Mapping Terms in Application and Implementation Domains

Oleksandr Panchenko, Hasso Plattner, Alexander Zeier

Hasso Plattner Institute, P.O. Box 900460, 14440 Potsdam, Germany

{panchenko, office-plattner, zeier}@hpi.uni-potsdam.de

Abstract

One problem of reengineering is the gap between terms from the application domain and terms from the implementation domain. Particularly, it is observable while searching source code: in search queries maintainers often use terms from the application domain that differ from terms in the implementation domain that have been selected by original developers. Therefore, many queries fail. This paper proposes the utilization of code structure for automatic mapping application domain terms and implementation domain terms. In this way, implementation domain terms can be tagged by the identified terms from the application domain. Such redistribution of existing terms in the search index allows matching even those relevant documents which do not contain exact terms from the query. The feasibility of the proposed method was estimated by collecting statistics in an open-source project.

1 Introduction

Source code search relies on the quality of terms that original developers have used to name classes, methods, parameters, variables, and other identifiers. Since those terms often are short, not descriptive enough, and differ from standard English words, many search queries fail. While the maintainer can guess about the meaning of the identifier, search tools can match terms (identifiers) only if these were explicitly included in the search query. The root of this challenge is that original programmers can select arbitrary terms for identifiers. While experienced maintainers, familiar with the source code, feel comfortable using implementation domain terms, novice maintainers prefer using terms from the application domain [1]. Therefore queries should be first mapped to the terms from the implementation domain. Thus, search queries with application domain terms should be able to match relevant documents despite the fact that these documents do not necessarily contain these terms. One possible solution is to artificially extend the query before the execution [2]. Another possibility is to enrich documents by related terms from the application domain while indexing. The paper proposes using the structure of code to find relevant terms and to add these additional terms to the index.

2 Code Structure for Terms Redistribution

Consider the example given in the Figure 1. The maintainer would like to find the functionality for tracking promotional discounts that are based on the code of shipment preference. First he tried to find it using the query "tracking discount". However, a large number of hits were returned. Therefore, the maintainer tried to refine the query to "tracking discount shipment". In this case no results were returned. The desired behavior of the search engine would be to return a reference to the class `WebController`, because the functionality responsible for the tracking order code based on shipment preferences is invoked from this class (lines 12,13) and used for discount calculation (line 16). Nevertheless, state-of-the-art methods cannot match all necessary terms in the class `WebController`. This example illustrates the importance of the *location* of the documentation. To improve recall each implementation domain term should be enriched (tagged) by corresponding terms from the application domain. In the example above, JavaDoc of the method `setTrackingNumber` could be attached to the method invocation in the class `WebController`. Therefore, the documentation can enrich scant method names with textual description and together with other methods' descriptions provide a solid lexical platform to search on it.

```
01 public class Order { ...
02 /** update order tracking code
03   * on a shipment preference */
04 public Map setTrackingNumber(Map context){
05   ...
06   } ...
07 }
08
```

```
09 public class WebController { ...
10 Order order = new Order(); ...
11
12 Map trackingNr
13      = order.setTrackingNumber(sendMap);
14 ...
15 ...
16 discount = calculateDiscount(trackingNr);
17 ...
18 }
```

Figure 1: Example listing (simplified)

The question addressed in this paper is *how to redistribute those terms so that each important implementation domain term is described by additional terms from the application domain?* This paper proposes utilizing structure of code on an example of most important operator – assignment operator. The approach proposes extending the assignment operator applied to operands to their semantic meanings: if the values of two operands are equal, their semantic meanings should be also equal.

If one part of the assignment operator is expressed in application domain terms, the terms from implementation domain in another part can be tagged with corresponding terms from the application domain. This operation can be done while indexing, and redistributed terms can be stored into an index. To identify which of those terms are from the application domain a dictionary lookup can be made: for simplicity reasons English words from a dictionary and known abbreviations are supposed to be from the application domain. In the example above, the term `trackingNr` will be enriched by [order, tracking, number]. Besides the assignment operator there is a number of other patterns where this principle applies: (1) Replicate JavaDoc in all places where the method is invoked. Thus, the variable `trackingNr` will be enriched by [update, order, tracking, number, code, shipment, preference]. If no JavaDoc is available, other elements of method signature can be used, e.g., description of a type returned by the method. (2) Build a data flow graph, select a subgraph for each variable and find all operators that influence the value of this variable. Analysis of those indirect assignment operators delivers additional terms. (3) In a method invocation there is data flow due to parameter passing. This is essentially a set of actual-to-formal assignments.

The proposed approach can efficiently handle synonyms, because even short descriptions provide additional synonyms. In the example above the description of the method `setTrackingNumber` adds several synonyms (set – update, number – code).

3 Statistics

To estimate the feasibility of the method, source code of a 30k LOC Java open source project was statistically investigated. To automate this process a prototype was implemented that uses Lucene[1] search engine. The implemented prototype enables Lucene to extract structural information from source code, to resolve assignment operators, and to write additional mapped terms to the index. All information is automatically captured during indexing and stored in the index file. After the indexing is finished, searching can be done in the usual way. Queries will return not only documents which directly match terms from the query, but also those relevant documents which contain implementation domain terms related to the application domain terms from the query.

In the previous section it was assumed that internal documentation (e.g., JavaDoc or comments) can provide a mapping to the application domain. To check the effectiveness of the embedded documentation, terms expressed in user-defined strings were compared with terms provided by existing internal comments. Before that, the implementation domain terms were automatically tokenized based on Hungarian notation. For example, identifier `TokenizeJavaMethodNames` was tokenized into [tokenize, java, method, names]. All terms, both from code and comments, were stemmed before indexing.

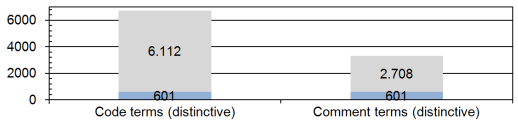

Figure 2: Vocabulary

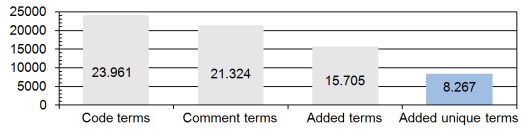

Figure 3: Number of terms added to the index

Figure 2 compares the vocabulary of implementation terms (left) with the vocabulary of terms from embedded documentation (comments and JavaDoc). Whereas some terms (601 terms) are present in both vocabularies, terms from documentation significantly complement the vocabulary of implementation terms.

The next question is *how many application domain terms can be added to the index by applying the proposed method?* Figure 3 demonstrates results of a statistical analysis. The first column represents the size of the code dictionary (class, method, parameter, and variable names). The second column represents the size of the comment dictionary (terms used in comments and embedded documentation). While Figure 2 presents the number of distinctive terms aggregated over the entire project, Figure 3 represents the sum of terms which are distinctive inside each class. The terms in the first and the second columns can be matched by the regular search. The third column represents the number of terms added while applying the method. The fourth column represents those added terms which are distinct from already existing terms inside corresponding source code document. These terms can match queries which would not be matched by the regular source code search. The manual analysis of the added terms shows that in many cases these terms provide good description and synonyms for given functionality in application domain terms.

References

[1] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Comm. of the ACM*, 37(5):72–82, 1994.

[2] H. Liu and T. C. Lethbridge. Intelligent search methods for software maintenance. *Information Systems Frontiers*, 4(4):409–423, 2002.

[1] Apache Lucene, http://lucene.apache.org, last visited on 10.04.2011