Impact Analysis of Granularity Levels on Feature Location Technique

Chakkrit Tantithamthavorn, Akinori Ihara, Hideaki Hata, and Ken-ichi Matsumoto

Software Engineering Laboratory,
Graduate School of Information Science,
Nara Institute of Science and Technology, Japan
{chakkrit-t,akinori-i,hata,matumoto}@is.naist.jp
http://www.se-naist.jp

Abstract. Due to the increasing of software requirements and software features, modern software systems continue to grow in size and complexity. Locating source code entities that required to implement a feature in millions lines of code is labor and cost intensive for developers. To this end, several studies have proposed the use of Information Retrieval (IR) to rank source code entities based on their textual similarity to an issue report. The ranked source code entities could be at a class or function granularity level. Source code entities at the class-level are usually large in size and might contain a lot of functions that are not implemented for the feature. Hence, we conjecture that the class-level feature location technique requires more effort than function-level feature location technique. In this paper, we investigate the impact of granularity levels on a feature location technique. We also presented a new evaluation method using effort-based evaluation. The results indicated that function-level feature location technique outperforms class-level feature location technique. Moreover, function-level feature location technique also required 7 times less effort than class-level feature location technique to localize the first relevant source code entity. Therefore, we conclude that feature location technique at the function-level of program elements is effective in practice.

Keywords: Feature Location, Granularity Level, Effort-Based Evaluation

1 Introduction

In modern software development, software systems continue to grow in size and complexity due to the increasing of software requirements. The early version of the Eclipse Platform project[1], which was released in May 2001, consists of 283,229 lines of code, while the latest version released in January 2014, consists of 2,674,685 lines of code. In thirteen years, the size of the Eclipse Platform project has grown almost 10 folds. This explosive growth of software size has increased more rapidly than the ability of human to maintain them. As a result, identifying where and how a feature is implemented in the source code based on a given requirement in order to implement new features, enhance existing features, or fix bugs is painstaking and time-consuming for developers.

[1] http://www.ohloh.net/p/eclipse

To help developers locate the implementation of a given feature quickly, software engineering research has paid attention to the creation of fully automated feature location techniques. These techniques suggest the source code entities where a feature request will most likely be modified based on the description of the feature request. Current research uses Information Retrieval (IR) models to locate source code entities that are textually similar to a given feature request. These source code entities can be at various levels of granularity in the program elements e.g., *the class or file level* [1, 2], and *the function or method level* [3, 4].

Prior work reports that developers need to investigate most of the non-related functions in a class file, if the class is suggested to be modified based on a given feature request [5]. Dit et al. also stated that the more fine-grained the program elements located by a technique, the more specific and accurate the technique is [6]. Thus, this paper conjectures that *feature location technique at the function-level is effective in practice*.

In this paper, we investigate the impact of granularity levels in program element on a feature location technique. We compare the results of feature location techniques at the class-level and the function-level. We use the Vector Space Model (VSM) as a baseline model for feature location technique. We evaluate the techniques on 1,968 issue reports based on three large open-source software projects. In particular, we explore the following research questions:

RQ1: Does function-level feature location technique outperform class-level feature location technique?

Function-level feature location technique outperforms class-level feature location technique.

RQ2: How much effort does function-level feature location technique actually save over class-level feature location technique?

Function-level feature location technique requires 7 times less effort than class-level feature location technique to localize the first relevant source code entity.

Without a strong understanding of which granularity levels of feature location technique is practical for developers, the impact of feature location technique will be minimal, software quality assurance resources may not be properly allocated and software releases may be delivered late and over-budget.

2 Background and Motivation

In this section, we first present the motivation behind our research hypothesis. First, we discuss some of the common findings from previous studies. We then use an issue report from Eclipse project as a typical example and discuss the implication from our observation. Finally, we discuss the challenges of our study.

2.1 Common Findings

While feature location technique research at the class-level and the function-level has been well-studied so far, we raise a question that need to be investigated whether

function-level feature location technique is more practical than class-level feature location technique. We discuss some common findings to support our research hypothesis as follows:

Long documents are not suitable for feature location technique. Previous research reported that the Vector Space Model (VSM) is the best generic IR model for feature location technique [1]. However, VSM is far from perfect because this model prefers small documents during ranking [2, 7–9]. Source code entities at the class-level is often a long document, which decreases the similarity value of VSM. Zhou et al., [2] pointed out that long documents are poorly represented in the characterization of VSM because most of a given class commonly contains words that are not relevant to the issue report i.e., *noise*. An implication is that function-level feature location technique may be able to locate source code entities more accurate than class-level feature location technique.

Class-level feature location requires a large amount of extra effort to locate the implementation of features. Class-level feature location technique often leaves developers with a large amount of extra effort needed to examine all instructions in a class until a function is located, which is not practical for developers. Hata et al., [5] noticed that developers need to investigate most of the non-related functions in a class file, if the class file is suggested to be modified. Giger et al., [10] also noticed that narrowing down the location of source code entities can save manual inspection steps. Dit et al., [6] also stated that the more fine-grained the program elements located by a technique, the more specific and accurate the technique is. An implication is that a function-level feature location technique could save human effort needed to locate source code entities that implemented the feature.

Traditional evaluation techniques are not appropriate for granularity level comparison. Hata et al., [5] reported that the function sizes are nearly 10 times smaller than the class sizes. Since Arisholm et al. reported that the cost of such quality assurance activities on a source code is roughly proportional to the size of the source code entity [11], the effort required to read top k ranked classes and top k ranked functions are different. Therefore, traditional evaluation metrics cannot be used for a fair comparison at different granularities. An implication is that a new evaluation technique considering effort required to read top ranked entities are desirable to study the impact of granularity level in the program element on feature location technique.

2.2 Observation

We use an already-fixed issue report in Eclipse Platform's project as a typical example (see Figure 1). This issue report has ID 137088[2] and was reported on April 17, 2006 for Eclipse v3.2 under Ant component. It says that there is a bug of *StringIndexOutOfBoundsException* in the function *AntLaunchDelegate.appendProperty()* when a developer tried to launch an Ant script.

[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=137088

Issue ID: 137088
Product Platform
Comp. Ant **Ver** 3.2
Summary: *StringIndexOutOfBoundsException in AntLaunchDelegate.appendProperty()*
Description: *Im getting the following crash when I try to launch an Ant script and one of my user properties is an empty string.*

For example, AntLaunchDelegate.appendProperty(..., "myName", "")

Here's the offending line of code...

```
if (value.charAt(value.length() - 1)
    == class.separatorChar) {
    commandLine.append(
       class.separatorChar);
}
```

Related class file: ant/org.eclipse.ant.ui/Ant Tools Support/org/eclipse/ant/internal/ui/launchConfigurations/AntLaunchDelegate.java

```
private void appendProperty(...) {
    commandLine.append(" \"-D"); //$NON-NLS-1$
    commandLine.append(name);
    commandLine.append('=');
    commandLine.append(value);
-   if (value.charAt(value.length() - 1)
            == class.separatorChar) {
+   if (value.length() > 0 &&
            value.charAt(value.length() - 1)
            == class.separatorChar) {
      commandLine.append(class.separatorChar);
    }
    commandLine.append("\""); //$NON-NLS-1$
}
private void appendTaskAndTypes(...) {}
private AntRunner configureAntRunner(....){}
private StringBuffer generateCommandLine(...){}
private StringBuffer generateVMArguments(...){}
protected IBreakpoint[] getBreakpoints(...){}
private String getSWTLibraryLocation(){}
private void handleException(...){}
public void launch(...){}
private void runInSameVM(...){}
private void runInSeparateVM(...){}
private void setDefaultVM(...){}
private void setProcessAttributes(...){}
private String stripUnescapedQuotes(...){}
```

Fig. 1. The issue report ID #137088 in Eclipse project and one changed line of their related class.

To identify the corresponding class file, we look into a snippet of commit logs[3] related to this issue. We found that there is only one class file (i.e., *AntLaunchDelegate.java*) that is changed to fix this issue. Without comments and headers, the size of the class file is 619 lines of code. We also found that there is only one changed line i.e., the *if* statement in the function *appendProperty()* (see Figure 1). We can imply from this observation that even if class-level feature location technique can successfully identify the relevant class file, it requires a huge amount of effort to locate the specific location of the source code that implement this feature. The observation of this example is consistent with previous findings [5, 10].

2.3 Challenges

Lack of availability of baseline datasets. Current research has built their ground-truth datasets at only one granularity level. This did not allow us to investigate the comparative results at different granularity levels. To get around this problem, we built a collection of 1,968 issue reports from three large open source software projects of the Eclipse software. These issue reports were linked to actual relevant classes and functions based on a given issue report.

[3] We used the command "git show -U0 f89a9e717db328f421432a7890b58bd656adc242" to identify changed files.

Lack of practical grounding with current evaluation methods Current research often evaluates their approach using common IR evaluation metrics such as top-*k* accuracy, precision and recall. However, these metrics only focus on the performance of ranking models without considering the human effort to read the top-*k* suggested source code entities. These metrics also do not provide a practical comparison because they ignore the different sizes of classes and functions. To tackle this challenge, we introduce a new evaluation metric which take into consideration effort. We assume that developers will inspect all the returned suggested source code entities. Therefore, we represent the term "effort" with lines of code (LOC). This allowed us to compare the performance at different granularity levels.

3 Study Design

We first present the studied projects that we used in our experiment. Second, we provide data preparation procedure. Third, we present the workflow of feature location technique using the Vector Space Model (VSM). Finally, we present the effort-based evaluation.

3.1 Studied Projects

In this study, we study three large open source software projects under Eclipse Platform, Eclipse PDE, and Eclipse JDT. All subject systems were written in Java. There are two reasons to choose these three projects. First, these projects are large, active and real-world systems, which allow us to perform a realistic evaluation of our model under their testing system. Second, each software project carefully maintains issue tracking system and source code version control repositories, which allow us to build our *ground-truth* datasets to evaluate our approach. Table 1 describes the statistics summary of dataset in more details.

Table 1. Statistics summary of studied projects.

Project Name	**Study Period**	**Issues**	**# of classes**	**# of functions**	**# LOC**
Eclipse Platform	May 2, 2001 - Dec 31, 2012	744	1,758	7,121	165,404
Eclipse PDE	June 5, 2001 - Dec 31, 2012	756	4,979	26,339	271,002
Eclipse JDT	May 24, 2001 - Dec 31, 2012	468	4,222	49,486	1,076,985
	Total	1,968	10,959	82,946	1,513,391

3.2 Data Preparation Procedure

We first obtain the source code information from version control system (VCS) and the issue report information from the issue tracking system. We then create the ground-truth dataset to test the studied projects.

Source Code Information Generally, most of popular version control systems such as Git or Subversion keep track of source code entities only in class-level. Obtaining source code information at the function-level from existing large software project is challenging. We have to keep track source code histories at the function-level form an entire software project. To do this, it is a painstaking and time-consuming activity. In this study, we used *Historage* [12] – a fine-grained version control system (VCS), which allow us to analyze source code histories at the function and class-levels from an entire software project.

Issue Report Information We obtain issue report information from the Eclipse issue tracking system[4]. We select only already-fixed issue reports which labeled as "FIXED". We also exclude issue reports where we could not establish a link to the source code entities.

Creating the Ground-Truth Data We identify changes from commit logs using the SZZ algorithm [13]. This algorithm parses the commit log messages from the source code repository Historage, looking for messages such as "Fixed issue #137088" or similar variations. If found, the algorithm establishes a link between all the source code entities in the commit transaction with the identified issue report ID. The result is a set of links between issue reports and source code entities, which we use to evaluate our approach under test.

3.3 Workflow of Feature Location Technique Using the Vector Space Model

This section provides a brief overview of the workflow for IR-based feature location technique using the Vector Space Model (VSM). First, we perform data preprocessing on the source code and issue reports. Second, we perform indexing. When a new issue report is received, we treat the issue report as a query and source code entities as a document corpus. Third, we calculate the degree of relevancy using VSM to find the most relevant source code entities based on textual features. Fourth, and Finally, we return the top k search results to the developers. We describe the details of this processing in the following steps.

Step 1: Data Preprocessing This step extracts semantic words from source code entities and issue reports. For each source code entity, we perform data preprocessing on both the class and function-levels. We remove all punctuation signs and digits. We then

[4] https://bugs.eclipse.org/bugs/

split all words into tokens and normalize them by transforming them to lower case. For multiple-word identifiers such as *GetInitialValue()*, we do not separate them into single words. The study by Sinha et al. [14] claimed that "Doing any pre-processing of the code to split identifiers into words did not yield benefits." We also do not perform any stemming process retaining the original meaning. The experiment by Hill et al. [15] concludes that "Stemming has relatively little effect." Finally, we remove common English words (e.g. a, an, the) and general programming language words (e.g. int, double, char)

Step 2: Indexing In this step, weights calculated from similarity scores represent the importance of individual words. In this study, we use $tf.idf$ weighting, a combination of the frequency of the index term occurrences in a document (the *term frequency*, or tf) and the frequency of index term occurrences over the entire collection of documents (*inverse document frequency*, or idf). The idf aims to give high weights to terms that occur in very few documents. Among many variations of weights, the *logarithm variant* was used because it can lead to better performance [16, 7]. A typical formula for tf and idf is shown in Equation 1.

$$tf(t,d) = log(f_{td}) + 1, idf(t) = log(\frac{N}{1+n_t}) \tag{1}$$

where t represents an index term, d represents a particular document, f_{td} is the number of term t occurrences in document d, N is the total number of documents, and n_t is the number of documents in which term t occurs. Finally, each term weight $w_{t\in d}$ in the document vector $\overrightarrow{V_d}$ and its norm $|\overrightarrow{V_d}|$ is calculated as follows:

$$w_{t\in d} = tf_{td} \times idf_t = (log(f_{td}) + 1) \times log\Big(\frac{N}{n_t}\Big) \tag{2}$$

$$|\overrightarrow{V_d}| = \sqrt{\sum_{t\in d}\Big((log(f_{td}) + 1) \times log\Big(\frac{N}{n_t}\Big)\Big)} \tag{3}$$

Likewise, we also obtain the vector of term weights for the query $\overrightarrow{V_q}$ and its norm $|\overrightarrow{V_q}|$.

Step 3: Similarity Function In this step, we calculate the degree of similarity between an issue report and source code entities using *cosine similarity* as shown in Equation 4. With this equation, source code entities with the highest scores are considered as the most similar to a given issue report.

$$SimilarityFunction(q,d) = \frac{\overrightarrow{V_q} \bullet \overrightarrow{V_d}}{|\overrightarrow{V_q}||\overrightarrow{V_d}|} \tag{4}$$

Step 4: Retrieval & ranking Suggested source code entities are retrieved by the model. Such a model assigns a relevant score to each source code entity that is textually similar to an issue report. This score is then used to order the entities, and a developer can select the entities with the highest score to implement new features, enhance existing features, or fix bugs.

3.4 Effort-Based Evaluation

We used source lines of code (LOC) as a proxy to measure the effort required to inspect a code. Arisholm et al., [11] pointed out that the cost of such quality assurance activities on a source code is roughly proportional to the size of the source code entity. This metric represent "effort size" with lines of code (LOC). After the classes and functions are ranked, we take all the top classes and functions whose the cumulative sum of LOC is less than or equal to a window.

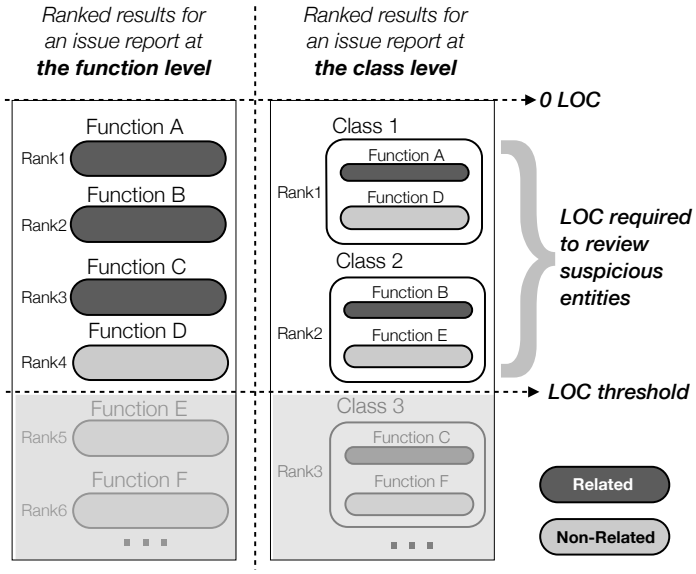

Fig. 2. Overview of effort-based evaluation.

Figure 2 shows an example of suggested source code entities at the function and class levels based on a given issue report. These entities are ranked by their relevant scores calculated from Equation 4, so the entities most likely related to an issue report are at the top. A dark gray label refers to an entity related to an issue report, while a light grey label refers to a entity which is not related to an issue report. We determine a baseline LOC threshold as the effort available for review. This LOC threshold allows us to measure the performance at different granularity levels for a fair comparison, which is appropriate for a granularity level comparison study.

4 Results

4.1 Performance

RQ1: Does function-level feature location technique outperform class-level feature location technique?

Analysis Method To answer RQ1, we performed two experiments. First, we used the *LOC-based performance* to assess the performance of feature location technique at the function and class levels. In the evaluation, for each issue report, we first obtained the rank of relevant functions and classes by calculation from Equation 4. We then checked the ranks of these locations from the search results. We performed this evaluation for all issue reports and calculated the percentage of successfully localized issue reports. For the comparison, we set the LOC threshold size ranging from 500 to 5,000 LOC with the idea that 5,000 is a reasonable number of LOC for a developer to search through before growing impatient and resorting to other means of feature location technique.

Second, we performed paired-statistical tests to measure the performance improvements brought by function-level feature location technique. In the paired-statistical tests, two chosen approaches must have the same number of data points. We formulated a one-tailed null hypothesis and the related alternative hypothesis as follows:

H_{01}: *There is no statistical difference in terms of accuracy between function-level feature location technique and class-level feature location technique.*

H_{a1}: *The LOC-based performance of function-level feature location technique is greater than the LOC-based performance of class-level feature location technique.*

We used the Wilcoxon Signed-Rank test to reject the null hypotheses H_{01} using the 1% confident level (i.e., $p - value < 0.01$).

Evaluation Metric To assess the performance, we define a metric called the *LOC-based performance* designed to measure the performance of feature location technique by taking into consideration effort. This metric will measure the percentage of successfully localized issue reports. We consider an issue report to be successfully localized if at least one suggested entity in the baseline ground-truth dataset was returned below a baseline LOC threshold. If none of the suggested entities was returned, the issue report cannot be localized. We denote the number of issue reports as N_C if the issue report is successfully localized, or N_{IC} if the issue report is not successfully localized. The following formula computes the percentage of issue reports for which are successfully localized:

$$\text{LOC-based Performance} = \frac{N_C}{N_C + N_{IC}} \tag{5}$$

To illustration, suppose that source code is selected for inspection using a feature location technique that orders source code entities (e.g., classes or functions) in terms of their similarity measure. Then, if an inspection budget allows inspection of 5,000 LOC of the source code, the higher the number of successfully localized issue reports, the better the technique is.

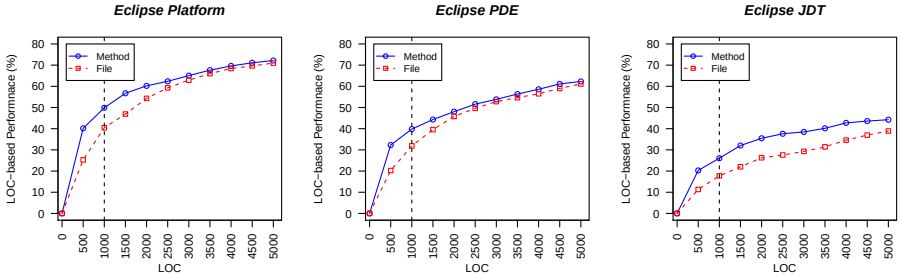

Fig. 3. The performance of function-level feature location technique compared to class-level feature location technique using LOC-based performance

Results Figure 3 shows the performance of function-level feature location technique compared to class-level feature location technique using LOC-based performance. The x-axis denotes the total LOC to be examined. The y-axis represents the LOC-based performance values. For every studied project, we used a comparative size at 1,000 LOC threshold. It is apparent from this figure that function-level feature location technique correctly identifies the location of source code entities for a larger number of issue reports than class-level feature location technique. For the Eclipse Platform project, function-level feature location technique can successfully identify 49.86%, while class-level feature location technique can identify 40.45% of the issue reports. For the Eclipse PDE project, function-level feature location technique can identify 39.81%, while class-level feature location technique can identify 31.87% of the issue reports. For the Eclipse JDT project, function-level feature location technique can identify 26.07%, while class-level feature location technique can identify 17.74% of the issue reports. Furthermore, we also have statistical significant evidence to reject the null hypothesis H_{01}. The p-values for all studied systems are below the standard significant value, $p-value < 0.01$.

Summary For these three studied projects, we can confidently conclude that function-level feature location technique is significantly achieves better performance than class-level feature location technique.

4.2 Efficiency

RQ2: How much effort does function-level feature location technique actually save over class-level feature location technique?

Analysis Method To answer RQ2, we consider efficiency in two dimensions: (1) effort required to find the first relevant source code entity; and (2) effort required to find 80% of the relevant source code entities. We performed this experiment for all issue reports and represented the results by box-plots of the distribution of LOC that needed to be examined. Intuitively, the less effort required to review, the more efficient the technique is.

Then, we assess whether the differences in the amount of LOC to be reviewed to find the first relevant source code entity and 80% of the relevant source code entities are statistically significant between function-level feature location technique and class-level feature location technique. To select an appropriate statistical test, we use the Shapiro-Wilk test to analyze the distributions of our data points. We observe that these distributions do not follow a normal distribution. Thus, we use a nonparametric test, i.e., Wilcoxon-Mann-Whitney test, to test our null hypotheses to answer **RQ2**. We reject the null hypotheses H_{02}, H_{03} using the 1% confident level (i.e., $p - value < 0.01$). We formulate one-tailed null hypotheses and the related alternative hypotheses as follows:

H_{02}: *There is no statistical difference in terms of the amount of LOC to be reviewed to find the first relevant source code entity between function-level feature location technique and class-level feature location technique.*

H_{a2}: *The amount of LOC to be reviewed to find the first relevant source code entity of function-level feature location technique is less than class-level feature location technique.*

H_{03}: *There is no statistical difference in terms of the amount of LOC to review to find 80% of the relevant source code entities between function-level feature location technique and class-level feature location technique.*

H_{a3}: *The amount of LOC to be reviewed to find 80% of the relevant source code entities of function-level feature location technique is less than class-level feature location technique.*

Result

(1) Effort Required to Find The First Relevant Source Code Entity For each studied project, as shown in Figure 4, we plotted a quartile box plot to show the distribution of the amount of LOC that must be reviewed to identify the first relevant source code entities. The x-axis represents the level of granularity, either function-level or class-level. The y-axis represents the total LOC to be examined. From this figure, it is apparent that, function-level feature location technique required less effort than class-level feature location technique to identify the first feature request location. For the Eclipse Platform, function-level feature location technique required 113 LOC (median), while class-level feature location technique required 906 LOC (median) to find the first feature request location. This means function-level feature location technique requires less effort than class-level feature location technique to find the first feature request location. For the Eclipse PDE, function-level feature location technique requires 154 LOC, while class-level feature location technique requires 861 LOC to find the first feature request location. For the Eclipse JDT, function-level feature location technique requires 248 LOC, while class-level feature location technique requires 1,839 LOC to find the first feature request location.

These results indicate that for these projects, function-level feature location technique required less effort than class-level feature location technique to find the first relevant source code entity. Interestingly, for those issue reports with function-level feature location technique, the maximum LOC to identify the first relevant source code

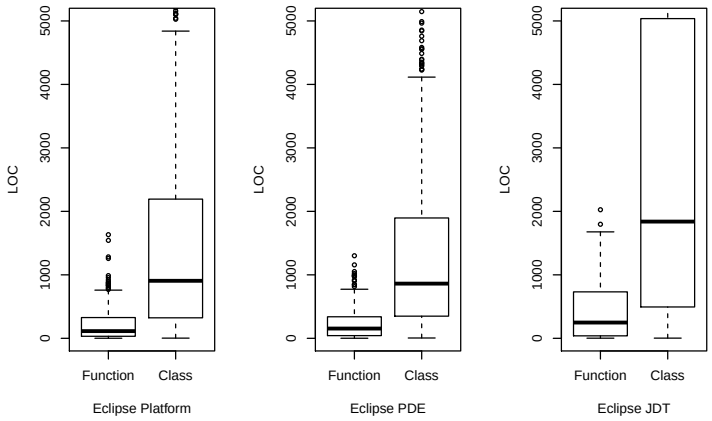

Fig. 4. Distribution of LOC that need to be examined to identify the first relevant source code entity.

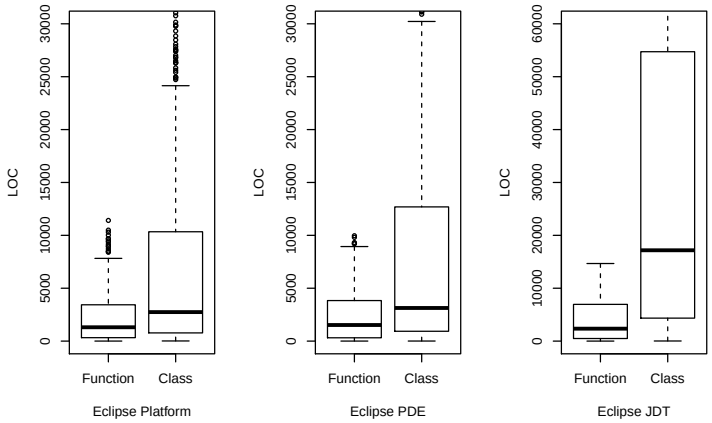

Fig. 5. Distribution of LOC that need to be examined to localized 80% of relevant source code entities.

entity ranged from 1,633 LOC to 2,026 LOC, while the maximum LOC for class-level feature location technique ranged from 10,789 LOC to 51,775 LOC. These results help us confirm that function-level feature location technique required less effort than class-level feature location technique to find the first relevant source code entity. We also found that the difference in terms of the amount of LOC to be reviewed to find the first

relevant source code entity are statistically significant ($p - value < 0.01$) to reject the null hypothesis H_{02}.

(2) Effort Required to Find 80% of The Relevant Source Code Entities Figure 5 shows the distributions of the amount of LOC that needed review to localize 80% of the relevant source code entities. The x-axis represents the level of granularity, either function-level or class-level. The y-axis represents the total LOC to be examined. For the Eclipse Platform, function-level feature location technique required 1,309 LOC (median) to find 80% of the relevant source code entities, while class-level feature location technique required 2,744 LOC (median). For the Eclipse PDE, function-level feature location technique required 1,522 LOC and class-level feature location technique required 3,132 LOC to find 80% of the relevant source code entities. For the Eclipse JDT, function-level feature location technique required 2,354 LOC and class-level feature location technique required 17,176 LOC to find 80% of the relevant source code entities. We also found that the difference in terms of the amount of LOC to review to find 80% of the relevant source code entities are statistically significant ($p - value < 0.01$) to reject the null hypothesis H_{03}.

Summary The results indicate for these projects, function-level feature location technique required 7 times less effort than class-level feature location technique to localize the first relevant source code entity and 4.4 times less effort to localize 80% of the relevant source code entities. We can conclude that function-level feature location technique can effectively save human effort in identifying relevant source code entities.

5 Threats to Validity

The main threats to internal validity lies in our truth data collection technique, which relies on the SZZ algorithm [13]. Although the SZZ algorithm is a commonly used technique for linking feature request reports to source code entities, Bird et al. reported that there is a linking bias in identifying feature requests with revision logs and feature request reports [17]. Recently, Nguyen et al. has been proposed a novel linking algorithm [18], which may alleviate this threat.

Threats to External Validity These are concerned with the generalization of our findings. In this paper, we used three large open source software under the Eclipse Project to conduct our case study. All these projects were written in Java. Although these are large real-world software projects, our results may not generalize to other open source or commercial software projects, especially in other programing languages.

Threats to Construct Validity The main threat refers to the effort-based evaluation. In our comparative study between function-level and class-level feature location technique, we used effort-based evaluation. However, the effort used here (LOC) may not reflect actual efforts. As a first approximation, it seems acceptable to represent the term effort using LOC because developers may consider much more complex relations or other deep dependencies. In future research, we may need to consider complexity and dependency metrics.

6 Conclusions and Future Works

In this research, we addressed the question of practical feature location technique. We investigated whether function-level feature location technique is more practical than class-level feature location technique by conducting a large-scale empirical study to compare the results of IR-based feature location technique at class and function levels. Our main findings are.

- For the same amount of inspection effort, function-level feature location technique outperforms class-level feature location technique. Especially at lower levels of inspection effort, function-level feature location technique correctly identifies the relevant source code entities for a larger number of issue reports than class-level feature location technique.
- Function-level feature location technique required 7 times less effort than class-level feature location technique to localize the first relevant source code entity. We also found that Function-level feature location technique required 4.4 times less effort than class-level feature location technique to find 80% of the relevant source code entities. These results indicate that function-level feature location technique can help developers locate a larger percentage of issue reports with less inspection efforts.

From these results, we conclude that feature location technique at the function-level of program elements is effective in practice. Using function-level feature location technique should help developers reduce the level of inspection effort needed to find feature requests, increase the number of relevant source code entities, and improve the overall handling of feature location technique.

Based on this study, future research will focus on empirical studies of actual efforts, and conduct experiments to confirm the results and functions with other open-source and commercial software projects, and other programing languages.

Acknowledgement

We would like to thank the anonymous reviewers for their very constructive feedback on early drafts of this work. This research is conducted as part of Grant-in-Aid for Young Scientists (B), 25730045, Grant-in-Aid for Young Scientists (Start-up), 25880015, and for Exploratory Research 25540026 by Japan Society for the Promotion of Science (JSPS).

References

1. Rao, S., Kak, A.: Retrieval from Software Libraries for Bug Localization : A Comparative Study of Generic and Composite Text Models. In: Proceedings of the 8th IEEE Working Conference on Mining Software Repositories (MSR'11). (2011) 43–52
2. Zhou, J., Zhang, H., Lo, D.: Where Should the Bugs Be Fixed ? In: Proceedings of the 34th International Conference on Software Engineering (ICSE'12). (2012) 14–24

3. Lukins, S.K., Kraft, N.a., Etzkorn, L.H.: Bug Localization Using Latent Dirichlet Allocation. Information and Software Technology **52**(9) (September 2010) 972–990
4. Wang, S., Khomh, F., Zou, Y.: Improving Bug Localization using Correlations in Crash Reports. In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13). (2013)
5. Hata, H., Mizuno, O., Kikuno, T.: Bug Prediction Based On Fine-Grained Module Histories. In: Proceedings of the 34th International Conference on Software Engineering (ICSE'12). (June 2012) 200–210
6. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature Location in Source Code : A Taxonomy and Survey. Journal of Software: Evolution and Process **25**(1) (2013) 53–95
7. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Number c. Cambridge Unviersity Press, New York, NY, USA (2008)
8. Tantithamthavorn, C., Teekavanich, R., Ihara, A., Matsumoto, K.: Mining A Change History to Quickly Identify Bug Locations : A Case Study of the Eclipse Project. In: Proceedings of the 2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'13). (2013) 108–113
9. Tantithamthavorn, C., Ihara, A., Matsumoto, K.: Using Co-change Histories to Improve Bug Localization Performance. In: Proceedings of the 14th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'13). (July 2013) 543–548
10. Giger, E., D'Ambros, M., Pinzger, M., Gall, H.C.: Method-Level Bug Prediction. In: Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement (ESEM'12). (2012) 171–180
11. Arisholm, E., Briand, L.C., Johannessen, E.B.: A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models. Journal of Systems and Software **83**(1) (January 2010) 2–17
12. Hata, H., Mizuno, O., Kikuno, T.: Historage : Fine-grained Version Control System for Java. In: Proceedings of the 12th International Workshop on Principles of Software Evolution (IWPSE11). (2011) 96–100
13. Zimmermann, T., Zeller, A.: When Do Changes Induce Fixes ? ACM SIGSOFT Software Engineering Notes **30**(4) (2005) 1–5
14. Sinha, V.S., Mani, S., Mukherjee, D.: Is Text Search an Effective Approach for Fault Localization : A Practitioners Perspective. In: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH'12). (2012) 159–170
15. Hill, E.: On the Use of Stemming for Concern Location and Bug Localization in Java. In: Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation (SCAM'12). (2012) 184–193
16. Croft, B., Metzler, D., Strohman, T.: Search Engines: Information Retrieval in Practice. Addison-Wesley (2010)
17. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.: Fair and Balanced ? Bias in Bug-Fix Datasets Categories and Subject Descriptors. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (FSE'09). (2009) 121–130
18. Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Nguyen, T.N.: Multi-Layered Approach for Recovering Links Between Bug Reports and Fixes. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12). (2012) 1–11