Locating Term Introductions to support Code Comprehension

Jan Nonnen, Daniel Speicher

University of Bonn
Computer Science III
Bonn, Germany

{nonnen,dsp}@cs.uni-bonn.de

Keywords: identifier analysis, name meaning, empirical evaluation, explorative approach, code comprehension

1 Introduction

For a compiler source code identifiers are simply artifacts to identify program elements like variables, fields, methods, or types. For a developer however names convey the intentions of the code, as Figure 1 illustrates. Beck emphasises in [1] the importance of intention revealing names. This judgement is founded on the fact that an identifier is written once, but read multiple times.

```
a) void transfer(Account from, to; float amount){
      from.debit(amount); to.credit(amount); }
b) void m1(T1 v1, v2; float v3){
      v1.m2(v3); v2.m3(v3); }
c) transfer from to amount
       from debit amount to credit amount
```

Figure 1: While for a compiler code snippet b) contains as much information as a), developers will hardly understand b) but get almost enough insight from the names in c).

Recently identifier analysis has become an active research area. Because of the low relevance for the compiler, identifiers offer little structure. On the other hand programs are not simply texts. Therefore the question about the appropriate algorithms and research methodology is still open.

Our work is motivated by the assumption, that the meaning of an identifier can be understood from the code in many cases, if the correct location of origin was found. This location can be either *temporal*, a specific point in time when the term was first used in a project, or *spatial*, a method or type expressing the concept. In this work we focus on spatial locations and propose a heuristic to approximate them. Six different heuristics where evaluated and combined into a single heuristic.

We first describe how identifiers are preprocessed to obtain the terms contained in the identifiers. These represent the foundation of our analysis which will then be used to define the concept of a term introduction. To find an heuristic for the concept of a term introduction, we performed an explorative study. We will present the design of this study and show some of our results.

2 Related work

Kuhn et al. [4] used Latent Semantic Indexing to cluster source code elements that contain similar vocabulary. These clusters should help to reveal the intention of the code and are considered as topics.

Høst et al. [3] defined method naming patterns and the corresponding method meaning. They mined 100 projects with a set of local behaviour patterns and used them to automatically obtain the meaning of method names.

3 Term Introduction

As a first step identifiers are chopped into a sequence of tokens, based on camel case rules. For example `createFisheyeFigures` yields the sequence (`create`, `fisheye`, `figures`). Then each token was normalized to the linguistic canonical form, which we refer to as **term**. For example in English "run", "runs", "ran" and "running" are forms of the same term, conventionally written as "run".

Identifier:	`createFisheyeFigures`
Tokens:	(`create`, `fisheye`, `figures`)
Terms:	((`create`, v), (`fisheye`, n), (`figure`, np))
Sorted:	((`create`, v), (`figure`, np), (`fisheye`, n))

Figure 2: Preprocessing of the identifiers. We use the part of speech tags v=verb, n=noun, and np=noun in plural.

Later a copy of each term sequence is sorted by **linguistic dominance**. A word a dominates word b, if b renders a more precisely, i.e. b depends on a. In our example: What is happening here? "create"! What is created? "figures"! What kind of figure? "fisheye"! To compute the linguistic dominance *part of speech tags* are derived for each term and the list is reordered accordingly. The sorting rules have been

developed by Falleri et al. in [2] and are described in detail as well in [5].

Finally we define that a tuple (t, c) with a type c and a term t used in c is a **term introduction**, if a meaning of t can be understood by reading the code in type c, hence c can be seen as the origin of this meaning of t.

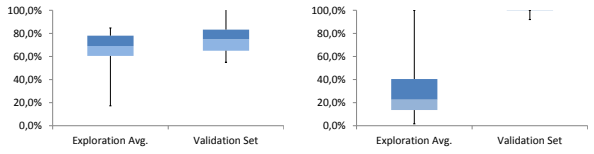
4 Study design

The main goal of the case study [5] was to find an accurate term introduction heuristic. It was designed as an explorative evaluation on 30 open source projects. Starting with a given initial set of heuristics, these were at first evaluated on 20 projects in the *exploration phase*. Afterwards, in the *validation phase*, the heuristics were fine tuned and re-validated on the remaining projects.

In total there were over 8000 samples drawn from all projects and manually classified. All samples were drawn randomly and independent, and especially positive and negative samples were drawn separately. A point estimator was used to estimate the total number of true positives and false negatives in each project. From those values, precision and recall were calculated. *Precision* is the ratio between correctly reported term introductions and the total number of reported introductions. Similar, *recall* is used to measure the effectiveness by calculating the ratio between correctly reported term introductions and the total number of correct introductions.

5 Heuristic and Results

The derived heuristic collects all atomic names of public or protected members. It then collects all terms in type names that specialize a set of commonly used terms. In an subsequent iterative step, terms in type names are collected where all but this term were already collected. From all collected locations, those are removed that depend on other collected locations.



(a) Precision (b) Recall

Figure 3: The left box plot of (a) and (b) shows the precision and recall of the exploration phase; the right box plot of the validation phase.

From the exploration phase we had learned to ignore tokens like “i”, “abstract”, “j”, “default”, and “impl” if they redundantly repeat static information. Furthermore we considered more instructive locations like properties, enumeration constants and ignored anonymous types or methods that just throw exceptions or do nothing.

The final introduction heuristic performed well in the validation phase and resolved many issues with the heuristics of the exploration phase. Figure 3 shows how recall and precision increased. The median of the precision was 75% and the median of the recall 100%.

6 Conclusion and Future Work

We conjecture that the increase in recall which in turn slowed down the increase in precision was due to the iterative step of our heuristic. Further research should verify this or find the real reasons.

A user study could help to improve the heuristics further and to understand the influence of different points of view and programming experience of developers. This could further lead to a better understanding of the underlying mental model as well as the intentions of a name. We plan to integrate deeper reflections of the code comprehension process.

Pennington [6] observed, that programmers at first develop a control flow abstraction of the program during code comprehension. A mental model is created by combining code fragments into bigger text structure abstractions and cross-referencing these abstractions. Our concept of term introductions supports a developer by cross-referencing terms contained in identifiers. A term introduction for a given term provides a project internal meaning, directly represented in the source code.

During evaluation situations were encountered in which not a single term, but a compound of terms had a meaning. For example in one case the term ”user” had a different meaning than the intended compound ”user-name”. An analysis of such co-occurring terms benefits precision and recall. Also many terms found were abbreviations; the current approach treats them as normal terms.

Given that our research question was hard to define strictly, our expectations had been moderate. Yet we found to have reached a good starting point for further improvements and first applications.

References

[1] K. Beck. *Implementation patterns*. Addison-Wesley Professional, 2006.

[2] J. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. Automatic Extraction of a WordNet-Like Identifier Network from Software. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 4–13. IEEE, 2010.

[3] E. Høst and B. Østvold. Debugging Method Names. In *Proc. 23rd ECOOP*, 2009.

[4] A. Kuhn, S. Ducasse, and T. Gīrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[5] J. Nonnen. Naming Consistency in Source Code Identifiers. Diploma Thesis, University of Bonn, 2011.

[6] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.