The Journal of Systems and Software 100 (2015) 117–128

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

From source code identifiers to natural language terms

Nuno Ramos Carvalho[a,*], José João Almeida[a], Pedro Rangel Henriques[a], Maria João Varanda[b]

[a] Department of Informatics, University of Minho, Campus de Gualtar, 4710-057 Braga, Portugal
[b] Polytechnic Institute of Bragança, Campus de Santa Apolónia, 5300-253 Bragança, Portugal

ARTICLE INFO

Article history:
Received 2 October 2013
Received in revised form 29 July 2014
Accepted 9 October 2014
Available online 31 October 2014

Keywords:
Program comprehension
Natural language processing
Identifier splitting

ABSTRACT

Program comprehension techniques often explore program identifiers, to infer knowledge about programs. The relevance of source code identifiers as one relevant source of information about programs is already established in the literature, as well as their direct impact on future comprehension tasks.

Most programming languages enforce some constrains on identifiers strings (e.g., white spaces or commas are not allowed). Also, programmers often use word combinations and abbreviations, to devise strings that represent single, or multiple, domain concepts in order to increase programming linguistic efficiency (convey more semantics writing less). These strings do not always use explicit marks to distinguish the terms used (e.g., CamelCase or underscores), so techniques often referred as *hard splitting* are not enough.

This paper introduces Lingua::IdSplitter a dictionary based algorithm for splitting and expanding strings that compose multi-term identifiers. It explores the use of general programming and abbreviations dictionaries, but also a custom dictionary automatically generated from software natural language content, prone to include application domain terms and specific abbreviations. This approach was applied to two software packages, written in C, achieving a f-measure of around 90% for correctly splitting and expanding identifiers. A comparison with current state-of-the-art approaches is also presented.

1. Introduction

Understanding source code is a requirement for software maintenance and evolution tasks (Von Mayrhauser and Vans, 1995; Corbi, 1989). Software reverse engineering, is a process that aims to infer how a program works by analyzing and inspecting its building blocks and how they interact to achieve their intended purpose (Nelson, n.d.; Chikofsky and Cross, 1990). Many of these techniques rely on mappings between human oriented concepts and program elements (Rajlich and Wilde, 2002). Identifiers are one of the major source of information about program elements (Caprile and Tonella, 1999, 2000), and their meaningfulness has a direct impact on future comprehension tasks (Lawrie et al., 2006). Today, most of the programming communities promote the use of best practices and coding standards, that usually include rules and naming conventions which tend to improve the quality of identifiers used (e.g., the style guide for the Python programming language[1]).

Program identifiers have been greatly explored in the context of program understanding: for concept and concern location (see, e.g., Shepherd et al., 2007; Marcus et al., 2004; Abebe and Tonella, 2010; Liu et al., 2007), relating documentation with source code (see, e.g., Antoniol et al., 2002; Yadla et al., 2005; Marcus and Maletic, 2003), and other assorted software analysis applications (see, e.g., Lawrie et al., 2007; Lawrie and Binkley, 2011; Enslen et al., 2009; Carvalho et al., 2012, 2014). All this work can benefit from better program identifiers handling, and in many cases results can be improved (Dit et al., 2011).

Programming languages grammars constrain the strings that can be used as identifiers, not allowing spaces and other special characters (e.g., commas). These also tend to be short and easy to remember. Thus, acronyms and abbreviations are frequently used to represent real world concepts. The major goal of the work described in this paper, and related work (see Section 2), is to

* Corresponding author. Tel.: +351 253 604 430.
E-mail addresses: narcarvalho@di.uminho.pt (N.R. Carvalho), jj@di.uminho.pt (J.J. Almeida), prh@di.uminho.pt (P.R. Henriques), mjoao@ipb.pt (M.J. Varanda).

[1] Available from: http://legacy.python.org/dev/peps/pep-0008/ (Last accessed: 31-03-2014).

http://dx.doi.org/10.1016/j.jss.2014.10.013
0164-1212/

promote strings used as identifiers in the program domain, to sets of terms representing concepts in the application domain.

Identifiers created using a single word (or abbreviation) are easier to relate with domain terms. The real challenge are compound identifiers, i.e., identifiers assembled using more than one string (each representing a term), because these strings need to be correctly isolated before they can be linked with domain concepts. Moreover, these strings can be abbreviations or acronyms, and not actual words, increasing the tokenization process difficulty. Sometimes an explicit mark is used to delimit the strings used, for example, the identifier "insert_user" uses the underscore as an explicit mark to clearly distinguish the word "insert" and the word "user". Another common explicit technique is the CamelCase notation, for example in the identifier "insertUserData" the words used are explicitly delimited with an uppercase letter. This trend of explicit word compounds are referred in the literature as *hard splits* (or *hard words*). Many times no explicit mark is used to delimit the words, for example the identifier "timesort", was formed by joining the words "time" and "sort", but there is no explicit mark where one word ends, and the next word begins. This is usually referred as *soft splits* (or *soft words*). Splitting *soft words* is more complex that *hard words*, and the complexity increases when acronyms or abbreviations are used instead of complete words (Lawrie et al., 2006, 2006, 2007).

This paper introduces Lingua::IdSplitter (henceforth abbreviated LIdS),[2] a simple and fast algorithm that addresses the problem of splitting *soft words*, and can cope with abbreviations, acronyms, or any type of linguistic short-cuts (for example, use only the first letter of a word). The algorithm calculates a ranked list of all the possible splits for an identifier, based on a set of dictionaries, and the top entry in the rank is proposed as the correct split. Besides the actual split, the result includes the set of full terms that compose the identifier, in case abbreviations were used for example. This technique can use an arbitrary set of dictionaries, but one of the major advantages of this approach is the use of a software specific dictionary computed automatically from the documentation corpus – built automatically and specific to each software package – using a combination of Natural Language Processing (NLP) techniques. This dictionary enables the algorithm to correctly handle identifiers splitting using arbitrary abbreviations or combinations of terms specific to the application domain, not prone to be present in more general programming dictionaries.

To validate this approach, the technique was applied to sets of identifiers extracted from two open source projects written in C, using a heterogeneous combination of techniques for multi-word identifiers. The calculated sets of splits were compared with the manual split (traditionally called the *oracle*) and the overall splitting accuracy for several different settings was above 80%. To compare LIdS's performance against other state-of-the-art approaches, LIdS was also applied to other case studies available in the literature, in order to compare the achieved results.

The remainder of this paper is organized as follows: Section 2 discusses some related work; Section 3 describes the proposed approach to split and expand identifiers; Section 4 describes in detail the experimental study done to validate LIdS effectiveness; Section 5 relates and compares this work with state-of-the-art techniques that address the same problem; and finally, Section 6 presents some closing remarks and trends for future work.

[2] LIdS is available under GNU General Public License in the official comprehensive Perl network (CPAN) from: http://search.cpan.org/dist/Lingua-IdSplitter/ (Last accessed: 09-07-2014).

2. Related work

The work by Caprile and Tonella (1999), describes their lexical, syntactical and semantic analysis of function identifiers. In this work the creation of a dictionary based on information extracted from the software (source code mainly) was also a concern, and a valuable source of information. It also helps to highlight the relevance of NLP techniques applied in the context of Program Comprehension.

Enslen et al. (2009) describe Samurai, an automatic approach to split identifiers that uses a scoring function based on program-specific and global frequency tables. These tables are built by mining strings frequency in source code. The main intuition behind this algorithm is that sub-strings used as part of an identifier are likely to be used in other identifier from the same software, or even in other programs. A similar concern is behind our proposed custom corpus-based dictionaries, the expressions and terms found in natural language text belonging to the software domain are prone to be used as identifiers.

TIDIER (Madani et al., 2010; Guerrouj et al., 2011) is another approach for identifiers splitting. This algorithm is based in the Dynamic Time Warping algorithm, initially devised to compute distances in the context of speech recognition. And tries to achieve the correct split by computing distances between the identifier and words found in a set of dictionaries. This algorithm shares some concerns with LIdS, namely: (1) the use of dictionaries, including domain specific dictionaries, (2) the inference of abbreviations is based on computing some kind of metric between the identifier and words found in dictionaries. A possible short-coming of this approach (and the previous one – Samurai) is that both can produce a different split for the same identifier in different iterations. TIDIER also does not handle splitting identifiers that contain single letter abbreviations (e.g., *"gchord"*). LIdS given the same input, and the same set of dictionaries, always computes the same split/expansion.

TRIS (Guerrouj et al., 2012) is a more recent technique for splitting and expanding program identifiers proposed by the same authors of TIDIER. It also uses a set of dictionaries, general and domain specific. TRIS handles the splitting and expansion as an optimization problem, divided in two stages. During the first stage a set of dictionary word transformations is created including corresponding costs, and during the second phase the goal is to find the optimal path in the expansion graph. The resulting split and expansion corresponds to the one with the minimal cost.

The GenTest normalization algorithm proposed by Lawrie et al. (2010) and Lawrie and Binkley (2011) involves vocabulary normalization found in software artifacts (e.g., source code, documentation) to improve Information Retrieval software analysis tools. This algorithm starts by scoring all the possible splits, and the resulting split is the one with the highest score. The scoring function is based in a set of metrics, based on internal information (e.g., word characteristics), and external information (e.g., dictionaries).

LINSEN is an approach for splitting identifiers, and expanding abbreviations, proposed by Corazza et al. (2012). The authors propose the use of the Baeza-Yates and Perleberg, an approximate string matching technique, and the use of several general and domain specific dictionaries, to find a mapping between program identifiers and the corresponding set of dictionary words.

The work by Sureka (2012), is a more recent approach for splitting identifiers using the Yahoo web search and image search similarity distance. The main idea is that strings used as identifiers represent concepts in real life, and documents indexed in search engines include images and text, providing information to compute possible splits scores.

Butler et al. (2011) describe the INTT algorithm, a technique for identifiers names automatic tokenization, with special focus on