On Recommending Meaningful Names in Source and UML

Adrian Kuhn
Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch/akuhn

ABSTRACT

Meaningful method names are crucial for the readability and maintainability of software. Existing naming conventions focus on syntactic details, leaving programmers with little or no support in choosing meaningful (domain) names. In this paper we propose to build a recommendation system that supports software developers and software architects when naming identifiers in source code as well as when naming elements in UML diagrams. We discuss related work, outline the design of such a recommendation system and discuss possible evaluation strategies.

1. INTRODUCTION

Recommendations systems support software developers in their work. For a survey of the current state of the art in recommendation systems, please refer to Happel and Maalej [3]. In the survey, they found that existing approaches focus on "you might like what similar developers like" scenarios. However, they found that structured artifacts and semantically well-defined development activities bear large potentials for further recommendation scenarios.

In this paper we propose to build a recommendation system that supports software developers and software architects when naming identifiers in source code as well as when naming elements in UML diagrams. By the taxonomy of Happel *et al.* this corresponds to proposing development information about code and artifacts (*i.e.* UML diagrams and other design documents that might require support for naming). Sharing of the naming information happens proactively as source code and design documents are published. Suggesting names based on existing names would be useful to increases the readability and comprehension of software systems and thus their maintenance and reuse.

In fact we might consider the naming of elements in source code as a *folksonomy* of user-generated content. In a folksonomy elements are labeled by users with tags that do not necessarily form a complete taxonomy, both formal and informal description are mixed. For example, the set of names of all variables with a given type can be considered as tags that describe the type: instances of the `Node` class might be called `tree` and `node` but also `temp` or `reply`.

In the remainder of this paper we present related work, outline the design of a naming recommendation system and discuss possible evaluation strategies.

RSSE'10, May 4, 2010, Cape Town, South Africa.

2. RECOMMENDATIONS SYSTEM

In this section we discuss related work and outline the possible components of a recommendation system that suggests names for source code as well as UML diagrams.

Høst *et al.* present the "programmer's phrase book", a recommendation system that is trained with the correlation between method names and method bodies [5]. They train a machine learning system with the gramatical structure of method names plus 12 metrics that are extracted from the byte code of the methods bodies. The gramatical structure of method names is modeled by a tree that contains nodes, such as *e.g.* `get`-noun or `find`-adjective-noun. The approach is fully automatic and requires no human intervention.

They trained the programmer's phrase book on 100 open source projects (written in Java) and used it as a case study to find methods that do not fit the trained system, to which they refer as naming bugs. They found a rate of 0-3% naming bugs, depending on the project, *i.e.* their system recommends the right name for 97% of all methods.

The programmer's phrase book is limited in at least two ways. First, it may only recommend the structure of method names (including some common verbs and adjectives) but not the right noun for specific domain terms. Second, it may not recommend names when no method body is available, as is *e.g.* the case when architects are about to design the UML diagrams of a future software system.

Hummel *et al.* present a recommendation system for the naming of UML elements [7]. The system is part of the CodeConjurer tool which uses their Merobase code search engine. Their system recommends the possibly missing methods and attributes during the design of UML diagrams. Whenever the architect adds, removes or changes an element of the UML diagram, an automatically composed search query is sent proactively to the Merobase search engine and the results are used to suggest the names for missing elements to the UML designer.

Their system thus goes beyond the second limitation of Høst's phrase book, that is, it can recommend names in the absence of source code and method bodies; it is however limited to the recommendation of entire names since method names are not broken up into their gramatical parts.

Both above systems are limited by their global reach, *i.e.* they do not take into account the domain of the local software system, and they do not consider the lexical context of a method's location in the source code.

Holmes presents Strathcona, a proactive recommendation system for source code examples [6]. He uses the structural context of the current source location to recommend examples. His prototype recommends full-text examples. However the same kind of context matching could be used to improve the results of a naming recommendation system.

In previous work we proposed *log likelihood-ratio* as an approach to compare the naming of software components [8]. We applied log likelihood-ratio to retrieve labels that describe software components, but also to compare two or more software components and to compare two versions of the same component. The same approach could be used to sort the results of a naming recommendation system by their likelihood with regard to the local context.

Latent semantic indexing (LSI) is used in information retrieval to reduce synonymy and polysemy in large text corpora [2]. Latent semantic indexing applies singular-value decomposition, a kind of eigenvalue factorization, on the term-document matrix of a text corpus. It has been applied in software engineering to recover traceability links, detect high-level clones, to measure coupling and cohesion, and to cluster software for reverse engineering (see Poshyvanyk and Marcus for a better coverage of related work [10]). Latent semantic indexing could be used to improve the results of a naming recommendation by offering the user to choose from synonymous names, but also by suggesting to replace two or more similar terms with one common name.

Hill *et al.* proposed to extract natural language phrases from source code [4]. They categorize these phrases into a hierarchy and use them to provide users with context-specific search results, however the same approach might be to improve naming recommendations with the current context. In fact, any code search approach might possibly be turned into a proactive background search that can be used to recommend names during development.

3. EVALUATION

In this section we discuss possible evaluation strategies of recommendations systems that recommend names. The evaluation of naming recommendations is limited by the subjective nature of naming. Naming conventions typically cover syntactic details only (which can be checked and measured mechanically) but do not provide an objective measured for the correct choice of domain terms.

It has been suggested to evaluate recommendations systems by replaying recorded IDE sessions [11], or by querying the code base for pairs of questions and answers [1]. For example, for each name in the source code we could extract a query with its type and context (but not its name) and then expect the name as correct answer. This evaluation can be fully automated and is useful to compare different recommendation algorithms, however it does not tell us about the user experience of the recommendation tool and thus not about its possible acceptance by industry and developers.

Similar limitations apply to qualitative approaches that manually evaluate the result of applying the tool to some selected tasks, see Bruch *et al.* [1] for a discussion.

In recent years, controlled experiments have become popular in computer science. Controlled experiments have been developed in social science as a formal tool to study the behavior of humans in a controlled setting. Their usefulness in computer science has been questioned as an "academic exercise" [12] for good reason. Software development is a complex activity and it is hard to identify and control *all* parameters that are not to be studied. Even if the study is performed with all scientific rigor, the obtained results are often not generalizable to an industry setting. Please refer to Segal for a full discussion [12].

User studies, on the other hand, are "informal" studies of the user experience that are run to learn about the reactions of users [9]. They are typically used as feedback for further iteration of the tool and to assess the usefulness of its application in industry. Hence, user studies seem like a good start to evaluate a naming recommendation systems. Unfortunately, it is not the type of study that is typically valued in academic software engineering research. According to Andrew Ko, the field still believes, for the most part, that the only study you can trust is a controlled experiments. In a recent interview[1], he thus called for more acceptance of user studies in software engineering research.

Acknowledgments.

We thank Nikolaus Schwarz for his corrections and feedback on this paper. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010).

4. REFERENCES

[1] M. Bruch, T. Schäfer, and M. Mezini. On evaluating recommender systems for api usages. In *RSSE'08*, pages 16–20, 2008. ACM.

[2] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[3] H. J. Happel and W. Maalej. Potentials and challenges of recommendation systems for software development. In *RSSE '08*, pages 11–15, 2008. ACM.

[4] E. Hill, L. Pollock, and K. V. Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *ICSE '09*, pages 232–242, 2009. IEEE.

[5] E. W. Hoest and B. M. OEstvold. Debugging method names. In ECOOP'09, LNCS, page 0–0. Springer, 2009.

[6] R. Holmes. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transaction Software Engineering*, 32(12):952–970, 2006.

[7] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEE Software*, 25(5):45–52, 2008.

[8] A. Kuhn. Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code. In *MSR '09*, pages 175–178. IEEE, 2009.

[9] F. Nielson and H. R. Nielsen. From CML to process algebra. In *CONCUR '93*, volume 715 of *LNCS*, pages 493–508. Springer-Verlag, 1993.

[10] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *ICPC '07*, pages 37–48, 2007. IEEE.

[11] R. Robbes. On the evaluation of recommender systems with recorded interactions. In SUITE '09, pages 45–48, 2009.

[12] J. Segal. The nature of evidence in empirical software engineering. In *STEP '0*, pages 40–47, Washington, DC, USA, 2003. IEEE.

[1] `http://andyjko.com/2009/09/29/emerson-murphy-hll-interviews-me-part-1`