Acta Univ. Sapientiae, Informatica, **2,** 1 (2010) 40–50

Automatic derivation of domain terms and concept location based on the analysis of the identifiers

Peter Václavík
Technical University of Košice
Faculty of Electrical Engineering and Informatics
Department of Computers and Informatics
email: Peter.Vaclavik@tuke.sk

Jaroslav Porubän
Technical University of Košice
Faculty of Electrical Engineering and Informatics
Department of Computers and Informatics
email: Jaroslav.Poruban@tuke.sk

Marek Mezei
Technical University of Košice
Faculty of Electrical Engineering and Informatics
Department of Computers and Informatics
email: marekmezei@gmail.com

Abstract. Developers express the meaning of the domain ideas in specifically selected identifiers and comments that form the target implemented code. Software maintenance requires knowledge and understanding of the encoded ideas. This paper presents a way how to create automatically domain vocabulary. Knowledge of domain vocabulary supports the comprehension of a specific domain for later code maintenance or evolution. We present experiments conducted in two selected domains: application servers and web frameworks. Knowledge of domain terms enables easy localization of chunks of code that belong to a certain term. We consider these chunks of code as "concepts" and their placement in the code as "concept location". Application developers may also benefit from the obtained domain terms. These terms are parts of speech that characterize a certain concept. Concepts are encoded in "classes" (OO paradigm) and

Computing Classification System 1998: D.2.8
Mathematics Subject Classification 2010: 68N99
Key words and phrases: program comprehension, domain knowledge, program quality, software measurement

the obtained vocabulary of terms supports the selection and the comprehension of the class' appropriate identifiers. We measured the following software products with our tool: JBoss, JOnAS, GlassFish, Tapestry, Google Web Toolkit and Echo2.

1 Introduction

Program comprehension is an essential part of software evolution and software maintenance: software that is not comprehended cannot be changed [5, 6, 7, 8].

Among the earliest results are the two classic theories of program comprehension, called *top-down* and *bottom-up* theories [9]. Bottom-up theory: Consider that understanding a program is obtained from source code reading and then mentally chunking or grouping the statements or control structures into higher abstract level, i.e. from bottom up. Such information is further aggregated until high-level abstraction of the program is obtained. Chunks are described as code fragments in programs. Available literature shows chunks to be used during the bottom-up approach of software comprehension. Chunks vary in size. Several chunks can be combined into larger chunks [1]. On the other hand, the top-down approach starts the comprehension process with a hypothesis concerning a high-level abstraction, which then will be further refined, leading to a hierarchical comprehension structure. The understanding of the program is developed from the confirmation or refutation of hypotheses.

An important task in program comprehension is to understand where and how the relevant concepts are located in the code. Concept location is the starting point for the desired program change. Concept location means a process where we assume that programmer understands the concept of the program domain, but does not know where is it located within the code. All domain concepts should map onto one or more fragments of the code. In other words, process of concept location is the process that finds that code-fragment [5].

Developers who are new to a project know little about the identifiers or comments in the source code, but it is likely that they have some knowledge about the problem domain of the software. In this paper, we present a new way of program comprehension that is based on naming of identifiers. When trying to understand the source code of a software system, developers usually start by locating familiar concepts in the source code. Keyword search is one of the most popular methods for this kind of task, but the success is strictly tied to the quality of the user queries and the words used to construct the identifiers and comments.

We present a way how to create a domain vocabulary automatically as a

result of source code analysis. We classify the parts of speech and measure their occurrence in the source code.

2 Motivation

Domain level knowledge is important when programmers attempt to understand a program. Programmer inspects source code structure that is directed by identifiers. The quality and the "orthogonality" of the identifiers in the source code affects the time of program comprehension. Next kinds of quality could be measured:

1. percentage of *fullword* identifiers,
2. percentage of *abbreviations* and *unrecognized* identifiers,
3. percentage of *domain terms* identified in the application.

Percentage of full word identifiers is very important in the case of absence of documentation. The first two qualities could be derived directly from the source code toward common vocabulary. We don't need any additional domain data source to get relevant results. The third quality is not derived directly. We need to make measurements in order to obtain domain vocabulary.

Usually we don't have domain terms of the analysed software product. The question is: *How can we create the vocabulary of terms for a particular domain?* In this paper we propose a way to derive it automatically.

Nowadays, the companies are affected by employee fluctuation, especially in the IT sector. Each company has ongoing projects in the phase of developing or maintenance. New developer participating in the project has to understand project to solve the assigned task. Domain terms are usually in the specification. The transition from specification to implementation is bound usually to the transformation of terms. For example, if the specification contains word **car**, that word could be changed to word **vehicle** in the implementation phase. In spite of the fact that the word **vehicle** is a hypernym of the word **car**, we cannot find the word "vehicle" by brute force through searching by keywords. That is the reason for looking for some statistical evidence that car is a vehicle. It means that there exists "gap" in the meanings between the words used in specification and implementation. Our goal is to eliminate partly "this kind" of gap.

Developers of new software products may put another question: *What kind of parts of speech is usually used for a particular category of identifiers?* We can measure it directly from the source code. We can also find, if the rules are domain specific or generally applicable.

3 Methodology of programm inspection

Full word identifiers provide better comprehension then single letters or abbreviations [3]. It is the reason why we want to provide a tool for measurement of this aspect of program quality. We use the WordNet database of words to identify the potential domain terms.

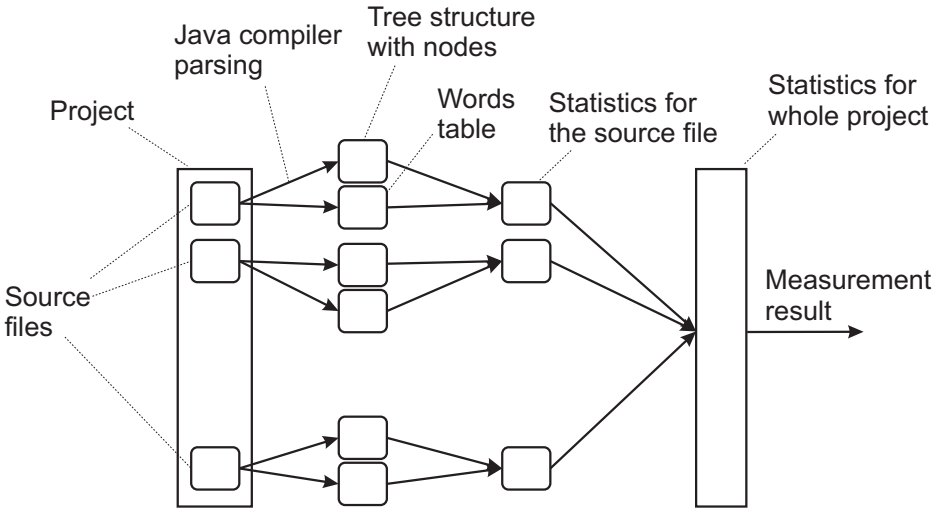

Figure 1: The methodology of program inspection

We apply our tool to well-known open-source projects. They belong to two domains: domain of application servers and domain of web frameworks. Each project consists of a set of source code files. We examine every source file separately. Based on information we have got by source files analysis we make measurements for the whole project. Our measurements follow the scheme shown on the Fig. 1.

1. First, we parse every source file using Java compiler. We build a tree structure of nodes. Each node belongs to one of the next types: *class*, *method*, *method parameter* or *class variable*. Then we process the names of each identified node. Name processing consists of splitting the name according to common naming conventions. For example, "setValue" is split into "set" and "value" words. After then we put all identified words into a table.
2. As a second step, we produce statistics for the source file. We examine which word belongs to class variable, method parameter, method or class

and also we try to assign part of speech to the words.

After source files analysis mentioned in previous steps we produce statistics for the whole project: we build a set of words containing all words used in the source files, and also we build a set of words used in the variables (class variables and method parameters), methods and classes. The set of words used in project will represent the software vocabulary for the particular project.

The software domain vocabulary represents the intersection of all software vocabularies of all software products of the same domain. Not all identified words are suitable candidates for the inclusion into software domain vocabulary. It is expected to apply filters in a process of source code analysis. So, the reason behind filtering is to eliminate terms that are irrelevant regarding the domain. As a final result we obtain a set of words ordered by occurrence. We obtain domain vocabulary as well as potential domain vocabulary (words are not identified in all measured software products).

As was mentioned in the previous section, the categorization in accordance to the parts of speech is expected in the experiment. It induces another problem: one word can belong to more parts of speech (e.g. "good" is adjective as well as noun). WordNet provides help in disambiguation and classification of words.

4 Experiments based on word analysis

WordNet provides a database of the most used words in the parts of speech. As was mentioned earlier, we have developed a tool to measure results in the graph, tabular and textual form. The tool's input is the project's source code. To present it we decide to inspect software products of two application domains:

- Java EE application server,
- Web framework.

We have selected next Java EE application servers:

- JOnAS 4.10.3 (http://jonas.ow2.org/),
- JBoss 5.0.1.GA (http://www.jboss.org/jbossas),
- GlassFish Server v2.1 (https://glassfish.dev.java.net/).

and web frameworks:

- Google Web Toolkit 1.5.3 (GWT) (http://code.google.com/intl/sk/webtoolkit/),

- Echo2 v2.1 (http://echo.nextapp.com/site/echo2),
- Tapestry 5.0.18 (http://tapestry.apache.org/).

	Glass.	**JOnAS**	**JBoss**	**Echo2**	**GWT**	**Tap.**
Number of source files	10553	3611	6448	402	593	1707
Number of words	10229	4502	5055	1044	2584	2154
Number of recognized words	4297 (42%)	2140 (48%)	2932 (58%)	903 (86%)	1582 (61%)	1714 (80%)
Number of not recognized words	5932 (58%)	2362 (52%)	2123 (42%)	141 (14%)	1002 (39%)	440 (20%)
Number of nouns	2361 (55%)	1311 (61%)	1687 (58%)	537 (60%)	839 (53%)	913 (54%)
Number of verbs	1259 (29%)	542 (25%)	842 (29%)	229 (25%)	484 (31%)	526 (30%)
Number of adjectives	549 (13%)	235 (11%)	330 (11%)	117 (13%)	213 (13%)	226 (13%)
Number of adverbs	128 (3%)	52 (3%)	73 (2%)	20 (2%)	46 (3%)	49 (3%)

Table 1: The number of recognized domain-terms for application servers and web frameworks

Table 1 summarizes data for the selected products. Other kind of results obtained from our tool in tabular form gives us information about identified words that are parts of software vocabulary. Now, each domain has three sets of software vocabulary. In our measurement we have selected the 50 most used words of each software vocabulary for further analysis. Their intersection is a set of terms belonging to the domain vocabulary. Table 2 presents the most used words and their occurrence. Words identified as domain terms are emphasized with bold letters. Potential domain terms recognized in two software products are emphasized with italic. Words that belong to only one software vocabulary are typed ordinary. Thanks to WordNet we can also identify semantically similar (synonyms, homonyms, hypernyms, and so on) words as a domain or potential domain term.

Glass.	JBoss	JOnAS	GWT	Echo2	Tap.
name (28727)	**name** (11774)	**name** (6950)	*type* (1774)	action (801)	**name** (1653)
value (9067)	**test** (7332)	**ejb** (2570)	**name** (1168)	*property* (506)	**value** (1140)
type (7550)	**id** (3169)	**test** (2179)	*method* (540)	**value** (475)	*type* (970)
class (6953)	**bean** (3154)	**id** (1602)	**class** (466)	*test* (395)	*class* (936)
object (5277)	**ejb** (2962)	*home* (1437)	*logger* (353)	*component* (369)	field (798)
id (4266)	**value** (2885)	**server** (1379)	**value** (344)	*element* (347)	page (778)
key (4641)	**type** (2473)	**type** (1251)	info (254)	**id** (331)	*component* (776)

Table 2: Application server and web framework domain terms recognition

5 An experiment on concept location

Within the next step we locate concepts encoded in keywords of the product. We use again WordNet for searching keywords. Programmers knows only the domain the software product it belongs to. They do not need to use exact words used in source code.

We present here an example of concept location. Lets suppose that somebody wants to change the algorithm for determining the parts of speech in our program. S/he needs to locate the concept of determining the parts of speech in the source code of the examined program. It is known that programmers and maintainers use different words to describe essentially the same or similar concepts [5]. Therefore the use of full-text search for concept location is very limited. We will try to find concepts based on semantic search.

In our example we assume that a concept is the identifier of a method or a class. We want to find a fragment in the source code where the parts of speech are located. We will try to find this code fragment based on this key-phrase: "*find word form*". For every keyword in our key-phrase we will make a database of related words – words that are in some semantic relationship to the keyword. Then, we will try to locate code fragment in our source code, where at least 1 occurrence for every keyword is found. This process is shown

on the Fig. 2. However we are not looking only for the keywords itself, but also for semantically related words. In our example, as a result we find a method with this definition:

```
public String getType(String word) {
  //Method source code
}
```

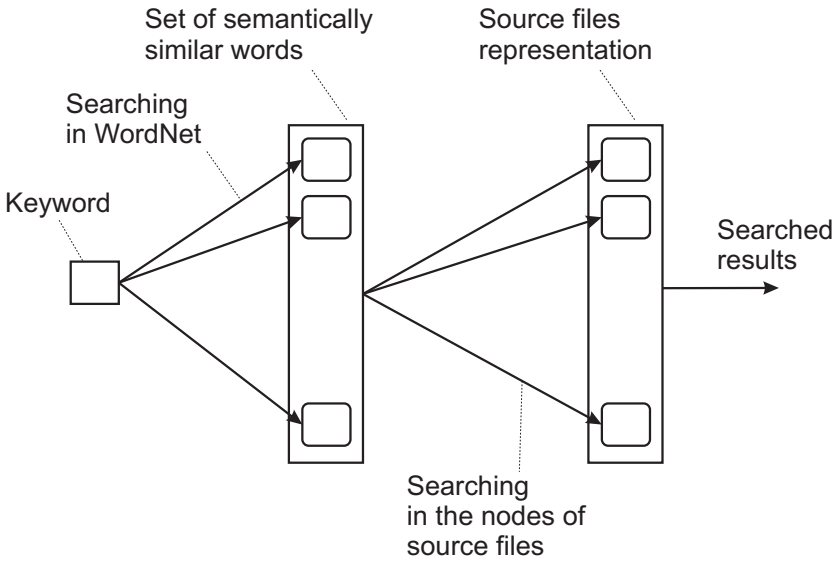

Figure 2: Process of concept location

We found the three keywords in this method definition based on these semantic relations:

1. *find–get*: "get" is a hypernym of "find". We found the word "get" in the method name.
2. *word*: we found the term "word" itself in the parameter name.
3. *form–type*: "type" is a hyponym of "form". We found the word "type" in the method name.

We can see on this example that we could not locate this concept easily using fulltext search, but we can locate it using search based on semantic relations.

6 Related and further steps

The study of software vocabularies. This study is focused on three research questions: (1) to what degree relate terms found in the source code to a particular domain?; (2) which is the preponderant source of domain terms: identifiers or comments?; and (3) to what degree are domain terms shared between several systems from the same domain? Within the studied software, we found that in average: 42% of the domain terms were used in the source code; 23% of the domain terms used in the source code are present in comments only, whereas only 11% in the identifiers alone, and there is a 63% agreement in the use of domain terms between any two software systems. They manually selected the most common concepts, based on several books and online sources. They chose 135 domain concepts. From the same resources, for each of these concepts one or more terms and standard abbreviations that describe the concept were manually selected and included in the domain vocabulary [2].

Our aim was to define the domain vocabulary automatically. Results from the experiments will be used to build domain vocabularies for other domains too. These vocabularies support more detailed automatic classification of software products. Our next experiments will include inspection of comments in the source code. This stream of research is strongly promoted by [2, 10, 11].

Concept location. One of the experiments in the area of mapping between source code and conceptualizations shared as ontology has been published in [4]. The programs regard themselves as knowledge bases built on the programs' identifiers and their relations implied by the programming language. This approach extracts concepts from code by mapping the identifiers and the relations between them to ontology. As a result, they explicitly link the sources with the semantics contained in ontology. This approach is demonstrated using on the one hand the relations within Java programs generated by the type and the module systems and on the other hand the WordNet ontology.

We are locating concepts by keywords specified by programmers. Concept location is based on searching names in the identifiers that are in some relation to the specified keywords. This approach supports easier understanding of higher-level abstractions within the inspected application. We will work further on the concept visualization as well as on concept location refinement issues.

7 Conclusions

We can conclude the experiment results briefly as follows:

- In general, the most used parts of speech for all inspected element types are nouns (57%).
- Application servers as well as GWT use a lot of not recognized words due to different identifiers.
- The most number of recognized words is used in Tapestry (80%) and Echo2 (86%) web frameworks. We can assume that the source code of both products could be well understandable.
- From the comprehension point of view the application servers are more complex than web frameworks.
- In spite of application servers' complexity, they are using more common domain terms. Application server domain vocabulary consists of other well-known terms like: "context", "session", "service", and so on.
- Concept location gives us opportunity to find source code fragments more efficiently and with better results than using classical keyword search.

Acknowledgement

This work was supported by VEGA Grant No. 1/4073/07 - Aspect-oriented Evolution of Complex Software System and by APVV Grant No. SK-CZ-0095-07 - Cooperation in Design and Implementation of Language Systems.

References

[1] Ch. Aschwanden, M. Crosby, Code scanning patterns in program comprehension, *Proc. 39th Hawaii International Conference on System Sciences (HICSS)*, Hawaii, 2006. ⇒41

[2] S. Haiduc, A. Marcus, On the use of domain terms in source code, *Proc. 16th IEEE International Conference on Program Comprehension*, 2008, pp. 113–122. ⇒48

[3] D. Lawrie, C. Morrell, H. Feild, D. Binkley, What's in a name? A study of identifiers, *Proc. 14th IEEE International Conference on Program Comprehension*, Athens, Greece, June 2006, pp. 3–12. ⇒43

[4] D. Ratiu, F. Deissenböck, Programs are knowledge bases, *Proc. 14th IEEE International Conference on Program Comprehension*, 2006, pp. 79–83. ⇒48

[5] V. Rajlich, N. Wilde, The role of concepts in program comprehension, *Proc. 10th International Workshop on Program Comprehension*, 2002, pp. 271–278. ⇒41, 46

[6] L. Samuelis, Notes on the emerging science of software evolution, in: *Handbook of Research on Modern Systems Analysis and Design Technologies and Applications*, Hershey: Information Science Reference, 2008, pp. 161–167. ⇒41

[7] L. Samuelis, Cs. Szabó, On the role of the incrementality principle in software evolution, *Egyptian Comput. Sci. J.*, **29,** 2 (2007) 107–112. ⇒41

[8] Cs. Szabó, L. Samuelis, Notes on the software evolution within test plans, *Acta Electrotechnica et Inform.*, **8,** 2 (2008) 56–63. ⇒41

[9] M. A. Storey, Theories, methods and tools in program comprehension: past, present and future, *Proc. 13th International Workshop on Program Comprehension*, 2005, pp. 181–191. ⇒41

[10] B. L. Vinz, L. H. Etzkorn, A synergistic approach to program comprehension, *Proc. 14th IEEE international Conference on Program Comprehension*, 2006, pp. 69–73. ⇒48

[11] B. L. Vinz, L. H. Etzkorn, Improving program comprehension by combining code understanding with comment understanding, *Knowledge-Based Syst.*, **21,** 8 (2008) 813–825. ⇒48

Received: August 30, 2009 • Revised February 25, 2010