Revisiting Context-Based Code Smells Prioritization: On Supporting Referred Context

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

ABSTRACT

Because numerous code smells are revealed by code smell detectors, many attempts have been undertaken to mitigate related problems by prioritizing and filtering code smells. We earlier proposed a technique to prioritize code smells by leveraging the context of the developers, i.e., the modules that the developers plan to implement. Our empirical studies revealed that the results of code smells prioritized using our technique are useful to support developers' implementation on the modules they intend to change. Nonetheless, in software change processes, developers often navigate through many modules and refer to them before making actual changes. Such modules are important when considering the developers' context. Therefore, it is essential to ascertain whether our technique can also support developers on modules to which they are going to refer to make changes. We conducted an empirical study of an open source project adopting tools for recording developers' interaction history. Our results demonstrate that the code smells prioritized using our approach can also be used to support developers for modules to which developers are going to refer, irrespective of the need for modification.

CCS CONCEPTS

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

KEYWORDS

code smell, issue tracking system, impact analysis, interaction history

ACM Reference format:
Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. 2017. Revisiting Context-Based Code Smells Prioritization. In *Proceedings of XP '17 Workshops, Cologne, Germany, May 22-26, 2017,* 5 pages.
DOI: .1145/3120459.3120463

1 INTRODUCTION

Code smells are often interpreted as indicators of problems or design flaws in source code [8]. They can be seen as factors that cause technical debts. For instance, a class having a God Class code smell is a class that tends to be packed with functionality from other classes and which tends to control data of other classes. As a result, the class is likely to be overly large and complex. This kind of design flaw can have a distinctly negative effect on software maintenance. It affects many perspectives of source code quality such as understandability and extendibility. Many studies have been conducted to investigate code smell effects [9, 19].

One important problem of current code smell detection is that the number of outputs is overwhelming. Consequently, developers do not use static analysis tools such as code smell detectors because of numerous detected warnings [10]. To resolve this shortcoming, numerous attempts have been made to reduce the number of code smells using different factors by filtering and prioritizing the code smells [1, 3, 7, 17].

Nevertheless, with the limited time available for solving code smells or improving the quality of the source code in real world situations, it is preferable to solve code smells in the modules related to the context of the developers, as reported in the literature. One characteristic that Murphy-Hill and Black presented in their "Seven Habits of a Highly Effective Smell Detector" paper [14] is *Context-Sensitivity*, which is to suggest code smells that are related to the current context of developers first. Yamashita et al. [20] also stated that developers in their study need code smell detectors that support context-sensitivity. Furthermore, Bavota et al. [2] recommended in their work that perspectives of developers need to be considered when recommending refactoring opportunities.

Our earlier work proposed a technique to prioritize code smells based on the developers' context [15]. The definition of *context* described herein is similar to *task context* defined by Kersten and Murphy as "the information–a graph of elements and relationships of program artifacts–that a programmer needs to know to complete that task" [12]. We estimated the context of developers by application of an impact analysis technique to textual information, e.g., summary and description of issues in an issue tracking system, to predict the modules that are likely to be locations for making changes to resolve issues. Then, we prioritized the list of code smells based on results obtained using an impact analysis technique. We evaluated our technique using a set of code smells occurring in the modules that were actually modified by developers as the oracle obtainable from source code repositories such as Git or Subversion. The results confirmed that our technique can be used to support modules that are going to be modified by developers.

However, to make changes, it is common that developers start the process by navigating through multiple places and referring to them before they actually modify the source code. We designated the modules to which developers are going to refer as *referred modules* and modules to which developers must make actual modifications as *modified modules*. When considering the developers' context to prioritize code smells, it is rational to estimate the modified modules as the context because those modules are the locations at which developers are going to make modifications directly. Nonetheless, the referred modules are also important to be used for estimation of the context because those modules are the modules where developers are going to read and comprehend although they require no modification. Consequently, it is important to investigate whether the context-based prioritization technique is useful to support both situations when considering the referred modules and the modified modules as the context of developers. However, in our previous work, we only studied and confirmed that the technique might be useful for situations involving modified modules. As a consequence, one important point remains in our context-based prioritization technique: whether our technique can support not only the modified modules but also the referred modules.

To bridge the gap as described earlier, we conducted an empirical study of the use of our technique to support the referred modules. Such information is obtainable using interaction-recording tools such as Mylyn [11]. Our result confirmed that our technique can be used to support the referred modules as well. In other words, if developers solve code smells according to the results of our technique, then it can help them improve several aspects of source code quality in the related modules. Consequently, it might reduce the cost for their implementation.

The main contribution of this work is a demonstration that the result of context-based code smells prioritization is useful not only to support developers during modifications but also during navigation and reference to source code for resolving issues in an issue tracking system.

The structure of this paper is organized as follows. Section 2 summarizes our previous context-based code smell prioritization approach. Section 3 presents a brief introduction to source code interaction history. We then report our empirical study in Section 4, discuss threats to validity in Section 5, and conclude this paper in Section 6.

2 CONTEXT-BASED CODE SMELLS PRIORITIZATION

Our earlier work proposed a technique to prioritize code smells using the context of developers. The technique emphasized support of issue-driven software development projects adopting an issue tracking system to manage their lists of issues. We simulated a real-world situation in which, before releasing major or minor versions, the development team must complete a list of issues. In this case, because the list of issues in issue tracking system includes information of the task that developers must perform, it is useful to estimate the developers' context. In other words, we can predict the modules that are likely to be related to each issue. Because those modules are likely to be modified by developers, we can regard the modules that are likely to be modified as an estimation of the developers' context.

Figure 1 includes an overview of the proposed technique and the evaluation. The inputs of the approach are a list of change descriptions from the issue tracking system and the source code of the targeted project. For each change description, we applied impact analysis to generate a list of modules that are likely to be the location for implementing the change. One example of the fundamental impact analysis technique is the Vector Space Model (VSM), representing documents and queries as vectors [16]. The technique then determines the similarity between documents and queries by calculation of the cosine similarity. In this case, documents are source code modules; queries are change descriptions. Therefore, we are using the technique to predict modules that resemble the change descriptions. We detect code smells of the specified project using an existing code smell detector such as a metric-based technique [13]. Subsequently, we calculate the proposed *Context Relevance Index* (*CRI*) for each code smell based on the result from impact analysis technique. The *CRI* value represents the relevance of each smell to the context of developers. Higher values of *CRI* signify greater relevance to the context. Finally, the technique outputs the new list of prioritized smells that is ordered by the *CRI* value. In summary, code smells that are likely to be modified by developers for many issues are put at the top of the list. The evaluations were conducted by comparing the results with a set of code smells occurring in the modules that were actually modified by developers. A detailed explanation was presented in our earlier report [15].

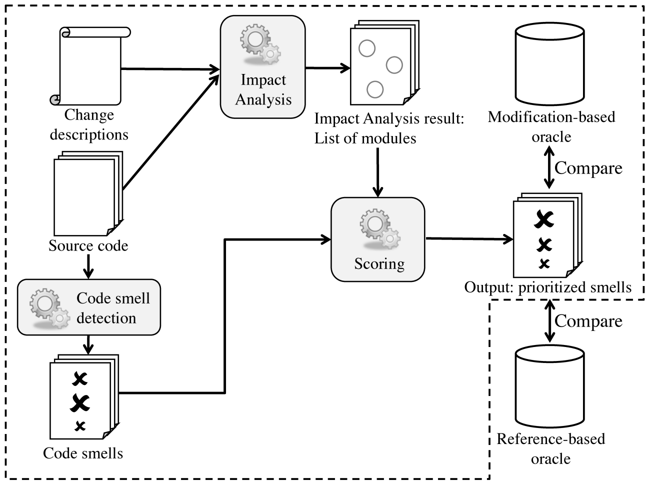

Figure 1: Overview of the proposed technique and the evaluation.

3 MYLYN'S INTERACTION HISTORY

Software development teams have been widely adopting version control systems such as Git or Subversion to keep track of source code changes. However, one shortcoming of the version control system is that it records only code components that have been modified at the commit time, while in fact, developers also refer to many code components that they might not have actually modified. To overcome such a lack of information, many interaction history recording tools such as Mylyn [11] have been proposed. Mylyn records developers' activities on the tasks that developers work on, for instance when they select text in the editor. Such interaction events are recorded in XML file format and are uploaded to the issue tracking system of each issue on which the developers worked. The attributes of each element of interaction event that we are considering include the following[1].

- **Kind**: type of interaction
- **StructureKind**: type of artifact interacted upon by developers
- **StructureHandle**: the artifact interacted upon by developers

In Mylyn, when developers select some text in the editor, the interaction event in which the value of *Kind* is *Edit* is going to be recorded. However, such information is insufficient to ascertain whether developers were referring only to the source code or were also modifying the source code. Therefore, to classify the referred and the modified modules, we used this information together with the information from a version control system. We regard the modules in the version control system as modified modules and regard the modules in the interaction history that are not in the version control system as the referred modules.

4 EMPIRICAL STUDY

4.1 Motivation

As described herein, our previous study used the oracle created solely based on changes in the version control system. Such information only contains modules that were changed at the commit time. However, it is very likely that there are also many other modules to which developers are going to refer even though they need not make any modification. Developers might need to understand those modules to be able to make the actual modification. Therefore, we want to confirm through this study whether the list of code smells that is prioritized by our technique can be useful to support not only the modules that developers are going to modify but also the modules to which developers are going to refer.

4.2 Study Design

4.2.1 Experimental Implementation. As in our previous work, we implemented an automated tool for conducting the experiment. Our tool was designed to connect with other tools such as TraceLab [5, 6] for carrying out impact analysis and inFusion[2] for detecting code smells used for this study.

To generate the gold sets or the modified modules based on the version control system, we used tools proposed by Dit et al. [6]. Their tools generate gold sets by comparing two versions of each file based on Eclipse's Abstract Syntax Tree (AST). The result of the tools is a list of modules that were modified for each commit.

4.2.2 Data Collection. For this study, we used the Mylyn Task project as our subject because Mylyn projects contains a large amount of Mylyn interaction history information [18]. We mined over 700 commits between ver. 3.07–3.21. from its version control system[3]. Then, because the developers of Mylyn projects are recommended to use Eclipse's plugin to commit changes to Git repository and because the commits include the URL of the bug that the developers are solving, we were able to create links between

[1]https://wiki.eclipse.org/Mylyn/Integrator_Reference

[2]Unfortunately, inFusion is no longer available.

[3]https://github.com/eclipse/mylyn.tasks

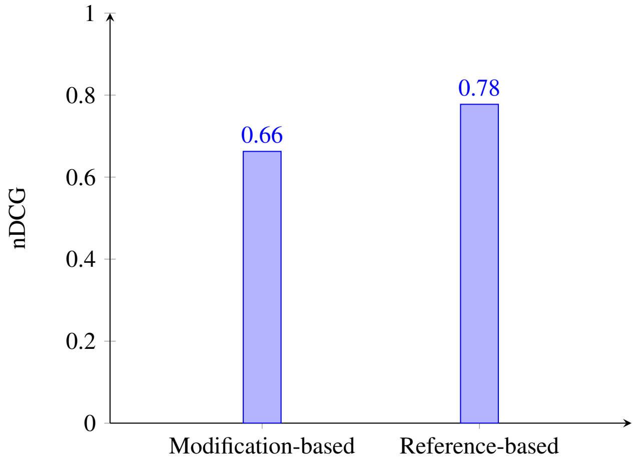

Figure 2: Comparison of the nDCG value between results of modification-based oracle and reference-based oracle.

commits and issues by searching for the pattern of corresponding bug's URL (such as https://bugs.eclipse.org/bugs/show_bug.cgi?id=500712) for each commit message. We created 516 links that included 334 unique issues. Subsequently, we crawled Mylyn's issue tracking system according to the URL that we obtained from commit messages. Because not every developer uploaded the interaction history to the issue tracking system, we had to filter out issues that did not contain interaction information. The reason behind this is that the interaction history was necessary for our study to analyze the modules that developers did not modify but referred to. In all, we obtained 95 issues that satisfy our requirements.

We defined oracles of two types: modification-based and reference-based. The modification-based oracle is a set of code smells that appear in the modules that were modified by developers, while the reference-based oracle is a set of code smells that appear in the modules that were referred to by developers. When we generated the oracle, out of 141 detected code smells, besides code smells that are not related to the context, we obtained 37 code smells in modification-based oracles and 61 code smells in reference-based oracles. In addition, the code smells that are only in the reference-based oracle are 27. The code smells that are only in the modification-based oracle are 3.

4.2.3 Data Analysis. For this assessment, we used Normalized Discounted Cumulative Gain (nDCG) [4], which is a popular metric for evaluating the quality of ranking documents. nDCG assigns a higher value to relevant documents appearing in the higher position on the list than the relevant documents appearing in the lower position of the list. Additionally, we can assign the degree of relevance to each relevant item. Therefore, in this study, we can use nDCG to reflect the quality of our technique, which is used to assign code smells that affect many issues at the top of the list. We calculated the nDCG value for both the modification-based oracle and the reference-based oracle.

Table 1: Top 10 Prioritized Code Smell Results

Rank	Smell Type	Class Name	CRI	#RI[a]	#MI[b]
1	God Class	TasksUiInternal	7.89	7	4
2	God Class	TasksUiPlugin	5.51	9	3
3	God Class	TaskListIndex	5.48	3	1
4	God Class	AbstractTaskEditorPage	5.36	3	2
5	God Class	TaskDataManager	5.29	4	2
6	God Class	TracRepositoryConnector	5.26	1	1
7	God Class	AttachmentUtil	5.24	4	1
8	God Class	SynchronizeTasksJob	5.17	3	0
9	Data Class	TaskData	4.97	3	0
10	God Class	BugzillaRepositoryConnector	4.69	0	2

[a]Number of referring issues
[b]Number of modifying issues

4.3 Results and Discussion

Figure 2 presents the results of our study. We obtained an nDCG value of 0.66 in the case of a modification-based oracle and 0.78 in the case of a reference-based oracle, which indicates that our technique not only prioritizes code smells that have an effect on the modules that developers are going to modify but also prioritizes code smells that have an effect on the modules to which developers are going to refer. Therefore, if developers solve code smells according to the list provided by our technique, then it is going to support them, e.g., make the code easier to understand, not only when they are modifying source code, but also when they are referring to the source code.

Table 1 presents the top 10 results of the code smells after being prioritized with our technique together with the number of issues that refer to and modify the module having the code smells. For instance, our technique predicted that the God Class code smell of TasksUiInternal class is related to many issues that developers must solve with *CRI* of 7.89. Our investigation of the actual interaction history of developers during the analyzed period revealed that for seven issues, developers referred to this class without actually modifying it. Subsequently, we investigated the change history from the version control system, which revealed that, for four issues, developers had to modify this class to resolve the issues. Therefore, if developers solve the code smell of this class before starting the implementation process, then several perspectives of this class, such as understandability and extendibility, can be improved. Consequently, the implementation costs of 11 issues related to this class can be reduced.

In addition, if we consider the case of God Class code smell in SynchronizeTasksJob class in the table, then it is apparent that no issue exists in the analyzed period that modified this class. Consequently, solving the code smell in this class does not support any code modification process of the developers. However, the analysis result shows that there are three issues to which developers referred to this class, even though they did not modify it. Therefore, although solving this code smell might be unable to support developers for the code modification, it can support developers when they refer to this code, i.e., improve the source code comprehensibility.

Furthermore, if we analyze the number of oracles as described earlier, then it is apparent that the code smells that are only in the reference-based oracle are 27. Such a number is high compared with other numbers such as code smells that are only in the modification-based oracle, which are only 3. Therefore, we conclude that the referred modules have a strong effect on developers' implementation process and that they should be considered when considering the developers' context. In addition, because our technique can predict numerous code smells that are in reference-based oracle (e.g., most items in top 10 ranks are predicted correctly), we can reach the conclusion that our context-based code smells prioritization can suggest not only smells in a modified context but also those in a referred context.

5 THREATS TO VALIDITY

As the main purpose of this study is to present early results of our study, there are some threats to validity that need to be addressed. First, we conducted the study on a small scale with only one project. Second, while other interaction recording tools might be able to used, we used only Mylyn for this study, and it might have caused false negatives. Finally, we used only one metric to assess this study. Nonetheless, we planned to justify these threats by conducting a larger scale study with different kinds of datasets and tools in the future. Additionaly, performing different evaluation schemes such as different kinds of metrics or with professional developers remains as a subject for our future work.

6 CONCLUSION

As described herein, we revisited our context-based code smells prioritization to ascertain whether the approach can be useful to support a situation in which developers refer to source code modules. We conducted an empirical study with an oracle based on a version control system that represents the modules which developers modified and an oracle based on the interaction history, which represents the modules to which developers referred to resolve the issues. Our results confirmed that the context-based code smells prioritization can support situations in which developers modify and refer to source code.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Roberta Arcoverde, Everton Guimaraes, Isela Macia, Alessandro Garcia, and Yuanfang Cai. 2013. Prioritization of Code Anomalies Based on Architecture Sensitiveness. In *Proceedings of the 27th Brazilian Symposium on Software Engineering (SBES'13)*. 69–78.
[2] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An Experimental Investigation on the Innate Relationship between Quality and Refactoring. *Journal of Systems and Software* 107 (2015), 1–14.
[3] Zadia Codabux and Byron J. Williams. 2016. Technical Debt Prioritization Using Predictive Analytics. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE'16)*. 704–706.
[4] Bruce Croft, Donald Metzler, and Trevor Strohman. 2009. *Search Engines: Information Retrieval in Practice* (1 ed.). Addison-Wesley Publishing Company.
[5] 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.

[6] 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.

[7] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Riccardo Roveda. 2015. Towards a Prioritization of Code Debt : A Code Smell Intensity Index. In *Proceedings of the Seventh IEEE International Workshop on Managing Technical Debt (MTD'15)*. 16–24.

[8] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[9] Felienne Hermans and Efthimia Aivaloglou. 2016. Do Code Smells Hamper Novice Programming? A Controlled Experiment on Scratch Programs. In *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16)*. 1–10.

[10] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 672–681.

[11] Mik Kersten and Gail C. Murphy. 2005. Mylar: A Degree-of-Interest Model for IDEs. In *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD'05)*. 159–168.

[12] Mik Kersten and Gail C. Murphy. 2006. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06)*. 1–11.

[13] Michele Lanza and Radu Marinescu. 2006. *Object-Oriented Metrics in Practice*. Springer.

[14] Emerson Murphy-Hill and Andrew P. Black. 2008. Seven Habits of a Highly Effective Smell Detector. In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering (RSSE'08)*. 36–40.

[15] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. 2016. Context-based code smells prioritization for prefactoring. In *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16)*. 1–10.

[16] Gerard Salton and Michael J. McGill. 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc.

[17] Santiago A. Vidal, Claudia Marcos, and J. Andrés Díaz-Pace. 2016. An Approach to Prioritize Code Smells for Refactoring. *Automated Software Engineering* 23, 3 (2016), 501–532.

[18] Akihiro Yamamori, Anders Mikael Hagward, and Takashi Kobayashi. 2017. Can Developers' Interaction Data Improve Change Recommendation?. In *Proceedings of the 41st Annual IEEE Computer Software and Applications Conference (COMPSAC'17)*.

[19] Aiko Yamashita and Leon Moonen. 2012. Do Code Smells Reflect Important Maintainability Aspects?. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*. 306–315.

[20] Aiko Yamashita and Leon Moonen. 2013. Do Developers Care about Code Smells? An Exploratory Survey. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13)*. 242–251.