PAPER *Special Section on Knowledge-Based Software Engineering*

An Investigative Study on How Developers Filter and Prioritize Code Smells*

Natthawute SAE-LIM†a), ***Nonmember***, **Shinpei HAYASHI**†, ***and*** **Motoshi SAEKI**†, ***Members***

SUMMARY Code smells are indicators of design flaws or problems in the source code. Various tools and techniques have been proposed for detecting code smells. These tools generally detect a large number of code smells, so approaches have also been developed for prioritizing and filtering code smells. However, lack of empirical data detailing how developers filter and prioritize code smells hinders improvements to these approaches. In this study, we investigated ten professional developers to determine the factors they use for filtering and prioritizing code smells in an open source project under the condition that they complete a list of five tasks. In total, we obtained 69 responses for code smell filtration and 50 responses for code smell prioritization from the ten professional developers. We found that *Task relevance* and *Smell severity* were most commonly considered during code smell filtration, 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.

key words: *code smells, code smell filtration, code smell prioritization, practitioner's perspective*

1. Introduction

Code smells were first defined to represent problems in source code possibly caused by bad design decisions [2], [3]. This definition mostly applies to descriptive languages, so many studies have aimed to interpret these smells in a formal manner. In particular, several previous techniques use source code metrics to detect code smells [4]–[8]. In addition, many attempts have been made to detect code smells using other information such as historical data [9], [10].

However, results from such techniques are numerous because of the large volume of source code. Therefore, the task of screening code smells is left to developers as they are unlikely to be able to solve all code smells due to time constraints. The task of screening code smells can be divided into two steps. The first step deals with code smell filtration, i.e., developers select only a subset of code smells that they think should be addressed at the moment. Subsequently, they prioritize the results of the filtration process in the second step, i.e., decide the order in which the code smells should be handled. To support this task, many code smell filtration and prioritization techniques have been proposed. The techniques have focused on different methods such as a severity-based technique where code smells with higher severity are given higher priority [11]–[13], a context-based technique where code smells are prioritized according to the specific contexts of developers [14], or by using combinations of multiple factors to prioritize code smells [15]–[17].

Even though the research community has developed several approaches to support the code smell detection process, there is still a need for better tools from practitioners [18]. This may indicate that the approaches are not aligned with practitioners' actual needs. However, improving these approaches has been hindered by the lack of empirical evidence regarding factors used for filtering and prioritizing code smells, i.e., there is no clear indication about what 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. For these reasons, it is essential to identify the factors that should be considered, e.g., by gathering practitioners' opinions.

To start with, we performed a study on professional developers to determine the factors that they consider to handle code smells, especially during the prefactoring phase. In this phase, developers refactor the source code before implementing their code [19]. As existing techniques can be classified into filtration and prioritization techniques, we consider both.

We conducted the study with ten professional developers by asking them to select and prioritize code smells of an open source project under the condition that they complete a list of five tasks. The reasons they used for selecting and prioritizing code smells were then collected and analyzed. In total, we obtained 69 responses for code smell filtration and 50 responses for code smell prioritization.

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

The remainder of this paper is organized as follows. First, we summarize related studies regarding empirical studies of code smells and techniques of code smell prior-

Manuscript received October 30, 2017.
Manuscript revised March 7, 2018.
Manuscript publicized April 20, 2018.

†The authors are with the School of Computing, Tokyo Institute of Technology, Tokyo, 152–8552 Japan.

*This paper is revised based on [1], which appeared in proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution, © 2017 IEEE.

a) E-mail: natthawute@se.cs.titech.ac.jp

DOI: 10.1587/transinf.2017KBP0006

itization and filtration in Sect. 2. In Sect. 3, we explain the design and details of our study. We present the analysis of the results in Sect. 4. Threats to validity are discussed in Sect. 5. Then, we presented our future work in Sect. 6. Finally, we give our conclusions and suggest further studies in Sect. 7.

2. Related Work

2.1 Empirical Studies on Code Smells

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. We outline empirical studies regarding code smells in this subsection.

Most previous studies have considered the negative effects of code smells. For example, Khomh et al. [20] found that classes with code smells are more likely to change and become faulty. Abbes et al. [21] 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 [22], who found that inter-smell relationships were related to maintenance problems. In addition, Yamashita and Moonen [23] explained that code smells can be used to partly reflect maintainability aspects of software. Sjøberg et al. [24], however, found that 12 of the code smells in their experiment did not significantly increase maintenance effort. Nevertheless, Soh et al. [25] revisited the same dataset to conduct a deeper investigation and found that different code smells, in fact, impact the effort of different maintenance activities. For instance, searching effort is affected by Feature Envy while Editing effort is affected by Data Clumps code smells.

In addition to studies into the effects of code smells, previous studies over the past decade have investigated how developers deal with code smells. Yamashita and Moonen [18] 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. [26] 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. [27] performed an exploratory survey to understand why code smells remain in source code and found that one of the reasons was concern about breaking client code. Peters and Zaidman [28] 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 considers how developers select and prioritize code smells. This gap causes some difficulty in the study of the techniques regarding code smell filtration and prioritization due to a lack of a clear direction for the research community to focus on. Thus, the aim of this study was to address this shortcoming and identify the issues that the research community should consider by identifying the factors that are used by professional developers.

2.2 Code Smell Prioritization and Filtration

As the large number of code smells detected by detection tools is one of the crucial challenges, many studies have been conducted to prioritize and filter the code smell detection results. This subsection summarizes recent studies on code smell prioritization and filtration.

Severity, describing how strong a code smell is, is one of the most popular factors used for prioritizing code smells. For instance, Marinescu defined severity as "Severities are computed by measuring how many times the value of a chosen metric exceeds a given threshold" [13]. Fontana et al. [11] also proposed the Intensity Index by calculating the distribution of software metrics. They represented the index as a numerical value between 1–10 and defined five intensity levels corresponding to each range: Very Low, Low, Mean, High, and Very High. This metric can be used to determine critical smells of the system.

Another approach that has been proposed to prioritize code smells is to use the context of developers. We previously defined the Context Relevance Index by applying an impact analysis technique to a list of issues in an issue tracking system and prioritized code smells that were most likely to be related to the context of developers [14], [29].

Furthermore, many approaches have been proposed by using not only one, but multiple factors for code smell prioritization. Arcoverde et al. [15] presented an approach using four heuristics: change density, error density, anomaly density, and architecture role, to prioritize code smells. Vidal et al. [16], instead, used three criteria: historical information of component modification, the relevance of the type of code smell from the developer's perspective, and modifiability scenarios of the system, to perform semi-automated prioritization of code smells. In our previous work [17], we also proposed a prioritization technique using the combination of code smell severity and the Context Relevance Index.

In addition to code smell prioritization, many studies have been conducted to filter code smell detection results to increase the accuracy of existing tools. Fontana et al. [12] considered the application domain of the system to calculate strong and weak filters. They used a strong filter to remove false positive code smells from detection results and a weak filter to identify code smell instances that are not likely to be problematic. In contrast, Ratiu et al. [30] used historical information to measure the stability of each code smell and filter out the instances that might not be harmful the system.

Although many approaches have been proposed, they use widely varying factors to prioritize and filter code smells. This may provide us with a hint that there is a lack of direction with respect to approach. In other words, researchers and tool developers do not actually realize the factors that should be considered in code smell filtration and prioritization. An underlying reason might be a lack of studies focusing on factors used by practitioners. However, in order to improve the usability and practicality of research tools in our field, the factors used by practitioners need to be considered when proposing new techniques. Therefore, the empirical evidence from this study should be able to provide an opportunity to support the advancement of such approaches by guiding the community to re-focus on the direction that practitioners need.

3. Study Design

As discussed earlier, the main purpose of this study is to identify the factors that practitioners use in the code smell filtration and prioritization process. The result of this study would allow researchers and tool developers improve code smell-related techniques such as filtration and prioritization by considering the use of the identified factors in the proposed techniques. As a result, the practicability and usability of research tools can be improved.

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 [31], [32]. 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 [17] on the code smells filtered by developers before working on specific tasks. While our previous work focused only on the code smell filtration process, this work also analyzed the code smell prioritization process used by developers. In addition, we collected the reasons for their decisions and included the results of their analyses in this study. The details of our study are explained in the following.

3.1 Research Questions

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

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

In the first question, the filtration 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 filtration process 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 filtration process, which could facilitate more practical approaches to code smell prioritization by focusing on the key problems.

In order to answer both research questions, we aim to quantify the following three aspects.

1. **Factors used by developers:** As the main purpose of this study is to understand the factors that developers use to handle code smells, considering the number of occurrences of each factor would allow us to see the ranking and distribution of the factors used by the developers.
2. **Combination of multiple factors used by developers:** As discussed in Sect. 2, previously proposed techniques used not only one, but multiple factors to prioritize code smells. Therefore, considering the number of factors that are considered together would enable us to identify the practitioners' need and verify whether the research community's direction is aligned with it.
3. **Effect of developers' experience:** In the empirical study regarding software development, the experience of developers plays a particularly important role, e.g., developers with more experience tend to make a decision differently than developers with less experience. Thus, analyzing the result by classifying the experience of the subjects would enable us to understand if there are any differences in the way more experienced developers handle code smells when compared to less experienced developers, i.e., there might be some factors that are considered by more experienced developers but not by less experienced developers and vice versa.

3.2 Data Collection

The data in this study were obtained from an extension of our previous study [17]. Our previous study dealt 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 ten 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 filter code smells that they considered should be refactored after considering all the information.

Our previous work [17] 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 filtration process, whereas this work also investigated the code smell prioritization process. Therefore, we added the following experimental tasks for our subjects and performed further investigations by: 1) gathering more concrete evidence of why the subjects filtered or did not filter a specific code smell; and 2) asking the subjects to prioritize the code smells that they filtered 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 filtration and prioritization of code smells.

3.3 Data Analysis

After obtaining the results from the participants, we used a coding technique from grounded theory [33] to analyze the results because it is suitable for studying human aspects of software engineering [34]. In addition, this technique has been used widely in software engineering research, including studies of code smells [18]. The initial codes were generated first for the analysis. The codes were not fixed or limited, so they could be modified, added, or deleted if 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 of the results of the coding process. The codes used in this study were extracted by analyzing the responses of the subjects. So, some of them might be the same terms as the ones used in the literature while some of them might be different. Some examples of responses and their corresponding codes are given in Table 1. 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 filter a specific code smell. At the end of the process, the investigators combined closely related codes and finally obtained 15 codes. The final 15 codes can be categorized into five groups based on the lexical or semantic characteristics of each code, e.g., the codes containing the word *smell* were categorized into the group titled *Smell* and the codes containing the word *task* were categorized into group titled *Task*. However, the codes *Testability*, *Readability*, *Maintainability* and *Understandability* do not have a word in common but all of them are aspects of software quality. Therefore, they were categorized into a group titled *Quality*. The complete list and the explanation of each code can be found in Table 2.

Table 1 Examples of responses and 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.	Task implementation cost
Util Class is a centric class. This class was invoked by many classes, so this class should be fixed first.	Module importance

Table 2 Final codes

Group	**Code**	**Description**
Smell	Smell severity	The subject states whether the code smell is severe or not severe.
	Smell coupling	The selected smell is related to another smell and should be solved together.
	Co-located smells	Multiple code smells appear in the same module and should be solved at the same time.
	Smell false positive	The subject does not consider that the result obtained by the code smell detector is a code smell.
Task	Task relevance	The smell either is related or not related to the subject's task, i.e., it appears in the module that the subject needs to modify to solve the task.
	Task importance	The related task is or is not important.
	Task implementation cost	The cost incurred for implementing a specific task is high or low.
	Task implementation risk	The risk of implementing a specific task is high or low.
Quality	Testability	The module is difficult to test, or the subject wants to improve the testability.
	Readability	The module is difficult to read, or the subject wants to improve the readability.
	Maintainability	The module is difficult to maintain, or the subject wants to improve the maintainability.
	Understandability	The module is difficult to understand, or the subject wants to improve the understandability.
Module	Module importance	The module plays an important role in the system, e.g., it is a main class that control business logic of the system.
	Module dependency	The module is dependent on another module and should be refactored in a specific order.
Refactoring	Refactoring cost	The cost incurred by performing refactoring operations to remove a code smell is high or low.

After obtaining the final codes, we have conducted follow-up interviews with three of the subjects to validate them. We provided the subjects' original responses, the codes and descriptions that we assigned in each response. Then, we asked them to confirm whether our codes and descriptions were aligned with their intention. All subjects in our follow-up studies confirmed and agreed with all codes that we assigned to their responses. In addition, because the factors identified might be specific to only the project used in this study, we have also asked the subjects whether the codes identified in this study were useful for filtering and prioritizing code smells in general. All of the subjects concurred.

To answer the first aspect (factors used by developers) and second aspect (combination of multiple factors used by developers) of each research question, we counted the number of codes and the number of each pair of codes that appeared together more than once respectively. As discussed earlier, one response may contain multiple codes as the subjects may have used multiple reasons to select or prioritize code smells. We then used the number of responses containing a specific code to answer each research question. As for the third aspect (effect of developers' experience), we classified the subjects with less than five years' experience as junior developers and the subjects with more than or equal to five years' experience as senior developers. As a consequence, each group comprised of five developers equally. We did not use the position that the subjects specified at the beginning of the experiment because it might be difficult to compare among different companies. We then analyzed the number of codes mentioned by each group to see the differences.

4. Analysis of Results

4.1 RQ1: What are the Factors Used by Developers in the Code Smell Filtration Process?

4.1.1 Factors Used by Developers

Table 3 shows the factors used by developers when filtering code smells according to our analysis. Column *#All* in the table represents the number of responses from all subjects of each code. For example, there are 33 responses from the subjects containing content identified as *Task relevance* and 11 responses from the subjects containing content identified as *Smell severity* in the first and second row respectively. Each response represents the input from the subject when selecting and prioritizing a specific code smell. Clearly, *Task relevance* was the most common factor used in the filtration process, where developers tended to filter 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 [31], [32].

Table 3 Factors used by developers in the code smell filtration process

Code	#All[1]	#Junior[2]	#Senior[3]
Task relevance	33	19	14
Smell severity	11	6	5
Task implementation cost	6	3	3
Testability	5	5	0
Co-located smells	4	1	3
Module importance	2	1	1
Readability	2	2	0
Smell false positive	2	1	1
Smell coupling	2	0	2
Maintainability	1	1	0
Understandability	1	1	0

[1] Number of responses from all subjects
[2] Number of responses from junior subjects
[3] Number of responses from senior subjects

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 filtered 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 filtering 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 filtered 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.

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

4.1.2 Combination of Multiple Factors Used by Developers

As mentioned earlier, previous studies often used multiple factors to detect and filter code smells, so we conducted further analysis on the factors considered together when devel-

Table 4 Combination of multiple factors used by developers in the code smell filtration process

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

opers filtered code smells, thereby obtaining insights into the factors that should be used together when detecting or filtering 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 4 represent the pairs of factors that the subjects considered together when filtering 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 filtered 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 4 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*.").

To sum up, *Task relevance* and *Smell severity* were the most common factors that were considered together by developers in the code smell filtration process.

4.1.3 Effect of Developers' Experience

As discussed earlier, since the experience of developers may affect the way they filter code smells, we also analyzed the codes classified by the years of experience of the subjects. The column *#Junior* and *#Senior* of Table 3 represents the number of responses from junior and senior subjects of each code respectively. The result from our analysis showed that *Task relevance* and *Smell severity* were still the most popular factors even though we considered each group individually. Although there were factors that senior developers considered but did not appear in junior developers' responses (e.g., *Smell coupling*) and vice versa (e.g., *Testability*), we found that such differences were minor. We concluded that there is no significant difference between how junior and senior developers filter code smells.

To conclude, the experience of developers was unlikely to affect the way they filter code smells.

Table 5 Factors used by developers in the code smell prioritization process

Code	#All[1]	#Junior[2]	#Senior[3]
Module importance	14	6	8
Task relevance	10	7	3
Testability	5	4	1
Smell severity	4	1	3
Maintainability	3	3	0
Refactoring cost	3	0	3
Co-located smells	2	1	1
Module dependency	2	0	2
Readability	2	2	0
Task implementation cost	2	2	0
Task importance	2	2	0
Task implementation risk	1	0	1

[1] Number of responses from all subjects
[2] Number of responses from junior subjects
[3] Number of responses from senior subjects

4.2 RQ2: What are the Factors Used by Developers in the Code Smell Prioritization Process?

4.2.1 Factors Used by Developers

Table 5 shows the factors used by developers in the code smell prioritization process. The column *#All* in the table represents the number of responses from all subjects of each code. In contrast to the factors used for code smell filtration, *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.

In summary, *Module importance* was the most common factor used in code smell prioritization process and the second was *Task relevance*.

Table 6 Combination of multiple factors used by developers in the code 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

4.2.2 Combination of Multiple Factors Used by Developers

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 combination of multiple factors used 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 filtration 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 6. According to these results, *Module importance* and *Task relevance* were the most common combination of multiple factors used 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.*").

To conclude, *Module importance* and *Task relevance* were the most common factors that were considered together in the code smell prioritization process.

4.2.3 Effect of Developers' Experience

Similarly to RQ1, we analyzed the results by classifying according to junior and senior developers. Columns *#Junior* and *#Senior* of Table 5 show the number of responses from junior and senior subjects of each code, respectively. The results were similar to those in RQ1 in the sense that there was no significant difference between both groups. The most common factors for both groups were still *Module importance* and *Task relevance*. While there were some cases that junior and senior developers used different factors for prioritizing code smells, we found such differences insignificant and concluded that the factors used for prioritizing code smells are unlikely to be affected by developers' experience.

In conclusion, developers' experience did not tend to affect how they prioritize code smells.

4.3 Implications

Table 7 shows the results of our survey regarding the factors identified in this study that have been considered in the literature. The scope of our survey was limited to the code smell prioritization and filtration approach. Code smell prioritization uses the result of the code smell detection approach as one of the inputs and generates a list where code smells are sorted by specific criteria while code smell filtration reduces the number of code smells based on specific criteria. We put a checkmark in the cell where the factor in each row was used in the approach in each column. It is noteworthy that we compared the definition of the factor of each work with the one in this study. Therefore, the term used in the original paper may not be exactly the same as the one defined in this manuscript. Furthermore, we list the factors that were used by work in the literature but were not identified in this study (e.g., *Change density*, *Error density*) in group *Others*.

The results showed that some of the factors identified in this study (*Smell severity*, *Co-located smells*, *Smell false positive*, *Task relevance*, and *Module importance*) were used in code smell prioritization and filtration approaches in the literature. This suggests that a part of the literature has used the factors that align with practitioners' needs. Therefore, we encourage the research community to continue focusing on the factors for future development, thus improving the practical utility of these research tools.

However, the results also show that existing techniques have paid most attention to factors in group *Smell*. In addition, out of the 15 factors identified in this study, ten of them have not been considered in the literature. This indicates that there are still many factors that need to be considered by the research community. For instance, even though factors in group *Task* have been considered in recent studies, the only factor that has been considered is *Task relevance*. In contrast, the factor *Task implementation cost* has never been considered although it was the third most common factor in the code smell filtration process identified in this study. In addition, factors in group *Quality* and *Refactoring* have never been taken into account even though they are closely related to code smells. We strongly recommend considering these factors when developing new techniques such as code smell filtration and prioritization.

Lastly, although there are some techniques that have proposed the combination of multiple factors when prioritizing code smells, there is still room for improvement in this aspect. For example, the combination of *Module importance* and *Task relevance* has never been proposed in spite

Table 7 Factors identified in this study that have been considered in the literature

		Prioritization						**Filtration**	
Group	**Code**	Marinescu [13]	Arcoverde et al. [15]	Fontana et al. [11]	Sae-Lim et al. [14]	Vidal et al. [16]	Sae-Lim et al. [17]	Ratiu et al. [30]	Fontana et al. [12]
Smell	Smell severity	✓		✓			✓		
	Smell coupling								
	Co-located smells		✓						
	Smell false positive								✓
Task	Task relevance				✓	✓	✓		
	Task importance								
	Task implementation cost								
	Task implementation risk								
Quality	Testability								
	Readability								
	Mantainability								
	Understandability								
Module	Module importance		✓						
	Module dependency								
Refactoring	Refactoring cost								
Others[1]	Change density		✓			✓		✓	
	Error density		✓						
	Smell type					✓			
	Smell persistence							✓	

[1] Other factors that were not identified in this study.

of being the most common combination of multiple factors used in the code smell prioritization process. Therefore, we encourage the development of new techniques that consider the combinations observed from this study.

5. 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. For example, our results in Table 7 shows that some existing techniques used factors such as *Change density* or *Error density* to prioritize or filter code smells. However, because this study did not include historical information, e.g., version control system or issue tracking system, we may have failed to identify such factors that rely on historical information. 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 filter code smells that are indirectly impacted by changes. However, conducting this type of study with the project that the subjects are working on is not practical due to the fact that we need to access to their source code and issue tracking system. Additionally, conducting a study regarding code smells with industrial developers using open source software source code is not uncommon (e.g., [26]). Inviting core developers of open source projects could become a part of our future work, but a low response rate is expected [26].

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 filtered or prioritized code smells. However, all subjects in our follow-up studies confirmed and agreed with all codes that we assigned to their responses. While only 10 codes were in this follow-up study, which does not cover all the codes identified in this study, we believe this threat should be minimized.

Another construct validity might depend on the validity of the answer that the subjects made, i.e., the subjects might have randomly selected or prioritized code smells without consideration. However, as our study design forced the subjects to state the reasons for the performed action, i.e., the reasons for selecting or prioritizing particular code smells, we believe this threat is mitigated. In addition, because we mainly focused on the reasons behind the selected or prioritize code smells in this study, the validity should depend on the reasons that the subjects state and not the actions that they performed, i.e., if the subjects did not state the reasons, the result will be invalid. However, such cases did not occur during our study.

Finally, 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. However, 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. Therefore, the factors identified in this study might be specific to only this project. However, in our follow-up studies, all of the subjects agreed that the codes identified in this study were useful for filtering and prioritizing code smells in general. This may indicate that the identified codes are not specific to the project used in this study. Nevertheless, a larger scale study might be beneficial.

6. Future Work

Our future work includes two parts. As discussed earlier, because we may have missed some details in this study, the first task is to conduct follow-up interviews with some of the subjects to obtain a more specific opinion on how they filter and prioritize smells. The second part of our future work is to propose code smell filtration and prioritization techniques based on the factors identified in this study that have not been considered by the research community as discussed in Sect. 4.3.

7. Conclusion

In this study, we conducted an experiment to determine how professional developers filtered 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 filtration, followed by *Smell severity*, and both were also often considered together during the filtration 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] N. Sae-Lim, S. Hayashi, and M. Saeki, "How do developers select and prioritize code smells? a preliminary study," Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17), pp.1–5, 2017.

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

[3] W.C. Wake, Refactoring Workbook, Addison-Wesley, 2003.

[4] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), pp.350–359, 2004.

[5] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F.L. Meur, "DECOR: A method for the specification and detection of code and design smells," IEEE Transactions on Software Engineering, vol.36, no.1, pp.20–36, 2010.

[6] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08), pp.329–331, 2008.

[7] M.J. Munro, "Product metrics for automatic identification of "Bad Smell" design problems in java source-code," Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05), p.15, 2005.

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

[9] 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," Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13), pp.268–278, 2013.

[10] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16), pp.1–10, 2016.

[11] F.A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," Proceedings of the IEEE Seventh International Workshop on Managing Technical Debt (MTD'15), pp.16–24, 2015.

[12] F.A. Fontana, V. Ferme, and M. Zanoni, "Poster: Filtering code smells detection results," Proceedings of the 37th International Conference on Software Engineering (ICSE'15), pp.803–804, 2015.

[13] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," IBM Journal of Research and Development, vol.56, no.5, pp.9:1–9:13, 2012.

[14] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based code smells prioritization for prefactoring," Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16), pp.1–10, 2016.

[15] R. Arcoverde, E. Guimaraes, I. Macia, A. Garcia, and Y. Cai, "Prioritization of code anomalies based on architecture sensitiveness," Proceedings of the 27th Brazilian Symposium on Software Engineering (SBES'13), pp.69–78, 2013.

[16] S.A. Vidal, C. Marcos, and J.A. Díaz-Pace, "An approach to prioritize code smells for refactoring," Automated Software Engineering, vol.23, no.3, pp.501–532, 2016.

[17] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based approach to prioritize code smells for prefactoring," Journal of Software: Evolution and Process, 2017. doi: 10.1002/smr.1886.

[18] A. Yamashita and L. Moonen, "Do developers care about code smells? An exploratory survey," Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13), pp.242–251, 2013.

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

[20] 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," Empirical Software Engineering, vol.17, no.3, pp.243–275, 2012.

[21] 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," Proceedings of the 15th European conference on Software maintenance and reengineering (CSMR'11), pp.181–190, 2011.

[22] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," Proceedings of the 35th International Conference on Software Engineering (ICSE'13), pp.682–691, 2013.
[23] A. Yamashita and L. Moonen, "To what extent can maintenance problems be predicted by code smell detection? – an empirical study," Information and Software Technology, vol.55, no.12, pp.2223–2242, 2013.
[24] D.I.K. Sjøberg, A. Yamashita, B.C.D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," IEEE Transactions on Software Engineering, vol.39, no.8, pp.1144–1156, 2013.
[25] Z. Soh, A. Yamashita, F. Khomh, and Y.-G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?," Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16), pp.393–402, 2016.
[26] 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," Proceedings of the IEEE international conference on Software maintenance and evolution (ICSME'14), pp.101–110, 2014.
[27] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," Proceedings of the 4th Workshop on Refactoring Tools (WRT'11), pp.33–36, 2011.
[28] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12), pp.411–416, 2012.
[29] N. Sae-Lim, S. Hayashi, and M. Saeki, "Revisiting context-based code smells prioritization: On supporting referred context," Proceedings of the XP2017 Scientific Workshops (XP'17), pp.3:1–3:5, 2017.
[30] D. Ratiu, S. Ducasse, T. Girba, and R. Marinescu, "Using history information to improve design flaws detection," Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR'04), pp.223–232, 2004.
[31] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," Journal of Systems and Software, vol.107, pp.1–14, 2015.
[32] D. Silva, N. Tsantalis, and M.T. Valente, "Why we refactor? Confessions of GitHub contributors," Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16), pp.858–870, 2016.
[33] A. Strauss and J. Corbin, Basics of qualitative research: Procedures and techniques for developing grounded theory, Thousand Oaks, CA: Sage, 1998.
[34] R. Hoda, J. Noble, and S. Marshall, "Using grounded theory to study the human aspects of software engineering," Proceedings of the Second Workshop on Human Aspects of Software Engineering (HAoSE'10), pp.5:1–5:2, 2010.

Natthawute Sae-Lim received a B.E. degree in computer engineering from King Mongkut's Institute of Technology Ladkrabang in 2012. He received an M.E. in computer science from Tokyo Institute of Technology in 2016. He is currently a doctoral student of department of computer science at Tokyo Institute of Technology. He had worked for Thomson Reuters before joining Tokyo Institute of Technology. His research interests include empirical software engineering, mining software repositories, and code smells.

Shinpei Hayashi received a B.E. degree in information engineering from Hokkaido University in 2004. He also respectively received M.E. and D.E. degrees in computer science from Tokyo Institute of Technology in 2006 and 2008. He is currently an associate professor of computer science at Tokyo Institute of Technology. His research interests include software changes and software development environment.

Motoshi Saeki respectively received a B.E. degree in electrical and electronic engineering, and M.E. and D.E. degrees in computer science from Tokyo Institute of Technology, in 1978, 1980, and 1983. He is currently a professor of computer science at Tokyo Institute of Technology. His research interests include requirements engineering, software design methods, software process modeling, and computer supported cooperative work (CSCW).