Toward Understanding How Developers Recognize Features in Source Code from Descriptions

Shinpei Hayashi
Tokyo Institute of Technology
Tokyo 152–8552, Japan
hayashi@se.cs.titech.ac.jp

Takashi Ishio
Osaka University
Osaka 565–0871, Japan
ishio@ist.osaka-u.ac.jp

Hiroshi Kazato
NTT DATA CORPORATION
Tokyo 135–8671, Japan
kazatoh@nttdata.co.jp

Tsuyoshi Oshima
NTT Software Innovation Center
Tokyo 180–8585, Japan
oshima.tsuyoshi@lab.ntt.co.jp

ABSTRACT

A basic clue of feature location available to developers is a description of a feature written in a natural language. However, a description of a feature does not clearly specify the boundary of the feature, while developers tend to locate the feature precisely by excluding marginal modules that are likely outside of the boundary. This paper addresses a question: does a clearer description of a feature enable developers to recognize the same sets of modules as relevant to the feature? Based on the conducted experiment with subjects, we conclude that different descriptions lead to a different set of modules.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering

General Terms

Experimentation, Human Factors

Keywords

Feature location, program understanding

1. INTRODUCTION

Practical software includes various features in source code. A *feature* represents a functionality that is defined by requirements and which is accessible to developers and users [2]. Since developers who know source files related to requirements can produce a software change more efficiently [4], locating a feature relevant to a change request in source code is an important step in software maintenance. Although a feature is often localized in a small number of modules, developers have to pick out modules implementing a feature from the whole source code.

AOAsia'14, November 16, 2014, Hong Kong, China
ACM 978-1-4503-1818-1/14/11
http://dx.doi.org/10.1145/2666358.2666578

A basic clue of feature location available to developers is a description of a feature written in a natural language. For example, a change request submitted to an issue tracking system (ITS) explains an expected behavior of a feature. Given a description, developers use a keyword search tool to identify methods (in case of Java program) that are likely to be relevant to the feature [6]. Because the result may include false positives, developers must validate whether the methods are actually relevant to the feature of interest or not. Even if a perfect list of methods is given, developers may falsely recognize some of them as irrelevant [1, 3].

One cause of the difficulties in feature location is that the description of a feature does not clearly specify the boundary of a feature, while developers tend to precisely locate the feature by excluding marginal methods that are likely outside of the feature [3]. Since a feature in a program is dependent on other features in the program, developers have to infer the boundary of the feature from source code and their knowledge on the structure of the program. As a result, each developer identifies a different set of methods for the same feature. If a result is dependent on a developer, developers are hard to share the result of feature location tasks.

This paper addresses a question: does a clearer description of a feature enable developers to identify the same set of modules? If a description affects the result of feature location, a clearer description, *e.g.*, that written by the developers who implemented the feature, can help other developers identify the feature implementation more accurately.

As an initial investigation, we have conducted an experiment with 18 subjects. Subjects are divided into two groups; One group of subjects located a feature in methods using a description in an ITS, while the other did the same feature using a refined description including how the feature is related to its dependent features. As a result, a different description leads to a different set of methods, although we could not show a statistical difference.

In the rest of the paper, Sections 2 and 3 describe how our experiment has been conducted and its result. Section 4 discusses the future directions based on the result of the experiment.

2. EXPERIMENTAL SETUP

We have conducted a controlled experiment on feature location using human subjects. In the experiment, given a description of a feature, subjects find *relevant* and *useful* modules (*i.e.*, Java methods) in source code and vote them. We count the votes for each module and find the difference between the distribution of the votes using the original descriptions and that using the refined descrip-

Table 1: Prepared Features.

Feature	System	Issue ID	# modules in goldset	Short description
f_1	jEdit	1747300	13	Enable customization of folding presentation
f_2	muCommander	231	6	"Skip all" for errors that occur during a file transfer operation

As suggested in the [`http://www.mucommander.com/forums/viewtopic.php?f=2&t=938` forums]: adding a "Skip all" button when an error occurs in a multiple file move / copy operation would be a nice feature to have.

(a) Original description.

muCommander has a feature that copies/moves files selected by a user. When a user tries to execute a copy, muCommander shows a dialog to specify a destination directory. Pushing the Copy button in the dialog starts a copy process. If an error occurred during the copying of a file, then an error dialog shows a message and asks the user to skip the file, retry to copy the file, or cancel the copy process.
The new feature is "Skip All." The dialog to specify a destination directory has a new check box with the caption is "Skip errors." If a user checked the box, then muCommander automatically skips a file if an error occurred, without showing a dialog. The error message dialog also has a new button "Skip all." If the button is pushed, then muCommander shows no error dialog in further errors, as "Skip errors" is checked.

(b) Refined description.

Figure 1: Example of two descriptions.

tions.

We have selected two features for the experiment from an existing change-history-based feature location benchmark [2]. The list of these features is shown in Table 1. The reason why we used them is that we have already had the necessary materials of the experiment (see below) for these features as the preparation of our previous experiment [3].

As mentioned in the last section, we have prepared feature descriptions of two types: *original* and *refined*. The original descriptions were prepared using the benchmark [2]. The descriptions are feature requests submitted in an ITS. In the benchmark, these descriptions are automatically associated with their implementation in source code. We used them as the original descriptions. Meanwhile, the refined descriptions were prepared manually. Since original descriptions are described by a requester of the feature, a gap separates the descriptions and the actually implemented features; the original descriptions may not precisely describe modules associated with them in the benchmark. For a refined description, we used a single paragraph to describe the basic behavior without the associated feature (the *base* behavior hereafter), and another paragraph to describe the target feature to be located (the *core* behavior hereafter). The refined descriptions are based on the recorded changes in the benchmark; we tried to explain all the functionalities added by the recorded changes although the original descriptions do not mention some of them.

An example of the descriptions is presented in Figure 1. The original description is taken from Feature #231 of muCommander, whereas the refined description explicitly explains both the base behavior existing in muCommander and the core behavior newly implemented by the feature.

We recruited 18 students majoring in software engineering from two universities, Osaka University and Tokyo Institute of Technology, as subjects. Subjects are assigned two tasks, each requests them to locate a feature in source code and vote relevant and useful modules (methods) in Java programs. We carefully assigned the tasks to satisfy the following two constraints; (1) Every subject experiences an original and a refined descriptions, (2) Every subject locates both of two features, and (3) Every description is examined by at least 9 subjects. For each task, every subject is given a printed document for a task and Eclipse environment arranged for browsing and executing the source code of the target system. The document included the description of a feature and a quick reference guide of Eclipse. The subjects are prohibited to access other online materials such as the ITS.

Subjects were requested to find the modules relevant to the feature and those useful for understanding the feature, and to fill a spreadsheet with the result. A module is *relevant* to the given feature if at least a single line of code in the module is necessary to execute the feature, according to the definition of goldsets in the benchmark. A module is *useful* for understanding of the given feature if it contributes the understanding of the feature but it is not relevant. Two sessions were conducted after the introduction. In each session, the subjects performed a task in a limited time, 1 hr, for avoiding weariness.

3. RESULTS

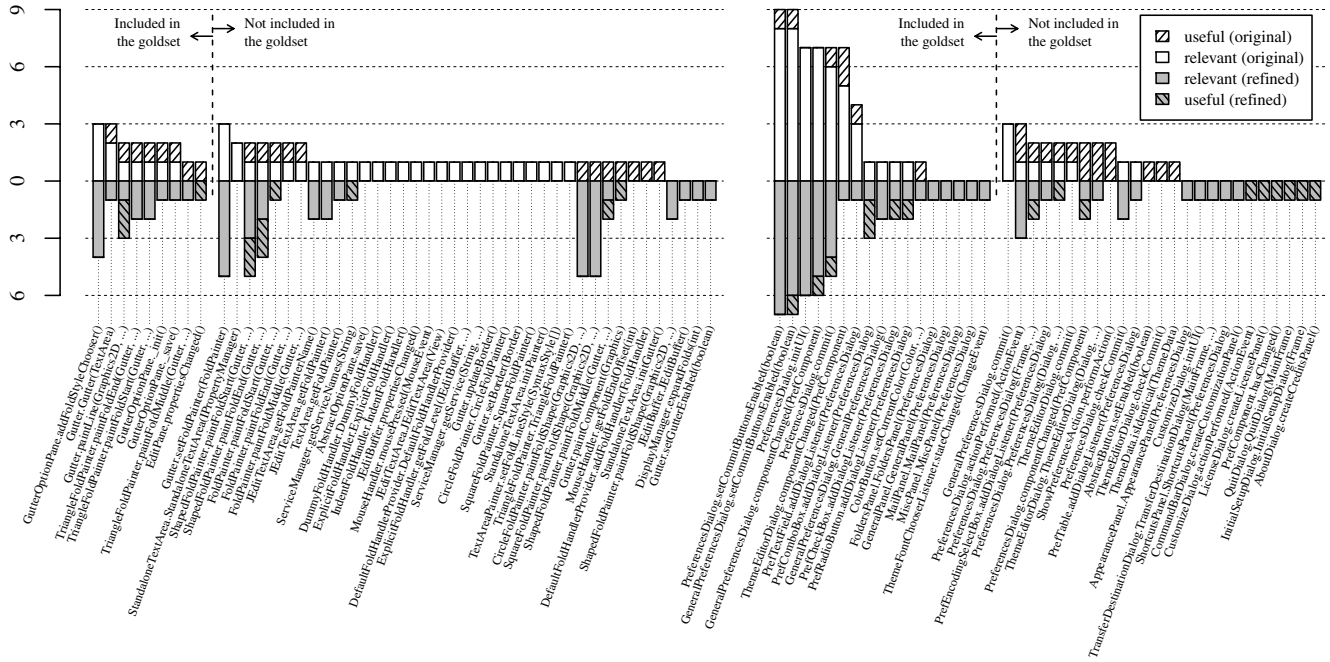
We obtained nine samples for each of original and refined descriptions for each feature. Figure 2 shows bar plots of the number of votes of the subjects according to the descriptions they used. For each plot, the modules are located on the horizontal axis, where the bars representing the number of subjects are cumulated on the vertical axis. The color and shape of the bars respectively represent the type of descriptions and votes of the subjects. Each plot has two heaps; the first heap (left) represents correct votes, *i.e.*, the modules in the first heap are the *goldset* modules defined by the benchmark, whereas the second heap (right) represents incorrect votes.

The votes were distributed over modules; many modules were voted by only one subject. This is not a surprising discovery; an empirical study by Robillard *et al.* [5] also reported that different subjects produced different concept assignments for the same concept.

Based on the result, we have observed several differences between the results of original and refined descriptions. This may be obvious, but it rather provides some important observations and insights. The following observations are mainly based on the modules voted more than two subjects:

- Some correct modules were voted by many subjects, and the others were not. For example, top two modules in Figure 2(b) have totally 16 votes, and we can regard that they are actually relevant modules for f_2. In contrast, low five correct modules were voted by only one subject. This result suggests a possibility that some modules regarded as relevant in the benchmark are actually not relevant to the target feature according to the developers' recognitions, *i.e.*, there are some incorrectness in the benchmark.

- For some modules, votes from refined descriptions are dominant rather than those from original descriptions in the first



(a) f_1: jEdit Feature #1747300.

(b) f_2: muCommander Feature #231.

Figure 2: Experimental Results.

heap. For the module `GeneralPreferencesDialog.GeneralPreferencesDialog()` as an example, which is located in the ninth in Figure 2(b), two subjects using the refined description voted as relevant although only one subject using the original description did so. This result suggests that refined descriptions are more helpful for developers than the original descriptions to recognize some aspects of the target feature accurately.

- The second heap of higher height/depth can be observed, *e.g.*, in Figure 2(a). This second heap means that many subjects judged that several modules are relevant or useful, but these modules are not actually relevant based on the goldset provided by the benchmark. This result suggests that a gap separating the benchmark built from change history and developers' judgments.

4. FURTHER DIRECTIONS

The above observations suggest the need of building another type of feature location benchmark. The existing feature location benchmark is generated automatically whereas another benchmark is human-curated one. Building a benchmark including human-curated goldsets is a challenging task because the resulting goldsets might depend on the curators. Meanwhile, a list of modules annotated with curators decisions, *e.g.*, the result of the experiment in this paper, can generate *fuzzy* goldsets, where the number of votes can be regarded as the likelihood of the relevance to the target feature. In addition, we are planning to use descriptions of variety types and explore how the types of descriptions relates to the usefulness for recognizing features.

Another direction is a support tool for developers writing a clearer description of a feature. A concrete description listing module names may be easily invalidated by a change, while an abstract description may be insufficient to recognize the feature in source code. The problem might be seen as a query refinement.

5. ACKNOWLEDGMENTS

This work was partly supported by KAKENHI Nos. 23700030, 26280021, and 60452413.

6. REFERENCES

[1] D. Cuddeback, A. Dekhtyar, and J. H. Hayes. Automated requirements traceability: The study of human analysts. In *Proc. 18th IEEE International Requirements Engineering Conference* (*RE'14*), pages 231–240, 2010.

[2] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[3] T. Ishio, S. Hayashi, H. Kazato, and T. Oshima. On the effectiveness of accuracy of automated feature location technique. In *Proc. 20th Working Conference on Reverse Engineering* (*WCRE'13*), pages 381–390, 2013.

[4] P. Mäder and A. Egyed. Assessing the effect of requirements traceability for software maintenance. In *Proc. 28th IEEE International Conference on Software Maintenance* (*ICSM'12*), pages 171–180, 2012.

[5] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, , and L. Pollock. An empirical study of the concept assignment problem. Technical Report SOCS-TR-2007.3, School of Computer Science, McGill University, 2007.

[6] J. Wang, X. Peng, Z. Xing, and W. Zhao. An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In *Proc. 27th IEEE International Conference on Software Maintenance* (*ICSM'11*), pages 213–222, 2011.