How Do Developers Select and Prioritize Code Smells? A Preliminary Study

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

***Abstract*—Code smells are considered to be indicators of design flaws or problems in source code. Various tools and techniques have been proposed for detecting code smells. The number of code smells detected by these tools is generally large, so approaches have also been developed for prioritizing and filtering code smells. However, the lack of empirical data regarding how developers select and prioritize code smells hinders improvements to these approaches. In this study, we investigated professional developers to determine the factors they use for selecting and prioritizing code smells. We found that *Task relevance* and *Smell severity* were most commonly considered during code smell selection, while *Module importance* and *Task relevance* were employed most often for code smell prioritization. These results may facilitate further research into code smell detection, prioritization, and filtration to better focus on the actual needs of developers.**

I. Introduction

Code smells were first defined by Fowler [1] to represent problems in source code possibly caused by bad design decisions. This definition mostly applies to descriptive languages so many studies have aimed to interpret these smells in a formal manner. In particular, several previous work use source code metrics to detect code smells (e.g., [2]). In addition, many attempts have been made to detect code smells using other information such as historical data [3].

However, the number of results from such techniques are numerous because of the large volumes of source code. Thus, research has focused on trying to reduce the amount of code smells by prioritizing or filtering them. Previously proposed techniques for prioritizing and filtering code smells have focused on different methods such as the severity-based technique where more severe code smells are given higher priority [4], [5], [6], the context-based technique where code smells are prioritized according to the specific contexts of developers [7], or by using combinations of multiple factors to prioritize code smells [8], [9], [10].

However, improving these methods has been hindered by the lack of empirical evidence regarding factors used for selecting and prioritizing code smells. There is no clear indication that which factors should be considered in each approach. One explanation for this situation is that empirical studies of code smells have focused mainly on the negative effects of code smells instead of how developers handle them.

To address this shortcoming, we performed a study on professional developers to determine the factors that they employ to handle code smells, especially, during the prefactoring phase. In this phase, developers refactor the source code before changing their source code in order to facilitate its implementation [11]. Since existing techniques can be classified into filtration and prioritization techniques, we consider them both.

The main contribution of this study is that we determined the factors considered by developers when selecting and prioritizing code smells for analysis in the prefactoring phase. These factors may help researchers and tool developers to focus on the most appropriate factors concerning code smells during prioritization and filtration. To the best of our knowledge, this is the first study to empirically investigate the factors considered by professional developers when selecting and prioritizing code smells.

The remainder of this paper is organized as follows. First, we summarize related research regarding empirical studies of code smells in Section II. In Section III, we explain the design and details of our study. We present the analysis of the results in Section IV. Threats to validity are discussed in Section V. Finally, we give our conclusions and suggest further research in Section VI.

II. Related Work

Code smells are a very active topic for academic and industrial researchers throughout the world, and thus many empirical studies have been conducted to obtain insights into the issues that the community should explore.

Most previous studies have considered the negative effects of code smells. For example, Khomh et al. [12] found that classes with code smells are more likely to change and become faulty. Abbes et al. [13] investigated the effects of code smells on program comprehension and found that a combination of Blob and Spaghetti Code significantly decreased the performance of developers. Their results agreed with those obtained by Yamashita and Moonen [14], who found that inter-smell relationships were related to maintenance problems. In addition, Yamashita and Moonen [15] explained that code smells can be used to partly reflect maintainability aspects of software.

In addition to research into the effects of code smells, previous studies over the past decade have investigated how developers deal with code smells. Thus, Yamashita and Moonen [16] reported a survey of 85 professional developers concerning code smells. Most of the subjects in their study stated the need for improved tools to detect code smells, especially tools with context-sensitive features. Palomba et al. [17] studied how developers perceive code smells and found that

they perceived code smells differently as problems according to the types of code smells. Arcoverde et al. [18] performed an exploratory survey to understand why code smells remain in source code and found that concern about breaking client code was one explanation. Peters and Zaidman [19] investigated the lifespan of code smells by mining a software repository to determine the perspectives of developers regarding code smells. They found that developers were aware of code smells, but they were unlikely to solve them.

Previous empirical studies have mainly focused on the negative effects of code smells and how developers perceive them, but there is no empirical study that has considered how developers select and prioritize code smells. Thus, the aim of this study was to address this shortcoming in order to identify the issues that the research community should consider.

III. Study Design

Many methods have been proposed based on the use of different factors to prioritize and filter code smells. Studies concluded that the main reason for refactoring by developers is to facilitate task implementation rather than removing the code smell itself [20], [21]. Therefore, in order to obtain empirical evidence of the factors used for filtering and prioritizing code smells, we conducted a study in a situation where many factors could be observed, including the factors related to the developer's tasks. Thus, we extended our previous controlled experiment [10] on the code smells selected by developers before working on specific tasks. The details of our previous study are explained in the following.

A. Research Questions

To obtain empirical evidence of how developers select and prioritize code smells, we focused on the following two research questions.

- **RQ1:** *What are the factors used by developers in the code smell selection process?*
- **RQ2:** *What are the factors used by developers in the code smell prioritization process?*

In the first question, the selection process is focused on code smells that developers need to address in a timely manner. The results could allow the research community to focus on factors used in practical code smell detection or filtration processes to reduce the number of false positives (code smells that are not harmful from the perspective of developers). The second question focuses on the prioritization process which defines the *order* in which code smells should be addressed based on the results of the code smell selection process, which could facilitate more practical approaches to code smell prioritization by focusing on the key problems.

B. Data Collection

The data in this study were obtained from an extension of our previous study [10]. Our previous study dealth with evaluating a code smell prioritization technique to determine whether it agreed with the process followed by professional developers. The previous study employed the source code for the JabRef project, a list of five issues, gold set methods (methods modified to address each issue), and 22 code smells belonging to the Blob Class, Data Class, God Class, and Schizophrenic Class as detected by inFusion ver. 1.9.0. The subjects comprised 10 professional developers with working experience ranging from 2–13 years. Each subject was provided with a list of five issues, including a summary and description, as typically shown on the issue tracking system for the task to be completed. In addition to the content for each issue, the subjects were provided with the solution for each issue in terms of the diff files in order to reduce the workload for the subjects. The subjects were then asked to consider all the issues and the related changes. Subsequently, a list of code smells, including the class name, package name, type of code smell, and a detailed description explaining why inFusion considered each problem to be a code smell were provided to the subjects. The source code for JabRef, including modules with code smells, was also provided. Finally, the subjects were requested to select code smells that they considered should be refactored after considering all the information.

TABLE I
Examples of Responses and Their Corresponding Codes

Response	Codes
It involves many issues	Task relevance
This file has to be changed according to the issue list in this release. The class has too many functionalities and it is also hard to navigate.	Task relevance, Smell severity
[The related task is] not difficult to fix.	Refactoring cost
Util Class is a centric class. This class was invoked by many classes, so this class should be fixed first.	Module importance

Our previous work [10] focused only on the modules that developers considered should be refactored, whereas the main purpose of this study was to determine *why* they made these decisions. In addition, our previous work focused only on the code smell selection process, whereas this work also investigated the code smell prioritization process. Therefore, we performed further investigations by: 1) gathering more concrete evidence of why the subjects selected or did not select a specific code smell; and 2) asking the subjects to prioritize the code smells that they selected using a ranking scale (i.e., 1, 2, 3, ...) as well as their reasons. The responses obtained from the subjects allowed us to analyze the factors that affected their selection and prioritization of code smells.

C. Data Analysis

After obtaining the results from the participants, we used a coding technique from grounded theory to analyze the results because it is suitable for studying human aspects of software engineering [22]. In addition, this technique has been used widely in software engineering research, including studies of code smells [16]. The initial codes were generated first for the analysis. The codes were not fixed or limited, so they could be modified, added, or deleted as necessary. Two of the authors then acted as investigators and completed the process by reading the responses of the subjects and assigning appropriate codes to each response. The investigators discussed the cases

in the event of a disagreement. At the end of the process, we combined closely related codes, and after analyzing the results, we finally obtained 15 codes. The explanation of each code is as follows:

1) *Co-located smells:* Multiple code smells appear in the same module.
2) *Maintainability:* The module is difficult to maintain, or the subjects want to improve the maintainability.
3) *Module dependency:* Code smells should be solved in a specific order.
4) *Module importance:* The module plays an important role in the system.
5) *Readability:* The module is difficult to read, or the subject wants to improve the readability.
6) *Refactoring cost:* The cost incurred by performing refactoring operations to remove a code smell.
7) *Smell false positive:* The subject does not consider that the result obtained by the code smell detector is a code smell.
8) *Smell severity:* The subject considers that the code smell is severe or not severe.
9) *Smell coupling:* One smell is related to another.
10) *Task implementation cost:* The cost incurred for implementing a specific task is high or low.
11) *Task implementation risk:* The risk of implementing a specific task is high or low.
12) *Task importance:* The related task is or is not important.
13) *Task relevance:* The smell is related to the subject's task.
14) *Testability:* The module is difficult to test, or the subjects want to improve the testability.
15) *Understandability:* The module is difficult to understand, or the subjects want to improve the understandability.

Some examples are given in Table I. For instance, the response "*It involves many issues*" was assigned with the code *Task relevance*, and the response "*This file has to be changed according to the issue list in this release. The class has too many functionalities and it is also hard to navigate.*" was assigned with the codes *Task relevance* and *Smell severity*. Moreover, codes such as *Smell false positive* were also assigned to the responses representing the reasons that the subject did not select a specific code smell.

IV. Analysis of Results

A. RQ1: What are the factors used by developers in the code smell selection process?

Table II shows the factors used by developers when selecting code smells according to our analysis. Clearly, *Task relevance* was the most common factor used in the selection process, where developers tended to select code smells related to their tasks. Some of the responses made by the subjects were very straightforward because they only considered the relevance to their tasks (e.g., "*It is related to the issue #4*"), whereas some of the responses were also concerned with factors in addition to *Task relevance* (e.g., the response "*It is related to an issue that we will address soon, and it would be good if we can separate the logic into another class to make it testable and more readable*" is concerned with the *Task relevance*, *Testability*, and *Readability*). These results support previous studies showing that developers tend to refactor source code mainly to support the implementation of their tasks [20], [21].

TABLE II
Factors Used in Smell Selection Process

Code	Number of responses
Task relevance	33
Smell severity	11
Task implementation cost	5
Testability	5
Co-located smells	4
Module importance	2
Readability	2
Smell false positive	2
Smell coupling	2
Maintainability	1
Refactoring cost	1
Understandability	1

TABLE III
Factors Considered Together in Smell Selection Process

Code	Number of responses
Task relevance, Smell severity	9
Task relevance, Testability	5
Task relevance, Readability	2
Task relevance, Smell coupling	2

The second most common factor was *Smell severity*. We did not provide the subjects with the severity values obtained from the code smell detector used in this experiment in order to prevent cognitive bias, i.e., the subjects might have selected code smells with high severity values without actually analyzing them. Instead, we provided the subjects with the source code related to each code smell for the analysis. When we analyzed the responses of the subjects, we assigned the *Smell severity* code to responses that contained some specific adverbs related to a degree such as *too* or *very*, e.g., "*Functions have too many dependencies for example screens, menus, etc.*" This indicated that the developers tended to consider the *Smell severity* when they were selecting code smells.

In addition to the two factors mentioned above, the subjects also considered other factors. For instance, one subject stated the reason why they did not choose a particular smell as: "*This should be selected but we have just added a parameter in this release. The major change is in FieldContentSelector.java.*" This indicates that the developer also considered the *Task implementation cost*. Another subject indicated the reason why they selected a code smell as "*Two code smells in one file,*" which demonstrates that factors such as *Co-located smells* were also considered by the subjects.

Furthermore, previous studies often used multiple factors to detect and filter code smells, so we also conducted a further analysis of the factors considered together when developers selected the code smells, thereby obtaining insights into the factors that should be used together when detecting or filtering

TABLE IV
FACTORS USED IN SMELL PRIORITIZATION PROCESS

Code	Number of responses
Module importance	14
Task relevance	10
Testability	5
Smell severity	4
Maintainability	3
Refactoring cost	3
Co-located smells	2
Module dependency	2
Readability	2
Task implementation cost	2
Task importance	2
Task implementation risk	1

TABLE V
FACTORS CONSIDERED TOGETHER IN SMELL PRIORITIZATION PROCESS

Code	Number of subjects
Module importance, Task relevance	4
Module importance, Testability	3
Task relevance, Testability	3
Co-located smells, Task relevance	2
Maintainability, Task relevance	2
Module importance, Readability	2
Module importance, Refactoring cost	2
Module importance, Smell severity	2
Readability, Task relevance	2
Readability, Testability	2
Smell severity, Task relevance	2

code smells. Thus, for each response given by the subjects, we counted each pair of codes that appeared together more than one time.

The results in Table III represent the pairs of factors that the subjects considered together when selecting a code smell, excluding the ones that were assigned to only one response. It is apparent that the *Task relevance* and *Smell severity* were the most common factors. For example, one of the subjects stated the reason why they selected a code smell as: "*This file has to be changed according to the issue list in this release. The functions are too long,*" thereby mentioning two factors. The second item in Table III comprises the combination of *Task relevance* and *Testability*, where the results show that *Task relevance* was the most common factor considered by developers but it was also often considered together with another factor, i.e., *Testability* in this case (e.g., "*In this release, we have to add specific behavior to fix a bug in issue #4, and thus we have to refactor the code so that we can write the unit test more easily.*").

In conclusion, *Task relevance* and *Smell severity* were the most common factors used by developers in the code smell selection process.

B. RQ2: What are the factors used by developers in the code smell prioritization process?

Table IV shows the factors used by developers in the code smell prioritization process. In contrast to the factors used for code smell selection, *Module importance* was the most common factor considered in the prioritization process. For instance, one of the subjects stated that they ranked a code smell with the highest priority because: "*Util Class is a centric class. This class is invoked by many classes. Thus, this class should be fixed first.*" Another subject stated that they ranked a code smell with the second highest priority because: "*It is important but less important than the main UI class.*" Thus, the developers tended to first prioritize modules with important roles in the system and then other modules with lower priority.

However, the second most common factor was still *Task relevance*. We found that the numbers of tasks related to code smells were often included in the responses. For example, one subject mentioned that they assigned a code smell as the first item to fix because: "*It involves many issues,*" and the reason for the second smell was: "*It only involves issue #1.*" Another subject gave a similar reason why a code smell was ranked first: "*This issue list has three issues related to this single file. This should be considered the highest priority to be fixed.*"

Furthermore, other factors such as the *Refactoring cost* (e.g., "*Low effort [for refactoring] is required.*") and *Module dependency* (e.g., "*This file should be addressed after the Util class to consider the lower risk of the code change.*") were also considered by the subjects.

As discussed earlier, many prioritization techniques have been proposed based on combinations of multiple factors despite the lack of empirical evidence in support of this approach. To address this shortcoming, we also analyzed the factors considered together when developers prioritized code smells. We asked the subjects to state their reason for prioritizing each code smell, but we combined the codes for every response given by a subject in the investigation, which differed from our analysis of the selection process. This is because code smell prioritization is a comparative process, i.e., the subjects had to compare different smells and give higher importance to one smell, but less importance to others. Thus, we could determine the pairs of factors used by developers to prioritize code smells. The results, excluding the ones that were assigned by only one subject, are presented in Table V. According to these results, *Module importance* and *Task relevance* were the most common factors considered together by developers when prioritizing code smells (e.g., "*It should be fixed first because it is related to the issue and it is a share class.*"). In addition, the second pair that developers used most often was *Module importance* and *Testability* (e.g., "*It would be better if the main class of the UI project is readable and testable. In addition, it would reduce the time required for testing.*").

In summary, *Module importance* was the most common factor used for prioritization and the second was *Task relevance*.

C. Implications

As some existing techniques have already used the same factors reported in this study [4], [5], [6], [7], [10], we encourage the research community to keep focusing on the factors, namely, *Task relevance* and *Smell severity* for future

development, thus improving the practical utility of these research tools.

In addition to the main factors discussed above, various other factors such as the *Task implementation cost*, *Module importance*, or *Module dependency* have not been considered by the research community when developing techniques. We strongly recommend considering these factors when developing new techniques such as code smells detection, filtration, and prioritization to propose opportunities for refactoring in the future.

Lastly, because we observed some combinations of factors assigned by many subjects, we encourage the development of new techniques that consider such combinations.

V. Threats To Validity

The internal validity of this study is dependent on the data set that we used. First, the data set was designed for evaluating a code smell prioritization technique for prefactoring, so it may have missed some factors considered by developers in different situations. Second, the subjects were not the main developers of the project investigated in this study. Therefore, the results may differ if experiments are conducted on a source code that the developers are familiar with; for instance, they may consider some code smells as false positives or may select code smells that are indirectly impacted by changes. Conducting investigations with the main developers of open source projects may be beneficial.

Similar to other empirical studies, the external validity of this study is dependent on the scale of the experiment. The subjects differed in terms of their background and the number of years of experience, but the number of developers who participated in this study was only ten, and they might not have been representative subjects. In addition, this study only investigated one project with four types of code smells. Thus, a larger scale study might be beneficial.

Finally, the construct validity might depend on the codes that we assigned to each response. The process was conducted by two investigators, but there may have been some bias during the process. Furthermore, the responses obtained from the subjects might not represent the actual reasons why they selected or prioritized code smells. To mitigate this threat to validity, we plan to conduct detailed follow-up interviews with some of the subjects.

VI. Conclusion

In this study, we conducted an experiment to determine how professional developers selected and prioritized code smells. Our findings agree with previous studies where developers gave higher priority to code smells related to their specific context, i.e., the tasks upon which they were working. First, *Task relevance* was the most common factor considered during code smell selection, followed by *Smell severity*, and both were also often considered together during the selection process. Second, *Module importance* was the most common factor in the code smell prioritization process, followed by *Task relevance*. Moreover, other factors such as the *Task implementation cost* and *Co-located smells* were considered in both processes. We recommend that researchers and tool developers focus on the factors used for code smell filtration and prioritization identified in this study.

Acknowledgments

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

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2006.

[3] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proc. ASE*, 2013, pp. 268–278.

[4] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt : A code smell intensity index," in *Proc. MTD*, 2015, pp. 16–24.

[5] F. A. Fontana, V. Ferme, and M. Zanoni, "Poster: Filtering code smells detection results," in *Proc. ICSE*, 2015, pp. 803–804.

[6] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM J. Res. Dev.*, vol. 56, no. 5, pp. 9:1–9:13, 2012.

[7] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based code smells prioritization for prefactoring," in *Proc. ICPC*, 2016, pp. 1–10.

[8] R. Arcoverde, E. Guimaraes, I. Macia, A. Garcia, and Y. Cai, "Prioritization of code anomalies based on architecture sensitiveness," in *Proc. SBES*, 2013, pp. 69–78.

[9] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Autom. Softw. Eng.*, vol. 23, no. 3, pp. 501–532, 2016.

[10] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based approach to prioritize code smells for prefactoring," *J. Softw. Evol. Proc.*, 2017. [Online]. Available: http://dx.doi.org/10.1002/smr.1886

[11] V. Rajlich, *Software Engineering: The Current Practice*. Chapman and Hall – CRC, 2011.

[12] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Emp. Softw. Eng.*, vol. 17, no. 3, pp. 243–275, 2012.

[13] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Proc CSMR*, 2011, pp. 181–190.

[14] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proc ICSE*, 2013, pp. 682–691.

[15] ——, "To what extent can maintenance problems be predicted by code smell detection?–an empirical study," *Inf. Softw. Technol.*, vol. 55, no. 12, pp. 2223–2242, 2013.

[16] ——, "Do developers care about code smells? An exploratory survey," in *Proc. WCRE*, 2013, pp. 242–251.

[17] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *Proc ICSME*, 2014, pp. 101–110.

[18] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in *Proc WRT*, 2011, pp. 33–36.

[19] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Proc CSMR*, 2012, pp. 411–416.

[20] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *J. Syst. Softw.*, vol. 107, pp. 1–14, 2015.

[21] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *Proc FSE*, 2016, pp. 858–870.

[22] R. Hoda, J. Noble, and S. Marshall, "Using grounded theory to study the human aspects of software engineering," in *Proc HAoSE*, 2010, pp. 5:1–5:2.