A Preliminary Study on Using Code Smells to Improve Bug Localization

Aoi Takahashi, Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki
School of Computing
Tokyo Institute of Technology
Tokyo 152–8552, Japan
{takahashi-a-at,natthawute,hayashi,saeki}@se.cs.titech.ac.jp

ABSTRACT

Bug localization is a technique that has been proposed to support the process of identifying the locations of bugs specified in a bug report. A traditional approach such as information retrieval (IR)-based bug localization calculates the similarity between the bug description and the source code and suggests locations that are likely to contain the bug. However, while many approaches have been proposed to improve the accuracy, the likelihood of each module having a bug is often overlooked or they are treated equally, whereas this may not be the case. For example, modules having code smells have been found to be more prone to changes and faults. Therefore, in this paper, we explore a first step toward leveraging code smells to improve bug localization. By combining the code smell severity with the textual similarity from IR-based bug localization, we can identify the modules that are not only similar to the bug description but also have a higher likelihood of containing bugs. Our preliminary evaluation on four open source projects shows that our technique can improve the baseline approach by 142.25% and 30.50% on average for method and class levels, respectively.

CCS CONCEPTS

• **Software and its engineering** → *Maintaining software*;

KEYWORDS

bug localization, code smell, information retrieval

ACM Reference Format:
Aoi Takahashi, Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. 2018. A Preliminary Study on Using Code Smells to Improve Bug Localization. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196321.3196361

1 INTRODUCTION

Bug localization refers to the process of identifying the location(s) of a given bug, which can be a tedious task in large-scale software development projects. Therefore, many ideas have been proposed to automate this process using software development information, such as bug description (information retrieval or IR-based) [7] or execution trace (dynamic analysis) [13] approaches.

To improve the accuracy of these approaches, many papers have proposed combining an approach with extra information to improve performance (e.g., hybrid approaches). For example, Tantithamthavorn et al. [11] used co-change history to improve bug localization under the assumption that the files that were changed together with a buggy file in the past should be fixed together. Zhou et al. [15] proposed BugLocator, an approach that combines similar bug reports that have been fixed in the past with an IR-based approach. Saha et al. [9] presented BLUiR, an approach that enhances the traditional approach by incorporating structural information, i.e., class names would be more important than comments or identifiers, in addition to using similar bug reports in the past. Furthermore, Wang et al. [12] proposed AmaLgam which combines version history, similarity report, and structural information to improve bug localization. In addition, Youm et al. [14] proposed Bug Localization using Integrated Analysis (BLIA) by analyzing texts, stack traces, and comments in bug reports, structured information of the source files, and the source code change history. More details on work regarding hybrid bug localization can be found in [10].

Most of the existing work try to improve the accuracy of bug localization by focusing on combining with extra information, which may not always be available. In addition, the likelihood of each module having a bug is often ignored or the modules are treated equally in existing approaches, even though this may not be the case. For instance, many papers have investigated and found that modules having code smells are more likely to be changed and faulty [4, 5]. As a result, in this paper, we explore a first step toward utilizing code smells, which can be directly generated from the source code, under the assumption that they can be used to improve bug localization.

The main contributions of this paper are as follows.

(1) We show that code smells can be combined with traditional bug localization to improve its accuracy.
(2) We propose a metric representing the combination of textual similarity and the property of code smell.
(3) We present an empirical study regarding the optimal proportion of textual similarity and the property of code smell.

The new idea of this paper is the use of code smells to improve bug localization because although code smells have been found to be related to change- and fault-proneness, no study has considered the use of code smells for bug localization. The paper most closely

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196361

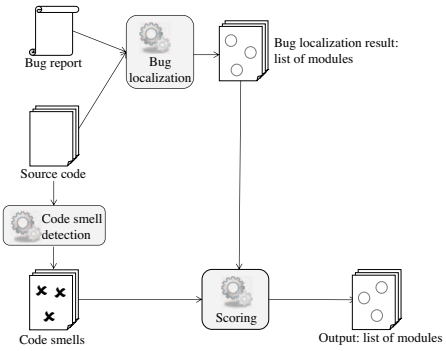


Figure 1: Overview of the proposed technique.

related to this work is a short survey on hybrid bug localization by Shi et al. [10]. Whereas the use of various information such as stack trace and version history commonly could be found in the literature according to the survey, the use of code smells has not been studied. Because we intend this study to be the first step toward exploring the usage of code smells to improve bug localization, we use only one property of code smell, i.e., severity, which describes how strong a given code smell is. Therefore, we welcome suggestions on how to exploit other properties of code smell such as the type of code smell.

The remainder of this paper is organized as follows. The next section explains our proposed technique. Then, we present a preliminary evaluation on four open source projects in Section 3.3. Finally, we conclude this study and state our future work in Section 4.

2 PROPOSED TECHNIQUE

We propose a technique to improve bug localization by using code smells. An overview of our proposed technique is shown in Figure 1, where each gray node represents a subprocess. First, we use the content of a bug report, containing summary and description, together with the source code as inputs for an existing bug localization approach. We then detect the code smells of the source code using existing tools. Finally, we calculate the score of each module by combining the outputs from the bug localization and code smell detection approaches. The next subsections explain each subprocess in detail.

2.1 Bug Localization

We use the Vector Space Model (VSM) approach for bug localization because it has been shown to be the most effective approach among IR-based techniques [8]. VSM takes a bug description and the source code as inputs and calculates the textual similarity $Sim(m)$ of each module m.

2.2 Code Smell Detection

For code smell detection, we choose an approach that takes source code as its only input and then generates a list of modules containing code smell together with its properties such as smell type or its severity. The severity of a code smell represents how strong the code smell is. For example, Marinescu [6] defined it as: "Severities are computed by measuring how many times the value of a chosen metric exceeds a given threshold". The severity of the code smell allows a developer to distinguish the importance of individual smells of the same type, which can then be used for prioritizing or filtering code smells. In this study, the summation of the severities of every code smell in module m is defined as $Sev(m)$. However, we only utilize code smells with the same granularity as the targeted module.

2.3 Scoring

In this paper, we combine traditional IR-based bug localization with a code smell property, i.e., severity. We propose the metric *Bug Likelihood Index* (*BLI*), which is a linear combination of the textual similarity from the traditional IR-based approach and the code smell severity. The underlying reason for choosing a linear combination is that it allows us to analyze the weight of each factor. In addition, it is commonly used in hybrid bug localization where multiple factors need to be combined [10–15].

To calculate *BLI* of a module m, we first need to normalize both the textual similarity between a bug description and the module m and the severity of code smell in module m to a scale of 0–1. The normalized textual similarity $Sim(m)$ and severity $Sev(m)$ are represented as $nSim(m)$ and $nSev(m)$ respectively. Then, *BLI* of module m can be calculated using the following formula:

$$BLI(m) = (1 - \alpha) \cdot nSim(m) + \alpha \cdot nSev(m)$$

where $0 \leq \alpha \leq 1$. Setting $\alpha = 0$ means that we ignore code smell severity, which is equivalent to using only the traditional IR-based approach. On the other hand, setting $\alpha = 1$ means that we ignore the textual similarity and use only code smell severity for localizing bugs. In addition, setting $\alpha = 0.5$ means that we give equal weight to both textual similarity and code smell severity. Thus, α is a parameter of our approach that can be adjusted for optimum performance. The details will be discussed in the next section.

3 PRELIMINARY EVALUATION

3.1 Research Questions

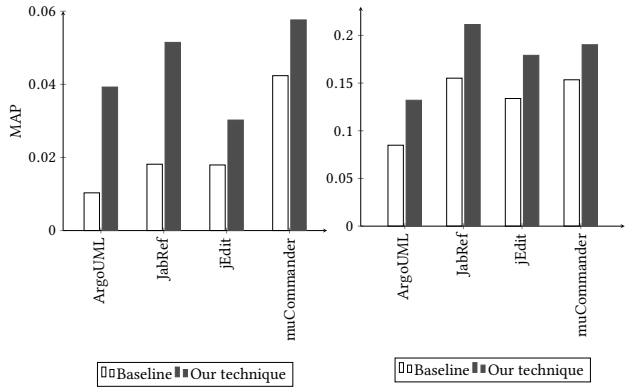
To validate the potential of our approach, we conduct an evaluation based on the following research questions.

RQ1: *Can our approach improve the traditional IR-based bug localization?*

Our hypothesis assumes that the modules having code smells are more likely to be the location of bugs. Therefore, to validate whether combining code smell property can help improve the traditional IR-based bug localization, we need to compare the accuracy of using only the traditional approach with the accuracy of our approach, i.e., combining with a property of code smell.

RQ2: *What is the best parameter value of our approach?*

As discussed earlier, the α value is the parameter of our approach representing the weight of each factor. Such a parameter can be optimized in an empirical manner, e.g., developers can adjust its value based on historical information. However, to understand this approach, we need to know how our technique performs for different α values on different projects. We suspect that our technique will perform differently for each α value and that its optimum value will depend on the project.



(a) Method level. **(b) Class level.**

Figure 2: Comparison of MAP values between the baseline and our approach.

3.2 Experimental Setup

3.2.1 Experimental Implementation. We implement a tool to support the automation process of the framework described in Figure 1. The tool takes the output of existing bug localization and code smell detection tools, calculates *BLI*, and then presents the results.

We use TraceLab [2] as bug localization tool to calculate the textual similarity between the source code and bug report because the tool contains the VSM component.

As for code smell detection, we use inFusion Ver. 1.9.0 because it can output many kinds of properties of a code smell, e.g., the severity of code smell.

3.2.2 Data Collection. In this study, we use the data set of Dit et al. [3] to conduct the experiment. The data set contains a list of issues from an issue tracking system that need to be fixed before a specific release, the source code snapshot of the version, and the modules that was modified to solve each issue (gold set modules) of four open source projects (ArgoUML, JabRef, jEdit, and muCommander). Because the data set was mainly created for benchmarking feature location techniques, which are closely related to and share similar settings as bug localization, it contains many types of issues, e.g., bugs, features, and patches. However, as this study focuses on bug localization, we extract only the issues classified as bugs. Thus, there are 74, 36, 86, and 81 bugs for ArgoUML, JabRef, jEdit, and muCommander, respectively.

3.2.3 Data Analysis. To evaluate our approach, we use the Mean Average Precision (MAP) [1] as a metric because it is widely used to evaluate ranking results, especially in bug localization [8–10, 12–15].

3.3 RQ1: Can our approach improve the traditional IR-based bug localization?

3.3.1 Study Design. To answer the first research question, we compute the accuracies of the traditional IR-based approach (our baseline) and compare them with the ones of our approach. We select the optimum parameter of each project in an empirical manner that will be discussed in RQ2.

3.3.2 Results and Discussion. Figure 2 presents the results of our experiment for the method level and class level. For the method level, our approach can improve the traditional IR-based bug localization approximately by 281%, 184%, 68%, and 36% for ArgoUML, JabRef, jEdit, and muCommander, respectively.

For the class level, the improvement is less than for the method level, but our method can still improve traditional IR-based bug localization for every project by approximately 36%, 34%, 24%, and 28% for ArgoUML, JabRef, jEdit, and muCommander, respectively.

By comparing the method and class levels, we can observe that the overall accuracies of the method level are lower than the ones of the class level. The underlying reason may be that method level modules are more difficult to predict due to the granularity of the modules, i.e., the method level modules are more fine-grained and have a higher number than the class level ones. In addition, when we consider the improvement of the traditional IR-based approach, we can see that the improvement of the method level is much higher than the one of the class level. This may be due to the low accuracy of the method level, which makes it easier to improve. In addition, another possible explanation is that the traditional IR-based approach may work better on the class level because of the larger amount of content when computing the textual similarity.

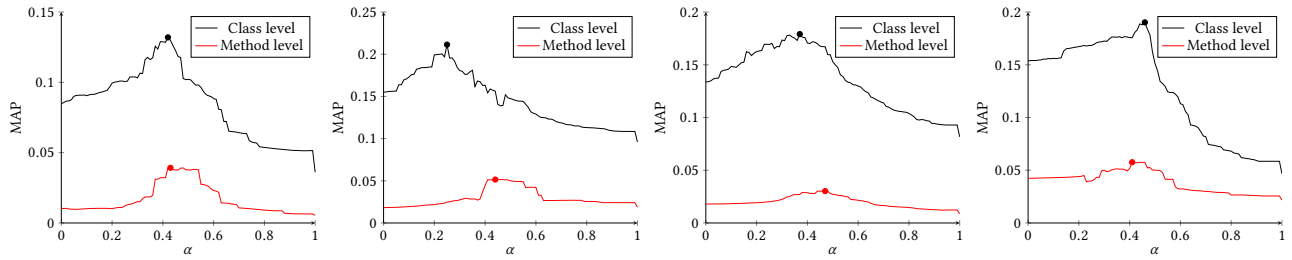
In conclusion, our approach can improve traditional IR-based bug localization by 142.25% and 30.5% on average for the method level and class level, respectively.

3.4 RQ2: What is the best parameter value of our approach?

3.4.1 Study Design. To answer the second research question, we conduct experiments using different α values in the range 0.00–1.00 with steps of 0.01 for each granularity (class level and method level). Then, we calculate MAP of each setting.

3.4.2 Results and Discussion. Figure 3 shows the result of our experiment. Each graph shows the MAP value at different α for each project. The point for $\alpha = 0$ refers to the accuracy when using only the traditional approach, whereas $\alpha = 1$ refers to the accuracy when using only the code smell property to localize bug. We can observe that, in most cases, our technique performs worst when using only code smell property ($\alpha = 1$). This is expected because code smell property does not contain the information of the bug report. However, when we combine the traditional approach with code smell property ($0 < \alpha < 1$), we can see improvements for some values. This indicates that combining the code smell property with the traditional approach can improve the accuracy depending on the weights of the combination.

Considering the class level (black lines), we can see that the points where MAP reaches its highest values are $\alpha = 0.42$, 0.25, 0.37, and 0.46 for ArgoUML, JabRef, jEdit, and muCommander, respectively (black dots). For the method level (red lines), the optimum weights are 0.43, 0.44, 0.47, and 0.41 for ArgoUML. JabRef, jEdit, and muCommander, respectively (red dots). The average optimum α value is approximately 0.41, indicating that our technique



(a) ArgoUML. (b) JabRef. (c) jEdit. (d) muCommander.

Figure 3: MAP of results obtained using different α values.

works best when the weight of the code smell severity is slightly lower than for the textual similarity from the traditional approach.

However, when comparing the optimum value of class level and method level, we can see that the optimum value of the class level is smaller than the one of the method level in most projects. As discussed in RQ1, one possible explanation is that the traditional bug localization performs better with class level and therefore need less extra information of code smells than for the method level.

To sum up, while the best parameter value may be different and depends on each project, the average value is approximately 0.41.

3.5 Threats to Validity

The first threat to validity may be the dependence on the parameter value that we chose in RQ1 as we selected the best value empirically. However, empirical tuning of hyper parameters is commonly done in bug localization [10–15]. In addition, in future work, we plan to conduct a study on a larger scale, i.e., not only single version, but throughout the software life cycle. In this way, we can employ machine learning techniques to tune the parameter and generalize our approach.

Moreover, the baseline used in this study is not a state-of-the-art approach but the traditional IR-based bug localization. Nevertheless, the main purpose of this paper is to explore the opportunity of using code smells to improve bug localization. In future work, we plan to combine code smells with a state-of-the-art approach and validate whether we can still improve the bug localization.

4 CONCLUSION

In this paper, we explored the first step of using code smells to improve bug localization by combining a property of code smell with existing information from an IR-based approach. By applying our approach to four open source projects on both the class and method levels we showed that it could improve the traditional bug localization approach by 142.25% and 30.50% on average for method and class levels, respectively.

We plan to explore other properties of code smell such as smell types to see how they impact the accuracy of bug localization. Additionally, we need to explore the possibility of combining code smells with not only IR-based but also other approaches such as static and dynamic bug localization.

ACKNOWLEDGMENTS

This work was partly supported by JSPS Grants-in-Aid for Scientific Research Numbers JP15K15970, JP15H02683, and JP15H02685.

REFERENCES

[1] Bruce Croft, Donald Metzler, and Trevor Strohman. 2009. *Search Engines: Information Retrieval in Practice* (1 ed.). Addison-Wesley Publishing Company.
[2] Bogdan Dit, Evan Moritz, and Denys Poshyvanyk. 2012. A tracelab-based solution for creating, conducting, and sharing feature location experiments. In *Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC'12)*. 203–208.
[3] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95.
[4] Latifa Guerrouj, Zeinab Kermansaravi, Venera Arnaoudova, Benjamin CM Fung, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2017. Investigating the relation between lexical smells and change-and fault-proneness: an empirical study. *Software Quality Journal* 25, 3 (2017), 641–670.
[5] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275.
[6] Radu Marinescu. 2012. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development* 56, 5 (2012), 9–1.
[7] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*. 263–272.
[8] Shivani Rao and Avinash Kak. 2011. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR'09)*. 43–52.
[9] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. 345–355.
[10] Zhendong Shi, Jacky Keung, Kwabena Ebo Bennin, and Xingjun Zhang. 2018. Comparing learning to rank techniques in hybrid bug localization. *Applied Soft Computing* 62 (2018), 636–648.
[11] Chakkrit Tantithamthavorn, Akinori Ihara, and Ken-ichi Matsumoto. 2013. Using co-change histories to improve bug localization performance. In *Proceedings of the 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'13)*. 543–548.
[12] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*. 53–63.
[13] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. 181–190.
[14] Klaus Changsun Youm, June Ahn, and Eunseok Lee. 2017. Improved bug localization based on code change histories and bug reports. *Information and Software Technology* 82 (2017), 177–192.
[15] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 14–24.