," *Journal of Software Maintenance*, vol. 15, no. 4, pp. 205–244, 2003.

[20] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems Languages and Applications*. ACM Press, 2000, pp. 166–177.

[21] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 287–300, 2008.