Received: 30 September 2016 | Revised: 17 May 2017 | Accepted: 24 May 2017

DOI: 10.1002/smr.1886

WILEY Journal of Software: Evolution and Process

SPECIAL ISSUE PAPER

Context-based approach to prioritize code smells for prefactoring

Natthawute Sae-Lim | **Shinpei Hayashi** | **Motoshi Saeki**

School of Computing, Tokyo Institute of Technology, Tokyo, Japan

Correspondence
Natthawute Sae-Lim, School of Computing, Tokyo Institute of Technology, Tokyo, Japan.
Email: natthawute@se.cs.titech.ac.jp

Funding information
JSPS Grants-in-Aid for Scientific Research, Grant/Award Number: JP15K15970, JP15H02685, JP15H02683

Existing techniques for detecting code smells (indicators of source code problems) do not consider the current context, which renders them unsuitable for developers who have a specific context, such as modules within their focus. Consequently, the developers must spend time identifying relevant smells. We propose a technique to prioritize code smells using the developers' context. Explicit data of the context are obtained using a list of issues extracted from an issue tracking system. We applied impact analysis to the list of issues and used the results to specify the context-relevant smells. Results show that our approach can provide developers with a list of prioritized code smells related to their current context. We conducted several empirical studies to investigate the characteristics of our technique and factors that might affect the ranking quality. Additionally, we conducted a controlled experiment with professional developers to evaluate our technique. The results demonstrate the effectiveness of our technique.

KEYWORDS

code smell, impact analysis, issue tracking system, prefactoring

1 | INTRODUCTION

Code smell, as defined by Fowler,[1-3] is an indicator of design flaws or problems in the source code. Code smells of many types are summarized as smell catalogs.[1-3] Code elements that are affected by code smells are recommended for improved quality because they are likely to cause difficulties in the future, as discussed in the literature, eg, making the source code more difficult to understand and maintain or increasing code components' fault-proneness. Moreover, Yamashita et al[4] found that code smells can affect important maintainability aspects. Hermans and Aivaloglou[5] conducted a controlled experiment, which revealed that code smells influence the block-based programming performance of novice programmers. Developers can improve the code by removing the code smells from the elements.

Many researchers have examined factors that can introduce code smells into the system. Tufano et al[6] conducted a large-scale empirical study, which revealed that the main activities that tend to introduce code smells are implementing features and enhancing existing ones. Moreover, they found that developers with high workloads and pressure are more likely to introduce a smelly code to the system. Vale et al[7] investigated the factors that can influence code smells based on a software developer's perspective. That investigation revealed that factors such as time pressure, bad design decisions, lack of business knowledge, inexperienced developers, little time for refactoring, and low priority assigned to software quality influence the severity of smelly codes.

For issue-driven software development projects, development teams tend to adopt an issue tracking system such as Jira or Bugzilla to manage their lists of issues, so-called backlogs. Such issues can be anything related to software change requests, eg, bug fixing or feature implementation. Developers apply refactoring techniques to source codes before implementing changes according to issues during the so-called prefactoring phase.[8] Given those assessments, modules that are relevant to developers' tasks are likely to have higher priority for fixing of code smells and improvement of code quality over others because fixing their code smells might contribute to the support of future implementation, for instance, by improving the understandability and extensibility of the source code. Consequently, such modules are suitable to be regarded as their context. As described in this paper, we use the definition of *context* similarly 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."[9] Because the tasks are described as issues and are managed in an issue tracking system, we use the information in the issue to estimate the context.

Nevertheless, most existing techniques for detecting code smells and recommending refactoring opportunities ignore such a context and output the results only from analyzing the whole source code without prioritizing the results or prioritizing them with factors that are unrelated to the developer context. Therefore, the time-consuming process of identifying relevant outputs is left to the developers, which is similar to the fact that static analysis tools are not used well by developers because of numerous detected warnings.[10] One example is the severity-based prioritization, which ranks first the most severe code smells.[11-13] This scheme is widely adopted when a developer wants to improve the overall system quality because solving code smells that contribute most to the decay of the source code is likely to be more useful than solving those that have little effect on the system. However, presuming the prefactoring phase, the developer has a specific context that he or she must work on. In this situation, solving the most severe code smell might not be the best strategy because solving such smells might not contribute directly to the support of developers' implementation. In contrast, solving code smells according to context-based detection or prioritization might provide better alternatives because solving context-related smells is likely to facilitate the implementation of developers. Furthermore, in a real-world situation, because development teams have limited time for delivering the product, and very little for improving the source code quality, it would be challenging to have permission from management for improving the overall software quality. However, in the prefactoring phase, the developer can use a part of the normal implementation time to improve the quality of only those modules that the developer is going to work on. This paper therefore particularly addresses the importance of context-based prioritization rather than severity-based prioritization to reflect the real-world situation in which the developer would have a limited amount of time to improve software quality.

Motivation. A clear necessity for a context-aware approach to code smell detection is apparent in recent research conducted in this area. Work done by Yamashita et al[14] seems to show that developers need code smell detectors that support context-sensitive or domain-specific strategies. Work done by Bavota et al[15] suggests that the developer's perspective should be considered when proposing refactoring recommendations. Many studies have emphasized the importance of this prefactoring phase. Meng et al[16] reported that developers do not typically refactor their code unless they must change the code to fix some bugs or introduce new features. Bavota et al[15] supported this statement by asserting the possibility that the only developer goal of refactoring is to prepare a source code for future changes. In addition, Silva et al[17] conducted a large-scale study on the motivations underlying refactoring. Their results show that developers perform refactoring activity based mainly on change requirements. In other words, developers do refactoring to facilitate the maintenance task that they are working on. On the basis of their study, they also suggested that research on refactoring recommendation should specifically examine maintenance task–oriented rather than code smell–oriented solutions. In maintenance tasks, developers specifically examine the modules to be maintained, ie, contexts for the maintenance task. Their work also suggests the motivation of context-based approaches.

Proposed solution. We propose a technique to prioritize code smells from code smell detectors by considering developers' current context. This technique specifically examines support of the *prefactoring* phase.[8] During the prefactoring phase, developers facilitate implementation by improving source code extensibility and understandability by refactoring the source code before implementing features. Using the proposed technique, we used change information in an issue tracking system that is to be implemented using a particular milestone to estimate the developers' context. Such changes are to be implemented by modifying or extending existing modules in the source code. These existing modules can be located using impact analysis.[8,18] We regard such relevant modules as their context because they are likely to be the location for implementation of new features in the change information. As a result, code smells that appear in relevant modules should be assigned higher priority so that the developers can readily distinguish them from irrelevant code smells. With support from our technique, developers can obtain a prioritized list of code smells based on their relevance to the context.

Evaluation scheme. We conducted empirical studies to assess characteristics of our technique as well as factors that might affect the results obtained using our technique. We also conducted a controlled experiment with professional developers to evaluate our technique. The results demonstrate the effectiveness of our technique.

Contribution. The main contributions of this paper are the following.

1. We propose a technique for prioritizing code smells using developers' context.
2. We define a context relevance index (CRI) based on automated impact analysis to use as a criterion to prioritize code smells.
3. We present empirical studies of the characteristics of this approach and factors that can affect the quality of the ranking results.
4. We present a controlled experiment showing that our technique can prioritize first code smells chosen to be prioritized by professional developers.

This paper is an extension of work reported originally in the Proceedings of the 24th IEEE International Conference on Program Comprehension.[19] The main differences between this and the earlier work are the following:

- We conducted a deeper investigation of the effect of impact analysis accuracy on our technique (research question [RQ]2); ie, more types of impact analysis are considered in this paper.
- We added a study of the combination of severity-based and context-based approaches (RQ4).
- Whereas an earlier work was based solely on an existing dataset, this paper also includes a controlled experiment with professional developers (RQ6 and RQ7).*

* In our previous paper,[19] there are a few minor errors caused by minor bugs in our implementation. We have alleviated them and updated the results in this paper. The changes are in RQ1 (normalized discounted cumulative gain [nDCG] value), RQ2 (Spearman correlation), RQ3 (ranking items), and RQ5 (nDCG value, mentioned in section IV-B in the previous paper). The errors do not affect the conclusion of the RQs in the previous paper.

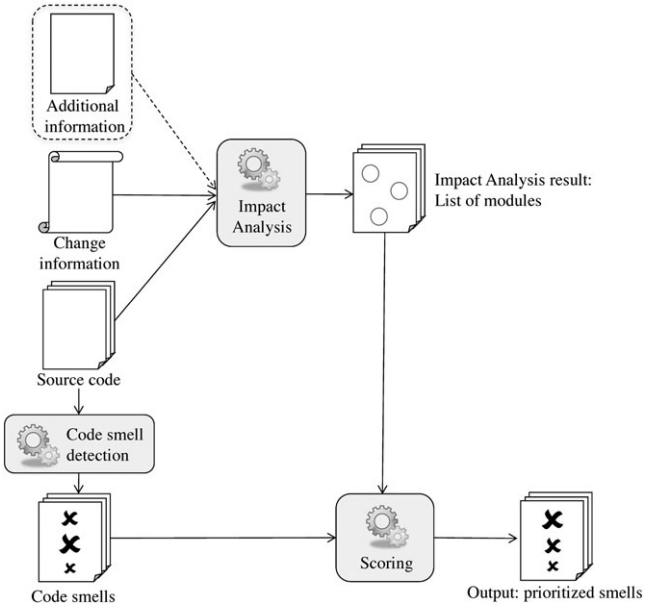

FIGURE 1 Overview of the proposed technique

Paper structure. The remainder of this paper is organized as described below. The next section presents our approach and its implementation. Section 3 presents RQs of this paper. Our empirical studies using existing datasets are presented in Section 4. We then show our controlled experiment with professional developers in Section 5. We summarize related work in Section 6. Finally, Section 7 concludes this paper and provides additional directions of research that should be explored.

2 | PROPOSED TECHNIQUE

2.1 | Framework for context-based code smell prioritization

We propose a technique for prioritizing code smell detection results from existing code smell detectors by considering the list of issues in the issue tracking system that developers must solve. Figure 1 presents an overview of our technique. Each gray node represents a subprocess of our technique. The input of the process is a list of change information obtained from the issue tracking system, the source code of the targeted project, and additional information for impact analysis such as the software repository. The output is a prioritized list of code smells based on their relevance to the developer context. Our approach first uses impact analysis to obtain a list of modules that are likely to be the targeted modules of each change information. Next, we generate a list of code smells by application of an existing code smell detector with the target project source code. Then, for each code smell in the list, we calculate the CRI based on the relevance of the list of modules from impact analysis. Finally, we output the prioritized list of code smells ordered by the CRI value.

We design our approach as a framework within which code smell detection and impact analysis are its hot spots. Our prioritization technique works by connection with existing impact analyses and code smell detection techniques and generates the results. In this sense, this framework is general and is independent of specific types of code smell detection technique or impact analysis. The following subsections present an explanation of these techniques and their application.

2.1.1 | Impact analysis

During software development processes, developers can modify some modules to satisfy a change request, such as one for bug fixing or feature implementation. Developers might first use their experience, system knowledge, or technique such as feature location[18] to identify at least 1 module that is relevant to the change request. Then, they perform impact analysis to specify the full impact set, where the system is likely to be affected by such changes.[8,20]

Impact analyses of various types require different inputs such as natural language query, execution scenario, and source code artifacts.[18] Gethers et al[21] proposed a combination technique to perform impact analysis depending on the source of contextual information available to each software project, such as information retrieval (IR), mining software repository (MSR), or dynamic analysis (Dyn).

Figure 2 presents an example of an issue in the issue tracking system of the jEdit (http://www.jedit.org/) project. We consider the information in the *Summary* together with the *Description* field as change information because they contain important details related to the change such as problems, reproduction steps, or even solutions. Therefore, this information is likely to be useful with impact analysis to identify source code components

Summary:	Autosave turned off for Untitled Documents.
Description:	I have an issue where sometimes untitled documents get left on to disk. I think it's an issue when the app closes improperly in some way. An option to never save untitled documents would be a nice feature and would solve this issue.

FIGURE 2 Portion of an issue in the issue tracking system

related to developers' context. We exclude comment fields in our approach attributable to the fact that it is not always available at the beginning of each release. They might be generated in later stages of software development such as when a developer starts working on the issue and need clarification. Change information can be extracted using the application programming interface of each issue tracking system or text mining tools such as Kurya by Kohan et al,[22] which is tailored specifically for mining issue tracking systems. Their tool can extract up to 3.5 documents per second at a network speed of 50 Mbps for responsive resources.

In our technique, we chose impact analyses that take change information i and source code C as inputs and provide a set of modules $M = \{\dots, m, \dots\}$ with their probability score as outputs because we specifically examine the support of issue-based software development projects in which developers tend to implement features or fix bugs by following the change information in an issue tracking system. Therefore, we can use the change information as inputs of impact analysis to identify a set of modules that are likely to be modified to achieve the change. Consequently, applying the prefactoring technique to these modules is likely to support the developers' implementation by improving the understandability or extensibility of the source code.

As described in this paper, we input a set of change information $I = \{i_1, \dots, i_n\}$ and source code C to the impact analysis and obtain a series of sets of modules $\{M_1, \dots, M_n\}$ together with its score indicating the relevance between the change information and the module. As described herein, modules can be either classes or methods.

2.1.2 | Code smell detection

Code smell detection generates a list of code smells from the targeted source code. One example of the approaches is to detect them based on particular metric values such as lines of code. The input is the source code that we wish to analyze. The output is a list of smells. Each smell consists of attributes $\langle type, entity, granularity, severity\rangle$, where *type* stands for the type of the detected smell; *entity* signifies the module having the detected smell; *granularity* denotes the level of code smell consisting of the subsystem, class, and method (details of the difference between method-level and class-level code smells will be discussed in Section 4); and *severity* is an integer value representing the smell strength, for example, Marinescu defined as "Severities are computed by measuring how many times the value of a chosen metric exceeds a given threshold."[13] Table 1 presents an example of the code smell result from inFusion. For example, the second row shows the Blob Class smell in `Buffer` class with severity 7. We have omitted the package and class name of the method-level smell.

In this approach, we apply a code smell detector and obtain the list of smells S.

2.1.3 | Scoring

To assign the priority of each smell, we define the CRI attribute. The value of the CRI attribute is calculated using the weighted summation of the number of the modules in the result from impact analysis that match each smell's entity. The CRI of each $s \in S$ is definable as

$$CRI(s) = \sum_{i=1}^{n} \sum_{m \in M_i} \begin{cases} w(m), & \text{if } match(m, entity), \\ 0, & \text{otherwise,} \end{cases}$$

where n stands for the number of considered issues and *entity* signifies the module having the detected smell, which can be a method, a class, or a subsystem. The predicate $match(m, entity)$ holds when module m is the same as or belongs to the *entity* of each smell. For example, if m is a method and *entity* having a code smell is also a method, then $match(m, entity)$ holds when m and *entity* are the same. For the case in which m is a method, but *entity* having a code smell is a class, $match(m, entity)$ holds when m is in the *entity* class. Finally, $w(m)$ stands for the probability score of a module, which is a result of impact analysis.

TABLE 1 Portion of code smell detector results

Type	Entity	Granularity	Severity
SAP Breakers	org/gjt/sp/jedit	Subsystem	6
Blob Class	org.gjt.sp.jedit.Buffer	Class	7
Feature Envy	buildDOM() : void	Method	10

2.2 | Implementing our approach

We have implemented an automated tool (https://github.com/salab/CodeSmellsPrioritizer) for use with the proposed technique. The chain is designed to connect with an existing impact analysis tool. When executed, our tool reads a file containing a list of code smells that were generated by a code smell detector and calculates the CRI of each smell based on the relevance of the result from impact analysis.

2.2.1 | Impact analysis

We consider an impact analysis as a hot spot of the framework. Therefore, it is not limited to any specific technique. As an implementation in this paper, we follow approaches described in a work by Gethers et al.[21] Their work was undertaken to integrate different impact analyses to improve the overall accuracy of the respective independent techniques but also showed that different impact analysis approaches yield different accuracies of the results. They discussed differences among combinations of these 3 techniques: IR, Dyn, and MSR. The next few paragraphs present overviews of the respective techniques we used.

Information retrieval, such as a vector space model, is done by representing documents and queries as vectors.[23] Then, the relevance of documents and queries is obtainable by calculating their mutual cosine similarity. For impact analysis, documents are source codes of a specific project. Queries are change requests from the issue tracking system. Results then show the relevance between the source code and change requests. As described in this paper, we follow Gethers et al in[21] using latent semantic indexing (LSI)[24] for IR-based impact analysis. Actually, LSI is based on a vector space model. It was created with the intention of handling the situation of synonymy, a group of words that share similar meanings, and polysemy, words that have multiple meanings. Specifically, LSI is used for assessment of the similarity between documents from the common words 2 documents contain rather than simple terms. The more words they have in common, the more similar they are. Details of the techniques we used can be found in an earlier report in the literature.[21]

Dynamic analysis uses the execution traces of the given program. Several technologies are available to collect the information, eg, Java Platform Debugger Architecture (https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/) or Test and Performance Tools Platform (https://projects.eclipse.org/projects/tptp.platform/). In some cases, the execution trace of the specified change request is attached by the submitter. However, developers themselves can obtain it by reproducing the steps specified in the change request as well. Such runtime information is useful to filter out the results from the IR technique because the modules that are not in the execution trace are unlikely to be affected by the change request. Therefore, it is useful as additional information in our technique.

Mining software repository is an approach to apply data mining techniques to software repositories such as version control systems. We specifically examine this technique on mining evolutionary coupling, ie, modules that tend to be modified together. Such a technique can be done using association mining; ie, the probability of *consequent Y* appearing when *antecedent X* appears ($X \Rightarrow Y$) is represented by *confidence* and *support* values. Practically speaking, when a developer does association mining, the developer will select a module to use as the starting point using the developer's own system knowledge or techniques such as feature location. Then, the MSR-based impact analyses will generate a list of modules that are often changed together with the starting point module. Tools such as ImpactMiner[25] are useful when using a software repository as additional information in our technique. The following paragraphs explain how we combined each technique.

Information retrieval and dynamic analysis In this combination, we first use the IR technique to obtain a list of modules that are likely to be related with the specified change information. Then, we use the result from Dyn to filter out the result from the IR technique. We remove the method in results from the IR if it is not in the result from Dyn because Dyn recorded the modules that have been executed during the operations. Therefore, modules that were not executed during the operations are unlikely to be related with the changes.

Information retrieval and mining software repositories The combination between IR and MSR can be done by first obtaining the list from each technique, as described earlier. Then, one must alternately merge the results from each list until the lists are exhausted. If the same module appears in both lists, then it will be selected only once, but the score value will be the accumulation from both lists.

Information retrieval, dynamic analysis, and mining software repositories According to Gethers et al,[21] this combination can alleviate the situation in which Dyn filters out the correct result from the IR list. Using the same heuristic as IR + MSR, this combination can be acquired by merging the results from IR + Dyn and MSR.

In summary, the techniques that we used are IR, the combination of IR + Dyn, the combination of IR + MSR, and the combination of IR + Dyn + MSR. For IR and IR + Dyn, we use tools proposed by Dit et al[26] together with a TraceLab-based solution[27,28] to compute LSI for IR impact analysis. Then, we filter out the unrelated modules using the execution trace available in the dataset, as explained later. Figure 3 portrays our configuration in TraceLab, which is based on a previous work.[28] For IR + MSR and IR + Dyn + MSR, we use ImpactMiner[25] to perform association mining for the MSR part. Then, we combine the results with IR and IR + Dyn as described earlier.

2.2.2 | Code smell detection

As discussed earlier, because we designed the approach as a framework, it is independent of specific code smell detection. As an implementation in this paper, for this study, we used inFusion version 1.9.0[29] (unfortunately, inFusion is no longer available) as a code smell detector because (1) it can detect code smells of 24 types, eg, Blob Class, Data Class, or Feature Envy; (2) all detected smells are associated with the severity score; and (3) its detection process can be assembled in an automated manner. These characteristics are suited to our approach.

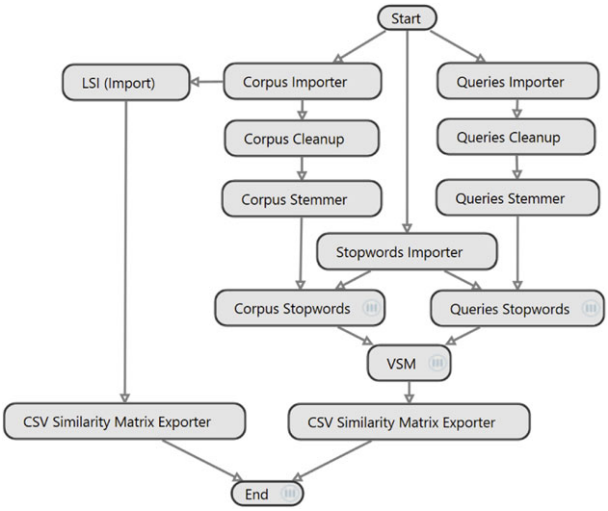

FIGURE 3 TraceLab configuration. LSI, latent semantic indexing; VSM, vector space model

TABLE 2 Target code smells detected by inFusion

Granularity	Type
Class	Blob Class[1,3,30]
	Data Class[1,3,31]
	Distorted Hierarchy[31]
	God Class[1,3,31]
	Refused Parent Bequest[3,31,32]
	Schizophrenic Class[31,32]
	Tradition Breakers[3,31]
Method	Blob Operation[1,3,30]
	Data Clumps[1]
	External Duplication[1,30,33]
	Feature Envy[1,3,31]
	Intensive Coupling[1,3,31]
	Internal Duplication[1,30,33]
	Message Chains[1,3,30]
	Shotgun Surgery[1,3]
	Sibling Duplication[1,30,33]

A list of code smells detected by inFusion that are considered in this paper, ie, the smells of class and method levels, together with each one's related literature are presented in Table 2.

2.2.3 | Scoring

Figure 4 represents an example of the CRI-calculating mechanism of the ArgoUML project using the IR + Dyn approach as an impact analysis. In this case, the God Class code smell in the `Project` class obtains a CRI value as high as 4.39 because the `Project` class appears on the top position in many results from impact analysis. For example, the impact analysis using the change information of issue #2500 predicts a `Project` class with a probability of 0.19, which is the third highest. On the other hand, the Blob Class code smell in the `GeneratorCSharp` class obtains only a 0.48 CRI value because the `GeneratorCSharp` class does not frequently appear in the top position of the results from impact analysis.

Using CRI to prioritize code smells instead of severity is likely to change the list characteristics. For instance, considering Figure 4, it is apparent that the Blob Class smell in the `GeneratorCSharp` class, which has severity of 8, is assigned the lower position on the context-based prioritized list. On the other hand, if one considers the code smell God Class of the class `Project` with severity 2, then this smell is assigned to the highest of the result in the context-based prioritized list, which indicates that even though it is not severe, our technique predicts that this smell is related to many issues associated with the issue tracking system of developers.

Therefore, when one prioritizes code smells using CRI, the most severe code smells might become least important. The least severe code smells might become most important depending on their relevance to the context of developers. The underlying reason is our assumption that solving smells with low severity but high relevance to the developers' context is better than solving smells with high severity but not relevant to the

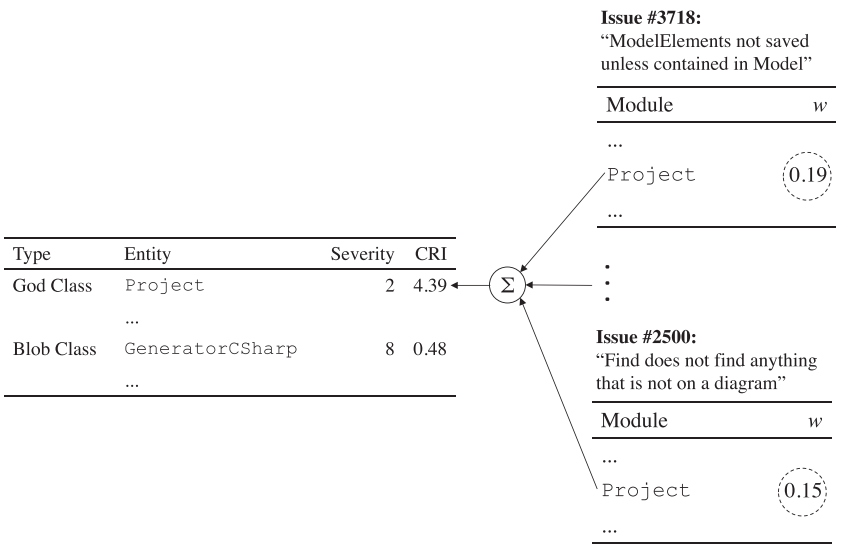

FIGURE 4 Example of a context relevance index (CRI)–calculating mechanism

developers' context. In the prefactoring phase, solving smells in the modules that developers are unlikely to touch is not going to support their implementation. However, solving smells for modules in which developers are likely to implement features or fix bugs, even though the code smells are not very severe, would facilitate their implementation by improving several factors such as understandability and extensibility. Details of the difference between severity-based and context-based prioritization will be discussed in Section 4.

3 | RESEARCH QUESTIONS

The main objectives of the studies described in this paper are 2-fold: (1) elucidating the characteristics of our approach and the factors that might influence the ranking results and (2) evaluating whether our approach can prioritize code smells that align with developers' needs in the prefactoring phase. We divide the RQs into empirical studies and controlled experiment. The next subsections explain the details together with their associated RQs.

3.1 | Empirical studies

In this study, we divided the studies into 2 parts: investigating factors affecting the performance of our approach and studying the meaningfulness and usefulness of the recommended prioritization. The first part is intended to study the factors that can affect our approach whereas the second part is designed to validate our approach and to analyze different opportunities to apply our technique. Particularly, we addressed the following RQs in each part.

3.1.1 | Investigating factors affecting the performance of our approach

- **RQ1:** *Which code smell granularity is more appropriate for our technique: coarse grained or fine grained?* Code smell detection tools often classify smell granularity as coarse grained or fine grained, ie, class level and method level. Solving code smells of both types is likely to improve the source code quality. Solving code smells in the module that a developer intends to modify would support its implementation. In the case of coarse-grained smells, solving the code smell in one class is likely to support the implementation under that class, including implementation of the methods of that class as well. However, in the case of fine-grained smells, solving the code smell in one method is likely to help only the implementation under that method, even though it is easier to understand how to solve the code smell. Therefore, we suspect that, in context-based prioritization, coarse-grained and fine-grained granularity code smells are expected to yield different ranking quality.
- **RQ2:** *Does the impact analysis accuracy affect the ranking quality?* As described in this paper, our proposed technique was designed to use existing impact analyses. However, as we discussed, impact analyses of many types have been proposed in the literature. Different techniques might use inputs or mechanisms of different types. Nevertheless, the main objective is to achieve accuracy that is as high as possible because accuracy is an important factor of impact analysis. Therefore, understanding how different impact analyses affect the ranking quality can enable us to find the appropriate impact analysis for context-based code smell prioritization.

3.1.2 | Studying the meaningfulness and usefulness of the recommended prioritization

- **RQ3:** *Does context-based smell prioritization provide more relevant results than severity-based smell prioritization?* The main objective of this study is to propose that context-based smell prioritization can be an efficient method for supporting the prefactoring phase. Therefore, we investigate this RQ to ascertain whether context-based smell prioritization produces more suitable results for supporting the prefactoring phase than the severity-based approach.
- **RQ4:** *What is the effect of the combination of severity-based and context-based prioritization?* In RQ3, we use context-based prioritization using the CRI value, with the expectation that code smells that are more related to the context would be at the top of the list. That is to say, the only factor that concerns the prioritization scheme is relevance to the context. However, the *severity* value, an extremely popular factor for prioritization, as we discussed, is also an important information related to each smell. Therefore, it would be useful if we were able to use not only the CRI but also *severity* to prioritize code smells to acquire the result where code smells with high relevance to the context *and high severity* are at the top of the list.
- **RQ5:** *Can our approach predict the smells to be refactored?* Although the aim of our technique is to *prioritize* code smells that developers should solve, one might have a question as to whether our technique can *predict* the smells that are going to be refactored by a developer, although such is not our intention. To answer this question, we conducted an empirical study to ascertain whether the provided ranking of smells by the proposed technique fits the modules to be refactored.

3.2 | Controlled experiment

The main goal of our approach is to prioritize code smells that can support a developer's prefactoring process. Therefore, we conducted a controlled experiment with professional developers with the aim of answering the following RQs:

- **RQ6:** *How does our technique prioritize code smells selected by professional developers in the prefactoring phase compared with a severity-based approach?* Because the objective of our approach is to put first code smells that developers should be solved during the prefactoring phase, it is important to evaluate whether the result of our approach is in accord with the developers' opinion.
- **RQ7:** *What is the most appropriate proportion of linear combination of severity-based and context-based prioritization for the prefactoring phase?* In RQ4, we study the usage of the linear combination between context-based and severity-based approaches. However, in this study, we want to find out the most appropriate proportion of context-based and severity-based approaches in the prefactoring phase.

4 | EMPIRICAL STUDIES

4.1 | Experimental setup

4.1.1 | Data collection

In this evaluation, 4 open-source projects, ArgoUML (http://argouml.tigris.org/), Jabref (http://www.jabref.org/), jEdit (http://www.jedit.org/), and muCommander (http://www.mucommander.com/), were our subjects because their data are available through the benchmark dataset of Dit et al.[18] The dataset includes lists of *Summary* and *Description* information, a list of executed methods, gold set methods, and methods that were actually modified to solve extractable issues in the issue tracking system, of each issue during the analyzed period. Table 3 presents information of our datasets including the size of the source code of the earlier version, the number of issues that we used between the 2 releases, and the number of smells detected by inFusion. For this study, we mined over 6000 commits between releases 0.14 and 0.22 of ArgoUML, over 1400 commits before release 2.0 of JabRef, over 900 commits between release 4.0 and 4.2 of jEdit, and over 1600 commits before release 0.8.0 of muCommander to conduct MSR-based impact analyses. We assumed a situation in which developers are working on upcoming releases where the modules that they must work on are regarded as their context. Consequently, the issues describing developers' task for the upcoming releases are used to estimate the context.

We first defined the oracle as a set of code smells occurring in the modules that were modified by developers during 2 releases according to the data in the benchmark dataset. For ArgoUML, we prepared the oracle by first applying the source code at version 0.22 to the code smell detector. Then we obtained the result. Next, we used the gold set methods from the benchmark dataset for versions 0.22 and 0.24. Finally, we intersected these 2 sets to obtain a list of smells that are actually related to the developer context. We applied the same process to JabRef versions 2.0–2.6, jEdit versions 4.2–4.3, and muCommander versions 0.8.0–0.8.5. The dataset used for this study can be downloaded from our website (http://www.se.cs.titech.ac.jp/data/JSEP-smells-prioritization/).

Regarding the baseline of our evaluation, we used the original result from inFusion sorted by the severity of the respective smells by application of a stable sort to the original result, which maintains the relative order of the results.

TABLE 3 Dataset information

Project	Version	Size (LOC)	No. of Issues	No. of Method-Level Smells	No. of Class-Level Smells
ArgoUML	0.22–0.24	309468	91	412	61
JabRef	2.0–2.6	72555	39	60	37
jEdit	4.2–4.3	140592	150	184	51
muCommander	0.8.0–0.8.5	88455	92	44	41

Abbreviation: LOC, lines of code.

4.1.2 | Selection of impact analysis techniques

For this study, except for RQ2, we used IR + Dyn for analysis because we wanted to reflect the real-world situation in which the availability of information is limited. In other words, the change information and execution trace, which are inputs of IR and Dyn, respectively, are information that can be commonly found in an issue-driven development project. We excluded the technique of MSR because we wanted to confirm that our technique can function properly without prior knowledge or experience of the system, which is a requirement of MSR-based impact analyses. Regarding RQ2, we included the technique of MSR because we wanted to compare the results of different impact analyses. Therefore, using a greater number of techniques is preferred for investigation.

4.1.3 | Data analysis

To evaluate the results, we used nDCG,[34,35] which is the normalization of the discounted cumulative gain (DCG), as a criterion. Actually, DCG is a popular measure for evaluating the quality of ranking documents with the assumption that the relevant documents appearing in a higher position of the list are more useful than those appearing in a lower position of the list. Furthermore, each relevant item can be graded by assigning a number according to the degree of relevance. The DCG is calculable using the following formula:

$$DCG_p = rel_1 + \sum_{i=1}^{p} \frac{rel_i}{\log_2 i}.$$

Therein, rel_i is the graded relevance of the result at rank i; p is the length of the given ranking. Then, nDCG can be computed by normalizing DCG as

$$nDCG_p = \frac{DCG_p}{IDCG_p},$$

where *IDCG* is the ideal DCG, ie, the maximum DCG value that is obtainable from the ranking result. However, because we defined the oracle as a set of code smells that might be relevant to the context of developers, the ideal DCG, in this case, represents an approximation that might or might not be the *ideal* ranking.

The relevant documents in this study are the code smells that match the items in the oracle. The retrieved documents are the code smells in the result from the code smell detector. We defined rel_i as the number of issues in the dataset that are related to a code smell. Our assumption is that solving a code smell related to multiple issues is more useful than solving one that is unrelated or related to only 1 issue. Therefore, a code smell that is related to many issues is expected to have a higher degree of relevance when calculating nDCG. Because our technique involves rearranging the result from a code smell detector and assigning a higher rank to relevant code smells, the nDCG of the result from our technique is expected to be higher than the baseline of our evaluation, ie, the original result from the code smell detector.

We calculated the nDCG of the baseline and calculated the result from our tools ordered by the CRI of each smell for all subjects.

4.2 | Investigating factors affecting the performance of our approach

4.2.1 | RQ1: Which code smell granularity is more appropriate for our technique: coarse-grained or fine-grained?

Study design. To answer this question, we conducted 2 independent experiments. We applied our technique for fine-grained (method-level) code smells in the first experiment and coarse-grained (class-level) code smells in the second one. We used method-level impact analysis to prioritize method-level code smells and class-level impact analysis for prioritizing class-level code smells.

Results and discussion. Figure 5 presents the results of our experiments. In the fine-grained case, our technique can provide ranking qualities with a minimum nDCG of 0.41 and maximum nDCG of 0.65. However, in the coarse-grained case, our technique can provide ranking qualities with a minimum nDCG of 0.68 and maximum nDCG of 0.97. As a comparison of these 2 cases shows, the coarse-grained granularity code smell can yield a better ranking.

When comparing the ranking quality between the baseline and the results obtained using our technique, in the case of the class-level code smell, our approach provides better ranking quality in every case. However, in the method-level code smell case, our approach generates better ranking quality than the baseline for the ArgoUML and jEdit projects but fails to do so for the JabRef and muCommander projects. We investigated the

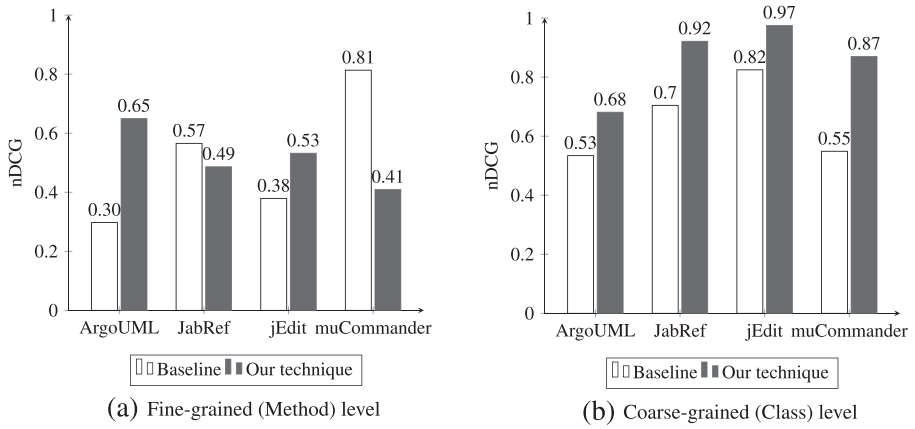

FIGURE 5 Baseline normalized discounted cumulative gain (nDCG) values and those obtained using our approach

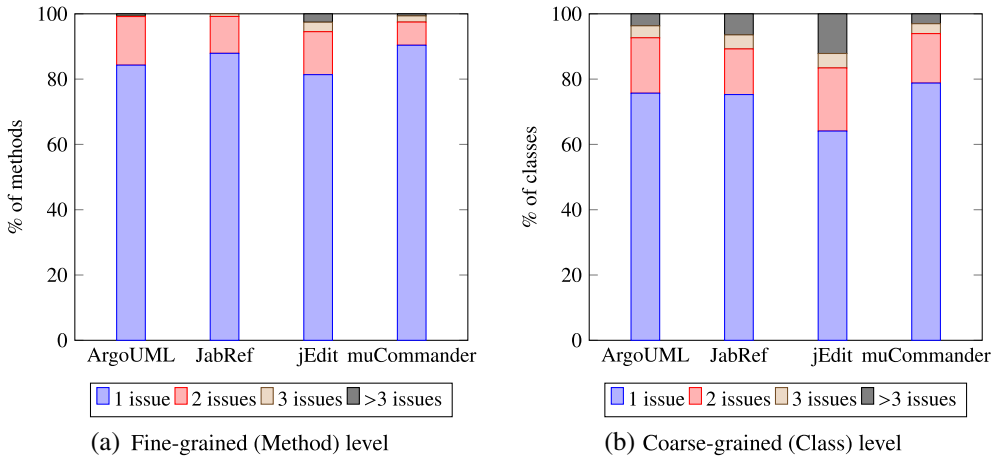

FIGURE 6 Commonality between method and class levels

results, which revealed that many smells have zero CRI, but they are related to the items in the oracle. Therefore, our approach predicted that these smells are unrelated to the developers' context, although they actually are related. One reason for that phenomenon might be the accuracy of impact analysis. Our technique relies solely on the impact analysis result. Therefore, the accuracy of impact analysis can also affect the accuracy of our technique. The impact analysis might have failed to locate the correct module that is the target of change information. Consequently, our method incorrectly predicted that this smell is unrelated to the developer context and thereby assigned it to the lower rank of the list. The impact of the accuracy of impact analysis to our approach is discussed in the next RQ. Such is not the case for class-level smells because the module *m* from impact analysis results must be equal to or belong to the *entity* of each smell *s* when we calculate the CRI value of each smell. Therefore, coarse-grained level code smells, such as class-level code smells, tend to satisfy the criterion more than the fine-grained level code smells, such as method-level code smells. This fact reflects that our technique is more appropriate for use with coarse-grained level code smells.

We conducted additional investigations to ascertain the reasons underlying this phenomenon. First, we analyzed the commonality of issues in the issue tracking system. As Figure 6 shows, most of the method was modified by only 1 issue. More precisely, the average numbers of issues per method are 1.17, 1.13, 1.28, and 1.13, respectively, for ArgoUML, JabRef, jEdit, and muCommander. However, many classes were modified by more than 1 issue. More precisely, the average numbers of issues per class were 1.37, 1.46, 2.01, and 1.36, respectively, for ArgoUML, JabRef, jEdit, and muCommander. In other words, method-level smells are too fine grained regarding the commonality of issues. It is difficult to specify relevant smells for the project's context. Preferably, coarse-grained ones can be related to multiple issues. Consequently, solving code smells at the class level can be expected to contribute more to issue implementation than solving the code smell at the method level.

Then we compared the accuracy between method-level impact analysis and class-level impact analysis. As Figure 7 shows, the F-1 scores of method-level impact analyses are 115.79%, 102.46%, 18.85%, and 84.12% lower than the class-level scores, respectively, for the ArgoUML, JabRef, jEdit, and muCommander projects. The accuracy of method-level impact analysis is significantly lower than the class-level accuracy, perhaps because of the nature of the impact analysis technique adopted in this research. It is easier to predict a relevant coarse-grained module (class) than a relevant

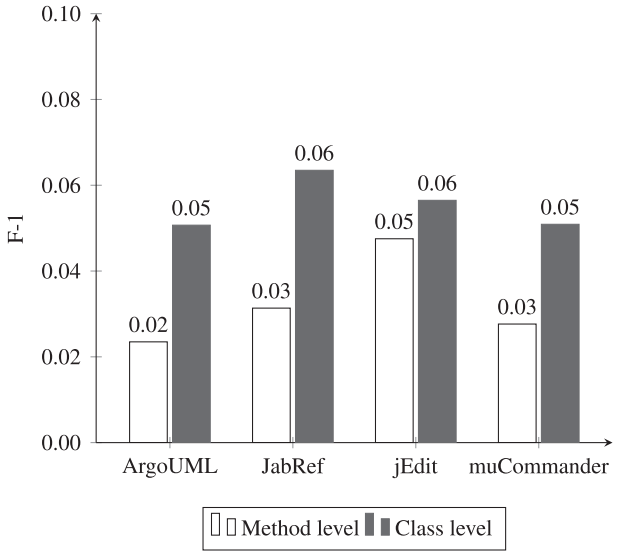

FIGURE 7 Accuracies of method-level impact analysis and class-level impact analysis

fine-grained module (method). Perhaps because of this reason, our technique can provide better ranking in every case of class-level code smells, but it failed to do so in the case of method-level code smells.

In conclusion, our technique is more appropriate with coarse-grained (class) level code smell.

4.2.2 | RQ2: Does impact analysis accuracy affect the ranking quality?

Study design. As explained earlier, a work by Gethers et al[21] shows that different impact analyses provide different accuracy. Therefore, to understand the impact of the accuracy of impact analysis on our ranking results, we used similar experimental settings to those of their work to observe differences related to accuracy among them. The techniques we used for this study include IR, IR + Dyn, IR + MSR, and IR + Dyn + MSR. Our assumption is that different techniques can be expected to influence the results of our approach.

They also concluded that different cut points, ie, the range of rankings that are going to be considered in the same technique, also contribute to the different accuracy of impact analysis. Moreover, the combination of different impact analyses might decrease the accuracy of impact analysis at certain cut points. Therefore, we also include different cut points of impact analysis (5, 10, 20, 30, and 40) into our setting.

In this study, we also divided impact analyses into 2 groups, which are impact analyses that do and do not require prior knowledge of the system. We assume that developers who have system knowledge know at least 1 module that is related to the issue while the developers who have no system knowledge do not possess such information. We want to ensure that our technique can function properly in both cases. Therefore, we conducted independent experiments between MSR-related impact analyses and other impact analyses. To simulate the activity that developers select the starting point for association mining, we randomly selected the module from the gold set provided in the dataset. Definitely, the accuracy of MSR-based impact analysis depends on the randomly selected starting point. For validity, we repeated each process 100 times and used the file that is closest to the average value for calculating nDCG and accuracy.

Results and discussion. We investigated the relation between nDCG and the accuracy of impact analysis using Spearman correlation coefficient to evaluate the association between 2 variables.

Figure 8 shows the relation between nDCG and the accuracy of each technique. The black points represent the data obtained using the IR and IR + Dyn techniques, whereas the red points represent data obtained using the IR + MSR and IR + Dyn + MSR techniques.

First, we consider all data points together: both black and red. By consideration of Figure 8c, the relation between nDCG and the F-1 measure of each technique is apparent. The value of Spearman correlation is approximately 0.29, indicating a weak but positive relation between the nDCG and F-1 measure. The reason underlying this weak relation might be the low values of F-1 measures. Figure 8a presents similar results to those of the F-1 measure. The Spearman correlation value is approximately 0.16, indicating a very weak but positive relation between nDCG and the precision value. We also suspect another reason underlying this relation: the low precision value. However, when considering Figure 8b, which represents the results of nDCG values at different recall values, the relation between the 2 variables can be understood. The calculated Spearman correlation value is approximately 0.48, indicating a stronger positive correlation than F-1 and precision. A tendency exists for high recall values to go with high nDCG values (and vice versa).

Then, if we consider only the MSR-related techniques (considering only the red points), it is apparent that they tend to yield higher nDCG values. We suspect that this is attributable to the way we conducted the MSR approach as explained earlier. The starting points of the MSR approach for association mining are randomly selected from the gold sets from each project; ie, they are the modules that are actually going to be

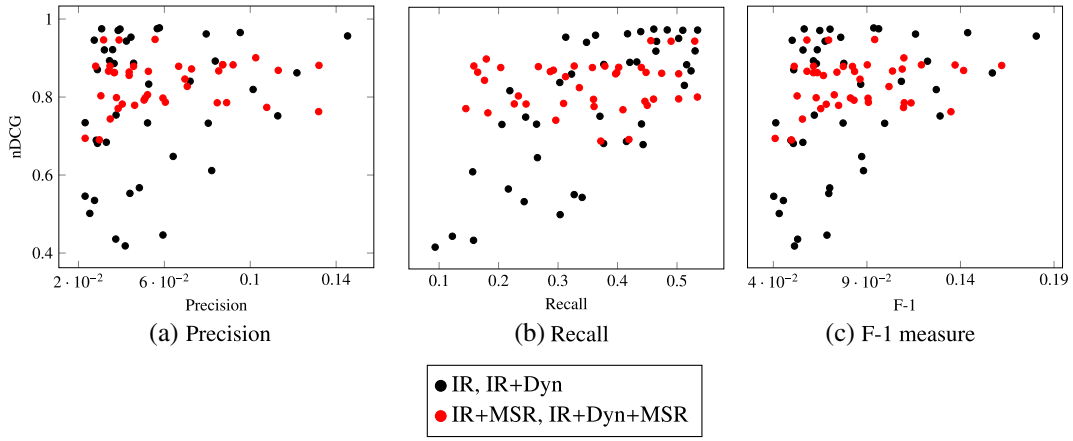

FIGURE 8 Relation between the impact analysis accuracy and the ranking quality of our technique. Dyn, dynamic analysis; IR, information retrieval; MSR, mining software repository

modified. As a result, the MSR-related techniques have a higher chance of predicting the correct modules. Consequently, they can provide higher nDCG values.

Because the MSR technique requires system knowledge of the developers and manual work to specify relevant modules, we also consider the correlation of only IR and IR + Dyn techniques by excluding MSR-related techniques (considering only black points). The values of Spearman correlation are approximately 0.38, 0.19, and 0.69, respectively, for F-1, precision, and recall. It is apparent that we also obtain positive correlation between nDCG and accuracy in this case.

To sum up, when one considers all techniques together, a positive correlation exists between nDCG and accuracy. However, if the MSR-related techniques are adopted, then the nDCG values are likely to be higher with the assumption that developers have system knowledge to specify the correct starting point modules. Nevertheless, if one considers only IR and IR + Dyn, which are techniques that can be fully automated using our approach, then we can also obtain positive correlations.

Therefore, we conclude that the accuracy of impact analysis tends to affect the quality of the ranking suggested by our technique. The higher the accuracy of impact analysis becomes, the better the quality of the ranking our technique can provide. Moreover, because the correlation coefficient between nDCG and recall is higher than those between nDCG and precision and between nDCG and recall, we can infer that recall probably affects our technique the most. Therefore, our technique is more suitable for high-recall impact analysis.

In summary, the accuracy of impact analysis tends to have a positive correlation with the ranking quality.

4.3 | Studying the meaningfulness and usefulness of the recommended prioritization

4.3.1 | RQ3: Does context-based smell prioritization provide more relevant results than severity-based smell prioritization?

Study design. We applied our technique to the dataset at the class level because it is suited to our approach, as discussed for RQ1. We used the combination of IR and Dyn with a cut point of 40 items because, as discussed earlier, the inputs of IR and Dyn techniques can be commonly found during software maintenance. We exclude the MSR technique because we want to confirm that our technique is useful properly without system knowledge or developer experience.

Results and discussion. As Figure 5b shows, the nDCG value of the results from our technique is higher than the nDCG of the baseline. Therefore, after the list of smells is prioritized using our technique, smells that are related to the developers' context were on the higher rank of the list. Consequently, developers can specifically examine the top-rank smells directly without specifying which smell is or is not related to their context.

Table 4 presents a comparison of the top 10 rankings with the baseline and our approach. Each row displays the rank of the smell, type of smell, the name of the module having the smell, and the number of issues actually modified in the module. We highlighted the code smell that is actually *relevant* to the developers' context, ie, smells in our oracle. The result of the baseline tends to be mixed with relevant and irrelevant smells, whereas our approach tends to put relevant smells at the top of the list. Moreover, the number of related issues of each smell in the baseline tends to be scattered, ie, sometimes with a high value at the top of the list and sometimes with a low value at the top of the list. In contrast, our approach tends to put code smells with the large number of related issues at the top of the list because our assumption is that solving code smells with the large number of related issues would be more beneficial to developers, ie, improving understandability and extensibility of the source code, than solving the code smell with a few related issues. The results can also be confirmed by calculating $nDCG_{10}$. For the baseline, we can obtain 0.22, 0.60, 0.66, and 0.35, respectively, for ArgoUML, JabRef, jEdit, and muCommander, whereas, for our approach, we can obtain 0.53, 0.85, 0.94, and 0.79 for the respective projects. The $nDCG_{10}$ obtained from the results from our approach is clearly higher than that of the baseline.

TABLE 4 Ranking items for the baseline and our approach

(a) ArgoUML: Baseline.

Rank	Class Name	Ranking	Severity	#Issues
1	Blob Class	GeneratorCSharp	8	
2	Blob Class	GeneratorJava	8	
3	God Class	FigAssociation	8	5
4	Blob Class	ParserDisplay	8	1
5	Blob Class	GeneratorPHP4	7	
6	Refused Parent Bequest	FigClassifierRole	7	3
7	Blob Class	Modeller	7	1
8	Schizophrenic Class	Import	6	
9	God Class	CoreFactoryMDRImpl	5	1
10	Refused Parent Bequest	StylePanelFigText	5	

(b) ArgoUML: Our approach.

Rank	Smell Type	Class Name	CRI	#Issues
1	God Class	Project	4.39	2
2	Schizophrenic Class	StylePanel	3.98	
3	God Class	ProjectBrowser	3.44	7
4	Blob Class	ProjectBrowser	3.44	7
5	God Class	ExtensionMechanisms...	2.19	1
6	God Class	FigEdgeModelElement	2.05	3
7	God Class	FigNodeModelElement	1.68	4
8	Schizophrenic Class	WizStep	1.60	
9	God Class	UMLMutableGraph...	1.58	
10	God Class	UmlFactoryMDRImpl	1.58	1

(c) JabRef: Baseline.

Rank	Class Name	Ranking	Severity	#Issues
1	God Class	BasePanel	10	4
2	God Class	JabRef	10	
3	God Class	EntryEditor	10	4
4	God Class	JabRefFrame	10	5
5	Data Class	GUIGlobals	10	
6	God Class	Util	5	7
7	God Class	EntryTableModel	5	
8	God Class	GroupsTree	5	
9	Schizophrenic Class	Option	4	
10	God Class	JabRefPreferences	4	3

(d) JabRef: Our approach.

Rank	Smell Type	Class Name	CRI	#Issues
1	God Class	JabRefFrame	2.94	5
2	God Class	Util	2.77	7
3	God Class	BasePanel	2.57	4
4	God Class	ImportFormatReader	1.93	
5	God Class	JabRef	1.91	
6	Blob Class	JabRef	1.91	
7	Schizophrenic Class	MainTableSelection...	1.89	2
8	God Class	EntryEditor	1.86	4
9	Blob Class	EntryEditor	1.86	4
10	God Class	BibTexDatabase	1.75	

(e) jEdit: Baseline.

Rank	Class Name	Ranking	Severity	#Issues
1	God Class	jEdit	10	9
2	God Class	View	10	12
3	Data Class	Debug	10	
4	God Class	Buffer	10	12
5	God Class	TokenMarker	9	4
6	God Class	Gutter	9	4
7	Data Class	DirectoryEntry	8	
8	Blob Class	JEditTextArea	8	5
9	God Class	ClassGeneratorUtil	7	
10	God Class	TarEntry	7	

(f) jEdit: Our approach.

Rank	Smell Type	Class Name	CRI	#Issues
1	God Class	Buffer	14.50	12
2	Blob Class	Buffer	14.50	12
3	God Class	jEdit	13.38	9
4	God Class	View	11.29	12
5	Blob Class	SearchAndReplace	9.35	7
6	God Class	Gutter	9.04	4
7	Schizophrenic Class	Gutter	9.04	4
8	God Class	StatusBar	8.47	3
9	God Class	TextAreaPainter	8.18	2
10	God Class	GUIUtilities	8.12	7

(g) muCommander: Baseline.

Rank	Class Name	Ranking	Severity	#Issues
1	God Class	JnlpTask	10	
2	God Class	StatusBar	7	
3	God Class	WindowManager	6	2
4	God Class	FileTable	6	10
5	Schizophrenic Class	CommandManager	5	1
6	God Class	FileFactory	5	3
7	God Class	HTTPFile	4	2
8	Data Class	isoPvd	4	
9	Data Class	FileCollisionChecker	4	
10	Schizophrenic Class	ActionKeymap	4	2

(h) muCommander: Our approach.

Rank	Smell Type	Class Name	CRI	#Issues
1	God Class	FolderPanel	7.21	6
2	Schizophrenic Class	LocalFile	6.34	6
3	God Class	MainFrame	4.94	3
4	God Class	WindowManager	4.28	2
5	God Class	FileTable	4.20	10
6	Schizophrenic Class	CommandManager	3.93	1
7	God Class	CommandManager	3.93	1
8	God Class	StatusBar	3.76	
9	Schizophrenic Class	StatusBar	3.76	
10	God Class	AbstractFile	3.74	2

We further assessed the result by considering the 2nd rank of jEdit project in the result obtained using our technique. That is the Blob Class smell of class `Buffer`. This smell was ranked 11th in the baseline. However, by application of our technique, this smell became the 2nd rank of the list with the highest *CRI* because this smell is related to many issues that must be resolved by the developers.

We confirmed that fact by investigating the actual changes made during release 4.2–4.3. Results showed that, out of 150 issues, 12 issues were implemented in this class, which includes this God Class smell. Therefore, if the developers realized the importance of this smell and fixed it, then it might facilitate their implementation by improving the understandability of source code for 12 issues.

In contrast, when considering the 1st rank of muCommander project in the baseline which is the God Class smell of the `JnlpTask` class, it is apparent that this smell was ranked 26th in the result from our technique. Our technique assigned this smell to the lower position of the list, specifically with

zero *CRI*, because it predicted that this smell is not related in any way with any issue in the issue tracking system. We also investigated the actual change and found no issues implemented in this class. Consequently, if the developers picked the 1st smell in the original result from the code smell detector and fixed it, then it might not support their implementation for any issue at all.

This evidence indicates that a list of smells ordered by the relevance to the developers' context can support developers' implementation more than the original order such as severity.

To summarize, context-based smell prioritization provides more relevant results than severity-based prioritization.

4.3.2 | RQ4: What is the effect of the combination of severity-based and context-based prioritization?

Study design. In this study, our purpose is to study the effects of combining both severity-based and context-based approaches. Although there are many ways to achieve such combinations, eg, using machine learning techniques, some approaches are not practical in this study because they require training data. Therefore, as a first step, we applied a simple linear combination in this study, which is expected to enable us to answer this RQ. We first define the *Ranking* as a key for prioritizing code smells. The *Ranking* is the linear combination of CRI and severity. The combination is calculable as

$$Ranking = \alpha \cdot nCRI + (1 - \alpha) nSeverity,$$

where $0 \leqslant \alpha \leqslant 1$. Setting $\alpha = 0$ means that we ignore CRI and use only severity for prioritization (pure severity-based prioritization). In contrast, setting $\alpha = 1$ means that we ignore severity and use only CRI for prioritization (pure context-based prioritization). *nCRI* is the linear normalization to range 0–1 of CRI of each smell in the list. Similarly, *nSeverity* is the linear normalization to range 0–1 of severity of each smell in the list. We let α stand for a parameter for adjusting the weight of *nCRI* and severity, for example, if α is equal to 0.5, then the weight of *nCRI* and severity will be equally important.

We conducted experiments to calculate nDCG for each project at different α values of 0.00–1.00 with the step of 0.01.

Results and discussion. Figure 9 represents the nDCG value of the results obtained using different α values (black lines). It is apparent that they are the points at which we obtain the lowest nDCG values, as expected, because these are where we use only severity if we consider the points at which α is equal to 0. However, if one considers the points where α are equal to 1, then it is apparent that these points tend to be the points at which we can obtain high nDCG values (0.68, 0.92, 0.97, and 0.87 for ArgoUML, JabRef, jEdit, and muCommander projects). Then, if one considers the points at which α are equal to 0.5, particularly when we regard CRI and severity as equally important, we can obtain the nDCG values of 0.70, 0.81,

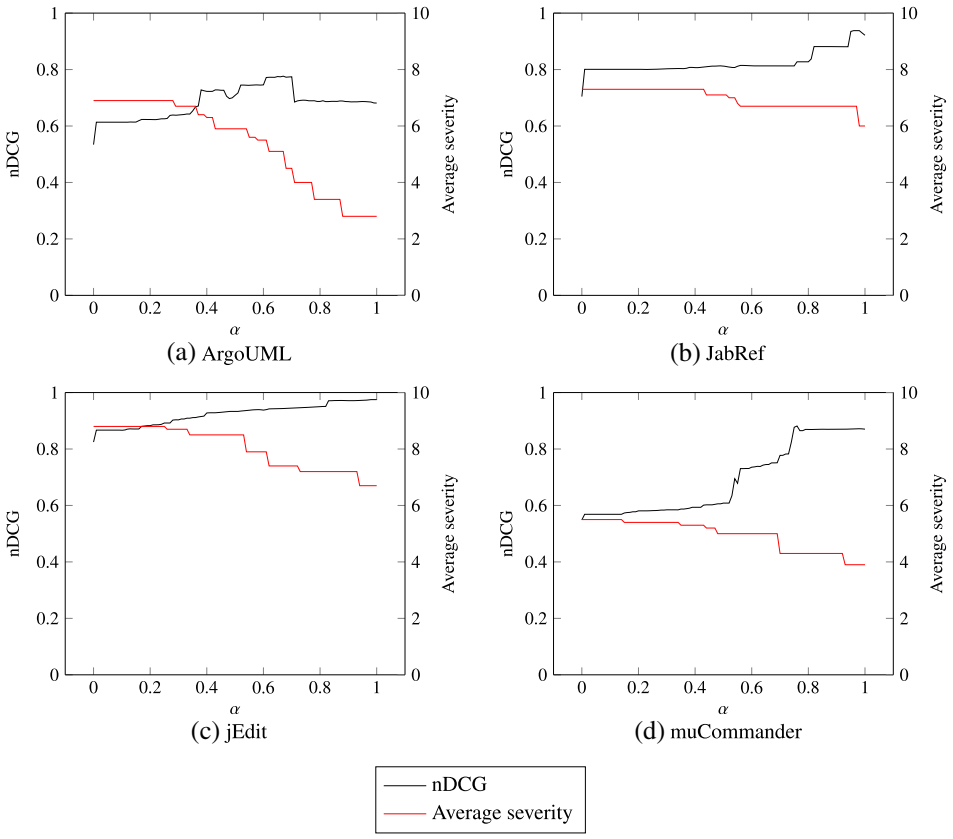

FIGURE 9 nDCG values and average severity of top 10 smells in the list of results obtained using different α values

TABLE 5 Ranking Items of ArgoUML between $\alpha = 1$ and $\alpha = 0.5$

(a) $\alpha = 1$.

Rank	Class Name	Ranking	#Issues	Severity
1	Project	1	2	2
2	StylePanel	0.91		3
3	ProjectBrowser	0.78	7	5
4	ProjectBrowser	0.78	7	3
5	ExtensionMechanisms...	0.50	1	2
6	FigEdgeModelElement	0.47	3	3
7	FigNodeModelElement	0.38	4	3
8	WizStep	0.37		1
9	UMLMutableGraph...	0.36		2
10	UmlFactoryMDRImpl	0.36	1	4

(b) $\alpha = 0.5$.

Rank	Class Name	Ranking	#Issues	Severity
1	ProjectBrowser	0.68	7	5
2	StylePanel	0.60		3
3	Project	0.57	2	2
4	GeneratorCSharp	0.55		8
5	GeneratorPHP4	0.55		7
6	FigAssociation	0.54	5	8
7	GeneratorJava	0.54		8
8	ProjectBrowser	0.54	7	3
9	ParserDisplay	0.50	1	8
10	FigClassifierRole	0.43	3	7

0.93, and 0.61 respectively for ArgoUML, JabRef, jEdit and muCommander projects. It is noteworthy that the value of nDCG is slightly lower than where α is equal to 1. However, we regard such decrements as rather small in light of the fact that we can use the severity value for prioritization.

Table 5 presents a comparison of the top 10 results of ArgoUML obtained when we apply our technique with $\alpha = 1$ (Context-based) and $\alpha = 0.5$ (Combination of context-based and severity-based). As discussed in earlier paragraphs, combining severity with CRI is expected to decrease the ranking quality slightly. It is apparent that the left table ($\alpha = 1$) can provide more relevant results than the right table ($\alpha = 0.5$). However, the right table tends to assign high severity code smells to the top of the list more than the left table. Specifically, the average severity of top 10 results of the left table is 2.8, whereas the right table shows average severity of 5.9.

Moreover, Figure 9 also represents the average severity of top 10 smells in the list of the results obtained using different α values (red lines). If one considers ArgoUML project, then it is apparent that the nDCG value is approximately the same (≈ 0.69) at the point where $\alpha = 0.71$ and $\alpha = 1$. However, the average severity value of top 10 smells in the list is 4 where $\alpha = 0.71$, and 2.8 at the point where $\alpha = 1$. Therefore, in this situation, reducing the α value not only yields a higher severity value, on average; the reduction also does not affect the nDCG value. In other words, reducing α value can put more severe code smells to the top position while maintaining the context relevance of the list. The same tendency is applicable with other projects as well.

Overall, it might be said that each combination tends to have specific characteristics based on the α value, as we expected. Low α values tend to generate results that put first code smells with high severity. This combination is useful for root canal refactoring,[36] when developers aim to improve overall source code quality. In root canal refactoring, developers might not have any specific context. For that reason, particularly addressing code smells that contribute most to the decay of the system might be preferable. On the other hand, high α values are likely to produce results that put first code smells that related to the context of developers. The situation that most fits this combination is when developers perform floss refactoring[36] which is when refactoring is performed together with other changes such as bug fix or feature implementation as in prefactoring phase. In this case, developers have a specific context on the task they have to work on. For that reason, putting high priority on smells that are likely to facilitate their task is expected to yield greater benefits. We find this can be an important parameter for developers to tune the most appropriate value depending on their circumstances.

To sum up, each combination of severity-based and context-based approach tends to generate results with different characteristics. However, using α value at a certain point might be able to produce results that put first code smells with high severity while maintaining the relevance to the context.

4.3.3 | RQ5: Can our approach predict the smells to be refactored?

Study design. Bavota et al[15] investigated the relation between the quality of a software product and related refactoring activities. They mined the history of 3 Java projects to verify whether developers apply refactoring operations to modules that are affected by code smells. They also provided a dataset of class level code smells that were *refactored* by developers. Therefore, we can conduct a similar study to that in Section 4, ie, calculating the accuracy of ranking using nDCG. Instead of using the oracle in Section 4 which is a set of code smells that occur in the modules that were *modified* by developers during 2 releases, we used a set of code smells that occur in the modules that were *refactored* by developers during 2 releases. We used the number of refactorings applied to smells extracted from the dataset as rel_i when calculating nDCG because we want to see whether our technique can predict modules that are going to be refactored by developers. Modules that will be refactored many times are expected to have higher nDCG than those with less application of refactoring operations. Moreover, because the only project that our study and their dataset has in common is ArgoUML, we conducted the study of only ArgoUML ver. 0.22–0.24.

Results and discussion. Figure 10 represents the obtained nDCG value of each impact analysis technique. We obtained the nDCG values of 0.32, 0.39, 0.22, 0.26, and 0.23, respectively, for the baseline (severity-based technique), IR, IR+Dyn, IR+MSR, and IR+MSR+Dyn (context-based techniques). In this case, we consider that all techniques produced rankings of poor quality. Therefore, we can simply conclude that our technique is unsuitable for predicting code smells that are going to be refactored by developers. The reason underlying this phenomenon is that developers in

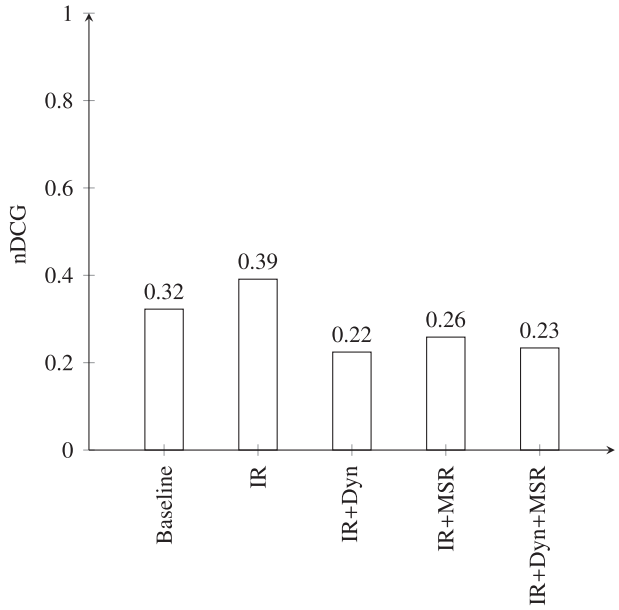

FIGURE 10 Accuracy of the ranking of ArgoUML using the number of refactorings applied to smells as the oracle

the current development style do not solve smells frequently.[15] Bavota et al concluded that, according to their dataset, only 42% of refactoring operations are applied to modules that are affected by code smells. However, only 7% of the operations are actually code smells removed from the system. Therefore, most recommended smells on the rankings were not actually refactored, even if they *should have been* refactored. Consequently, the results of this study indicate that code smells are not the only factor that causes developers to refactor the source code. However, finding the related factors is beyond the scope of this paper because our intention is not to predict the smells that current software developers are solving but to recommend the smells that are expected to have sufficient impact on their development.

To conclude, our technique is inappropriate for use with predicting smells to be refactored.

4.4 | Threats to validity

Internal validity is related to factors that might affect the outcome. In this case, the code smell detection tool that we are using, namely inFusion, detects code smells based only on object-oriented software metrics. Additionally, it might have produced some false negatives that might have an impact on our experiments. For example, if we consider modules that have metric values that are slightly below the thresholds defined using a code smell detector, then these modules are not classified as having code smells. However, if these modules are related to the context of the developers, for instance, if they are the location that developers are going to make some changes, then our technique will fail to output the results to developers even though they have poor quality because they were not in the original list. That is, they are not *code smells* according to the detection tools.

In addition, in some cases, the detector might produce some false positives. For example, a parser class having God Class code smell is not considered problematic because its scope is generally large and because refactoring it might even reduce the comprehensibility of the class.[37,38] To mitigate this threat, one of the authors has manually verified the top 10 results of both severity-based and context-based approaches following a false positive catalog of Fontana et al.[37,38] Although we did not notice any false positive in the top 10 results obtained using our approach, one false positive was found in the severity-based result of ArgoUML and JabRef project. We excluded the false positive smells and recalculated the $nDCG_{10}$. In RQ3, the $nDCG_{10}$ values are slightly changed from 0.22 and 0.60 to 0.29 and 0.62. In RQ5, the value changed from 0.22 to 0.24. We conclude that false positives have little effect on the results of this study. However, because the process of identifying false positives was performed by one of the authors who is not a main developer of the projects used for this study, we cannot ensure the completeness of the identified result. Conducting experiments with different code smell detection techniques or applying false positive detection to the process is expected to be worthwhile. Furthermore, We rely on the dataset of Dit et al[18] in this study. Although the dataset can be expected to have high quality because of change-based extraction, the possibility exists that errors in the dataset can affect our study results.

Construct validity deals with the relation between theories and observations. The largest threat to construct validity is the oracle dataset that we used. The oracle was extracted based on the changes in version control repositories. Therefore, it might lack some important modules that were not modified by solving the given issues but which should be refactored. An example is modules that must be understood to make changes. Refactoring them contributes to the improvement of understandability. In addition, one can improve the extensibility of a module without refactoring it directly, but by refactoring another related module instead. However, it is not easy to extract such modules in an empirical manner. Version control and issue tracking repositories do not include such information in a formal way. Although the use of fine-grained interaction history of developers such as Mylyn logs[39] might be useful to confirm the context of developers more explicitly, collecting the histories of such types is another challenge.[40]

Another construct validity is that the *CRI* used for prioritizing code smells does not express the relevance to the *context* of the developers completely. We calculate *CRI* by relying solely on impact analysis, which takes only the change information and the source codes as inputs. Such techniques guarantee neither accuracy nor completeness. Therefore, the *CRI* that we used in this research must be improved to express the relevance to the context of the developers better.

External validity is related to the generalization of the results. For this study, we conducted experiments on 4 Java open-source projects available in the dataset. However, we were unable to confirm that the results will remain the same when applying our technique to different projects that might have different size and language. Conducting experiments with more types of projects remains as a subject for our future work.

5 | CONTROLLED EXPERIMENT

As described herein, the main goal of our approach is to prioritize code smells that are related to the developers' context. We designed the approach under the assumption that code smells related to the developers' context should be solved first in the prefactoring phase because it might support the related implementation. We conducted this controlled experiment to verify whether the assumption holds, or not, and to confirm whether our approach can put first code smells that professional developers think should be solved in the prefactoring phase. Given the prefactoring phase situation, professional developers were asked to select code smells that they think should be solved. We then performed evaluations of our technique based on the list of code smells selected by developers. The next subsection explains details of our experiment.

5.1 | Experimental setup

5.1.1 | Case and subject selection

In the previous study, we used datasets of 4 open-source projects. Each project includes numerous issues and code smells because many developers were working on those issues. In this study, because we wanted to collect data from individual developers, using the same datasets would be impractical. Therefore, we design this study with the aim of a simulation case study based on a real project that can minimize subjects' task while generating sufficient data for us to analyze.

We chose JabRef ver. 2.0 as our case study because its size is the smallest among 4 projects that we used in our previous study. We selected 5 issues from the dataset because we think that it is an appropriate number for a developer to analyze in a short amount of time. Our choice of issues was not random but based on several factors. First, we filtered out the issues for which the solution set of classes, ie, those modified to solve the issue, contains no code smell because we wanted to examine modules having code smells. Then, we selected issues that modified fewer than 4 classes to lessen the developers' investigation time. Lastly, because we wanted the final set of issues to reflect our assumption that code smells related to many issues are more important than code smells related to few issues, we selected the final set of issues from the remaining 13 issues based on the commonality of each issue's solution set. Finally, we obtained the set of 5 issues shown in Table 6, which includes various commonality, ie, there are 2 classes modified by 1 issue, 1 class modified by 2 issues, and 1 class modified by 3 issues.

For code smells, we filtered out code smells in the package that are irrelevant to any issue to reduce the number of code smells. We obtained total 22 smells for which the severity of each smell is 1–10. The smell type consists of Blob Class, Data Class, God Class, and Schizophrenic Class.

As for subjects, we hired 10 professional developers having experience with or currently working on Java projects. Their experience was 2–13 years. Positions of the subjects were from entry level (eg, software engineer), senior level (eg, senior software engineer), to management level (eg, development group leader).

5.1.2 | Data collection

We first started our study by explaining an overview of JabRef application. Then, we explained the code smell concept and the definition of 4 code smells used for this study. The definition of prefactoring that is going to used for this study was explained afterward. The subjects were also asked to fill out a short survey about their current position and years of experience.

After completing the preparation, we explained the situation that they are assumed to be in a prefactoring phase of a particular release where they must solve a list of issues. We then showed the subjects a list of 5 issues that must be solved. Each issue includes a summary and description. Because the subjects were not the actual developers of the JabRef project, asking them to find the solution of each issue independently had the

TABLE 6 Issues used for this study

Issue number	Summary
1436014	No comma added to separate keywords
1535044	Month sorting
1601651	PDF subdirectory - missing first character
1738920	Windows Position in Multi-Monitor environment
1785416	inking with exact BibKey fails

potential to be time-consuming. In addition, if developers work on their projects, they are likely to be able to ascertain the solution, or the location for the solution, to the issue. Therefore, we also presented details of the solution, ie, the *diff* of the conducted change, of each issue to fill this gap. Subjects were then asked to read and understand all 5 issues. Subsequently, we showed them a list of 22 code smells. Each smell includes an entity name, package name, smell type, and detailed description as to why it is regarded as a code smell, eg, the list of methods that are the sources of the smell, provided by inFusion. Source code of the project, including the modules having code smells, were also presented to the subjects for analysis. The subjects were then asked to select code smells that should be refactored. We neither gave them any criteria nor the number of code smells they should select.

5.1.3 | Data analysis

Because our main goal is to evaluate the performance of our technique on prioritizing code smells that align with developers' selection, we use average precision (AP), which is a popular metric for evaluating ranking documents as a criterion. The reason that we do not use nDCG as in Section 4 is that we regard each code smell selected by developers as equally important. Because the purpose of nDCG is to evaluate ranking documents where each document is not equally important, we found that using AP is more appropriate and straightforward for this study. AP is calculable using the following formula:

$$AP = \frac{\sum_r P@r}{R}.$$

In that expression, r stands for the rank of each relevant document, R signifies the total number of relevant documents, and $P@r$ denotes the precision of the top-r retrieved documents. In this context, the relevant document is the code smell selected by developers. The retrieved document is the code smell in the result from the code smell detector. Therefore, the AP value of the result from our technique should be high if it can put code smells that developers selected for the top of the list. Additionally, we use the mean average precision (MAP) to summarize a set of AP values.

5.2 | RQ6: How does our technique prioritize code smells selected by professional developers in the prefactoring phase comparing to the severity-based approach?

5.2.1 | Study design

In this study, we want to evaluate our approach by comparison to the severity-based whether code smells that are put to the top of the list and by evaluating their agreement with professional developers' perspective. Therefore, we used code smells that were selected by developers as an oracle because they are what developers think should be solved during prefactoring phase. Then, we evaluated both techniques by calculating AP of each technique for each developer's answer. We made a box plot based on the obtained AP.

5.2.2 | Results and discussion

Figure 11 shows a box plot of the AP between severity-based and context-based approaches. The severity-based approach gives us a median AP of 0.36 whereas context-based approach yields 0.77. As might be readily apparent, the context-based approach outperformed the severity-based approach. We confirmed the results by application of the Wilcoxon rank sum test, which revealed that the results were statistically significant (p-value = 0.005). Therefore, our approach can generate a better ranking based on code smells that professional developers selected.

Table 7 presents the top 10 code smells from severity-based and context-based approaches. Each row displays the information of each smell and the number of developers that selected each smell. We highlighted the code smell that at least 1 developer selected during the experiment.

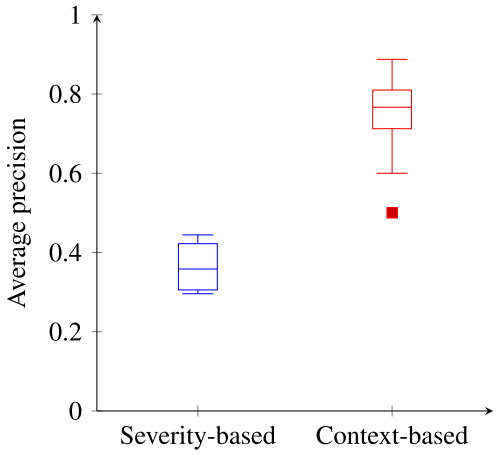

FIGURE 11 Average precision between severity-based and context-based approaches

TABLE 7 Ranking items for the severity-based and context-based approaches

(a) Severity-based

Rank	Smell Type	Class Name	Severity	#Devs
1	God Class	BasePanel	10	
2	God Class	JabRef	10	
3	God Class	EntryEditor	10	6
4	God Class	JabRefFrame	10	9
5	Data Class	GUIGlobals	10	
6	God Class	Util	5	9
7	God Class	EntryTableModel	5	
8	God Class	GroupsTree	5	
9	God Class	JabRefPreferences	4	9
10	God Class	EntryTable	4	

(b) Context-based

Rank	Smell Type	Class Name	CRI	#Devs
1	God Class	Util	0.46	9
2	God Class	JabRefFrame	0.28	9
3	God Class	BasePanel	0.22	
4	God Class	EntryEditor	0.21	6
5	Blob Class	EntryEditor	0.21	5
6	God Class	BibtexDatabase	0.18	
7	Data Class	GUIGlobals	0.18	
8	God Class	JabRef	0.15	
9	Blob Class	JabRef	0.15	
10	God Class	JabRefPreferences	0.13	9

It is noteworthy that the severity-based approach yields only 4 selected smells whereas the context-based yields 5 selected smells. In addition, the severity-based approach puts selected smells in rank as 3rd, 4th, 6th, and 9th whereas the context-based approach puts selected smells in rank as 1st, 2nd, 4th, 5th, and 10th. The difference is clearer if we consider the top 5 items because the context-based approach includes 4 selected smells, whereas the severity-based approach includes only 2 smells. This result suggests that the context-based approach is likely to be more practical for use in the prefactoring phase.

In addition, if we calculate AP using the oracle that we used in the previous study, ie, a set of code smells occurring in the modules that were modified by developers, we can obtain a value of 0.42 from severity-based and 0.81 from the context-based approach. By comparing these values to the MAP obtained from the oracle used for this study, which are 0.36 and 0.74 for severity-based and context-based respectively, we can infer that the results in this study support the results from our empirical studies for which we use a set of code smells occurring in the modified modules as an oracle.

In conclusion, results show that our context-based approach can prioritize code smells selected by professional developers better than the severity-based approach can.

5.3 | RQ7: What is the most appropriate proportion of linear combination of severity-based and context-based prioritization for the prefactoring phase?

5.3.1 | Study design

In RQ4, we studied the effect of the combination of the severity-based and context-based approaches. We concluded that high α value is appropriate for the situation in which developers who have a specific context perform refactoring to support the implementation such as prefactoring phase. Consequently, regarding this RQ, we conducted experiments using different α values. Then, we calculate MAP, which is the mean of AP of each developers' answer at different α values. This is expected to yield α that can generate the best MAP which is the most appropriate α value for prefactoring phase.

5.3.2 | Results and discussion

Figure 12 represents the MAPs of respective α values. In this case, it is apparent that the value of MAP tends to increase where α value increases, perhaps because of the fact that developers selected code smells to be refactored by relevance to the context, not by severity. Therefore, in this study, we conclude that using α value close to 1 is likely to yield best ranking results for prefactoring. The result here agrees with those of our previous study described in Section 4 showing that the prefactoring phase is more appropriate with high α value.

To sump up, α value close to 1 is most appropriate for the prefactoring phase.

5.4 | Threats to validity

Internal validity A salient concern might be the fact that the subjects are not main developers of JabRef project. Consequently, they might not have a deep understanding of the system. However, because we selected only issues that require simple modification (fewer than 4 locations) and because we provide them the solution of each issue as a guideline for understanding changes, we believe that we could mitigate this threat. Nevertheless, providing the subjects with the diffs of the solutions might have an impact on the results because the subjects might select code smells by biasedly map with the class name in each solution. Conducting an experiment with the core developers of the targeted project without providing the solution of the issues might be able to mitigate this threat.

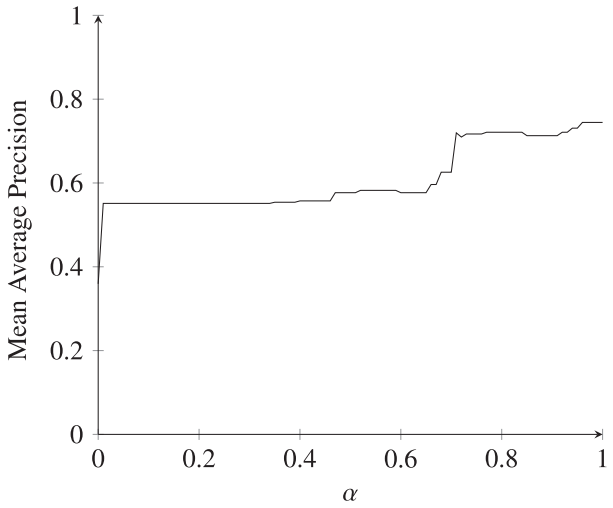

FIGURE 12 Mean Average Precision of results obtained using different α values

Another threat to internal validity might be the final set of issues to be used for this study. When selecting them, aside from filtering out by other factors, we strove to include various commonality of the issues, ie, some issues modified the same classes and some issues modified different classes. The reason is, as discussed, to reflect our assumption that smells related to many issues are more important than smells related to few issues. Therefore, selecting only those issues that modify the same class would not be able to verify the assumption and would be less beneficial.

External validity For this study, we conducted experiments on a reduced size of issues and code smells which might be a threat. However, asking developers to perform tasks on the whole set of data maybe impractical because it requires a considerable amount of time and effort. In addition, although the results of this study are similar to the results of the previous study for which we used the full set of data, we cannot ensure if the results are going to be applicable to other sets of issues. Such investigation is beyond the scope of this study and remains as a subject of our future work. The number of developers in this study is 10, which might not be representative. However, the subjects in our study are from a different background within a range of 2 to 13 years of experience. The positions of the subjects are also varied from junior, senior, and management level. Nonetheless, conducting the study of a larger scale might be beneficial.

6 | RELATED WORK

Since the concept of code smell has been introduced, many techniques have been proposed to facilitate the process of code smell detection. Nonetheless, because code smell detectors tend to generate huge numbers of smells, many techniques have also been proposed to reduce the number of code smell detection results, eg, prioritization and filtration. This section summarizes work related to code smell detection, prioritization, and filtration. Moreover, we conclude with presentation of some work in the literature that considers the developer context for supporting developers.

6.1 | Code smell detection

Research related to code smell detection has devoted the most attention to using metric values of source code for detection of code smells such as detection strategies reported by Marinescu,[41] DECOR by Moha et al,[42] jDeodorant by Tsantalis et al,[43] an approach by Munro et al,[44] and an approach by Lanza and Marinescu.[3] In addition, many attempts have been made at applying other techniques to enhance metric-based analysis. For example, Khomh et al[45] proposed a Bayesian approach that can work with incomplete data and which allows analysts to adjust the results with their knowledge. Oliveto et al[46] proposed a technique based on numerical analysis to overcome the code smell detector limitations that either have too strict rules or which require expensive tuning by an expert. Sahin et al[47] proposed code smell detection as a bilevel problem that allows the prediction of new code smells that differ from those of an example. Moreover, recent research such as that by Fontana et al[48,49] has underscored the use of machine learning for detecting code smells.

Although most works specifically examine metric-based detection, some attempts have been conducted to use information of different types to detect code smells. Palomba et al[50] proposed TACO as a technique that uses textual analysis to detect code smells. Their technique applies IR to textual elements such as source code identifiers and comments and calculates the probability that a code component is affected by a code smell. Additionally, they proposed HIST, which uses change history information mined from version control systems to detect code smells.[51] Specifically, they exploit association mining to analyze co-changes and to generate modules that are likely to be affected by code smells.

As we discussed earlier, because we designed our approach as a framework, it is useful with most code smell detectors. This benefit is expected to be more useful than strict reliance on a detection strategy.

6.2 | Code smell prioritization

Arcoverde et al[52] proposed a technique to prioritize code anomalies, or code smells, based on their potential to the software architecture degradation. They used 4 heuristics: change density, error density, anomaly density, and architecture role. They concluded that their technique is mostly useful in scenarios of several kinds: architectural problems in a system involving classes that changed together, architectural problems related to communicating across different classes, changes that are not predominantly perfective, some architecture roles that are affected by many code smells, and the architecture roles of the system are well defined and have distinct architectural relevance. In contrast, our technique emphasizes support of developers during the prefactoring phase. We devote attention only to smelly modules within the focus of developers because solving smells on the modules that developers are not going to touch would not support their implementation. Moreover, we limit our input to the issue tracking system to reflect real-world situations in which the availability of information is limited.

Fontana et al[11] introduced the Code Smell Intensity index as a criterion to prioritize code smells. The Code Smell Intensity index is computed by the distribution of software metrics and is intended to quantify the importance of each code smell instance. They presented the index as a numeric value of 1–10 and defined 5 intensity levels: Very Low, Low, Mean, High, and Very High. The approach devotes attention specifically to the most severe smell instances. Moreover, it is limited to code smells of 6 types: God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling, and Message Chain. In contrast, our approach specifically examines more context-related smell instances. Furthermore, ours is not limited to specific smells. The novel point of our approach compared to those others is that our approach is applicable to smells of many kinds. We prioritize every smell in the detection result based on the relevance to the developers' context.

Vidal et al[53] proposed a technique to prioritize code smells based on 3 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. The technique prioritizes code smells based on the context of developers using modifiability scenarios. However, such information requires experience and knowledge of the system, as well as manual work specifying scenarios and related source code components. In contrast, our technique can process this step automatically. Their technique also requires some information related to the change history of the system. Such information would mostly be available in the late stages of software development, where our technique is useful at any stage of software development. Consideration of other factors such as code smell type preferences of developers remains a subject of our future work.

Codabux and Williams[54] presented a framework for prioritizing technical debt using predictive analysis. They first consider classes that are defect-prone and change-prone as a technical debt item. They then calculate the *technical debt proneness* of each class using a prediction model that was constructed using a Bayesian Approach. The items are categorized into low, medium, and high groups using the prediction model. Finally, with the decision from project managers or developers, their technique generates technical debt items having the probability of being the most critical. As described earlier, their technique considers classes that are defect-prone and change-prone as a technical debt item. However, our technique specifically examines every code smell generated from detection tools. Moreover, their work requires a manual decision of the developers to generate output, whereas our technique emphasizes context-related results and process this step manually.

6.3 | Code smell filtration

Fontana et al[12] proposed a technique to reduce the number of code smell detection results by the application of strong and weak filters, which is calculated by consideration of the application domain of the system. A strong filter is intended to remove false positives from code smells detection results such as code smells in a test class. A weak filter identifies code smells that might not be a shortcoming of the system or which do not affect software quality, such as the Shotgun Surgery smell in getter and setter methods so that developers can solve them when they have time. However, the method limits the technique to code smells of 5 types: God Class, Data Class, Shotgun Surgery, Dispersed Coupling, and Message Chains, whereas our technique has no such limitation. Moreover, the aim of their technique is to improve the accuracy of code smell detection, whereas our technique intends to provide developers with relevant code smells.

Ratiu et al[55] used historical information to increase the accuracy of code smell detection. They measured the stability, ie, how often the class changed (at least 1 method was added or removed) and persistence, ie, how long the class has been affected by code smells, of classes. Then they used the results to filter out the entities that might not adversely affect the original results detected using a single-version strategy. They also identified most dangerous smells using additional analyzed historical information. However, their technique is limited to God Class and Data Class code smells. Similarly to the works described earlier, the approach specifically examines improvement of the accuracy of smell detection by filtering out smells that are unlikely to cause problems with the system, whereas our technique emphasizes the context of developers.

6.4 | Context-based approach

Hayashi et al[56] proposed a technique to suggest refactoring operations using a sequence of program modification. Their work uses the editing operation by which a developer performs during the program modification activities, eg, copy paste, to estimate the developer's context. The technique specifically examines suggesting refactoring operations only on the modules that the developer is currently working on without devoting attention to other modules, even though they are affected by code smells.

Liu et al[57] presented a monitor-based instant refactoring framework to suggest refactoring opportunities to a developer. The technique will run in the background during the code changing process of the developer. The technique will then invoke the code smell detector and warn developers to solve the code smells if such changes are likely to introduce code smells. In other words, this technique is designed to detect code smells only on the modules on which developers are currently working.

Morales et al[58] proposed ReCon, an approach that uses a developer's context for automatic refactoring of code smells. They defined the refactoring strategies for code smells of 4 types: Lazy Class, Long Parameter List, Spaghetti Code, and Speculative Generality. Then they used the *task context* to limit the process scope. The task context is the source code entities that developers used when working on maintenance task. Such information can be extracted from monitoring tools log such as Mylyn. They used metaheuristics techniques, which are Simulated Annealing, Genetic Algorithm, and Variable Neighborhood Search, to generate the sequence of refactoring operations that can remove code smells.

Even though the resources for context estimation differ, the techniques described earlier, and our technique specifically examines the *current context* of the developer. Nevertheless, although our work specifically examines supporting the *prefactoring* phase, when developers refactor source code to prepare for implementation, their work can be regarded as supporting the *postfactoring* phase because the technique works during the source code editing process used by developers.

7 | CONCLUSION

As described herein, we proposed a technique for prioritizing code smell detection results by consideration of the developers' current context. We estimated the context of the developers by application of impact analysis to the list of issues in issue tracking system and used the results to calculate the CRI of each code smell. The results obtained using our technique are a list of smells, prioritized based on their relevance to the developer context. Smells that are more relevant to the developer context are ranked higher on the list. Therefore, our approach can assist developers in prioritizing code smells for the prefactoring phase. Our technique is useful for planning how to prefactor the source code before implementing sets of issues in an issue-tracking system.

We conducted empirical studies of 4 open-source projects related to the use of developers' context and smell prioritization to study the characteristics of our technique and the factors that may affect the result's quality. First, results show that coarse-grained code smells can provide a better ranking than fine-grained code smells for several reasons. Second, the accuracy of impact analysis tends to affect the ranking quality. Third, results show that our context-based smell prioritization technique can provide more relevant results than the severity-based smell prioritization. Fourth, each combination of severity-based and context-based approach is appropriate for different situations. Finally, results show that, because of the current of software development style, our approach is inappropriate for predicting smells that are going to be refactored.

Additionally, we conducted a controlled experiment and showed that our technique can put first code smells that are agreed with developers' choices. We also concluded that the combination approach with high α value is more practical in prefactoring phase.

Our future work includes the examination of case studies to confirm that relevant code smells, as defined in this context, are useful to developers. Furthermore, other factors that might affect developers' decisions related to fixing smells must be considered, eg, the effort needed to fix the smells and the importance of the issues. In addition, more projects must be undertaken to evaluate our technique.

ACKNOWLEDGEMENTS

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

ORCID

Natthawute Sae-Lim http://orcid.org/0000-0001-7122-8350

REFERENCES

1. Fowler M. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley; 1999.
2. Wake WC. *Refactoring Workbook*. Boston, MA, USA: Addison-Wesley; 2003.
3. Lanza M, Marinescu R. *Object-Oriented Metrics in Practice*. Berlin, Germany: Springer; 2006.
4. Yamashita A, Moonen L. Do code smells reflect important maintainability aspects? In: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12), Riva del Garda, Trento, Italy; 2012:306-315.
5. Hermans F, Aivaloglou E. 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), Austin, TX, USA; 2016:1-10.
6. Tufano M, Palomba F, Bavota G, Oliveto R. When and why your code starts to smell bad. In: Proceedings of the 37th International Conference on Software Engineering (ICSE'15), Florence, Italy; 2015:404-414.
7. Vale T, Souza IS. Influencing factors on code smells and software maintainability: A cross-case study. In: Proceedings of the Second Workshop on Software Visualization, Evolution and Maintenance (VEM'14), Maceio, AL, Brazil; 2014:86-93.
8. Rajlich V. *Software Engineering: The Current Practice*. Boca Raton, FL, USA: Chapman and Hall – CRC; 2011.

9. Kersten M, Murphy GC. Using task context to improve programmer productivity. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06), Portland, OR, USA; 2006:1-11.
10. Johnson B, Song Y, Murphy-Hill ER, Bowdidge RW. Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 35th International Conference on Software Engineering (ICSE'13), San Francisco, CA, USA; 2013:672-681.
11. Fontana FA, Ferme V, Zanoni M, Roveda R. Towards a prioritization of code debt : A code smell intensity index. In: Proceedings of the IEEE Seventh International Workshop on Managing Technical Debt (MTD'15), Bremen, Germany; 2015:16-24.
12. Fontana FA, Ferme V, Zanoni M. Poster: Filtering code smells detection results. In: Proceedings of the 37th International Conference on Software Engineering (ICSE'15), Florence, Italy; 2015:803-804.
13. Marinescu R. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*. 2012;56(5):9:1-9:13.
14. Yamashita A, Moonen L. Do developers care about code smells? An exploratory survey. In: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13), Koblenz, Germany; 2013:242-251.
15. Bavota G, De L A, Di P M, Oliveto R, Palomba F. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*. 2015;107:1-14.
16. Meng N, Hua L, Kim M, McKinley KS. Does automated refactoring obviate systematic editing? In: Proceedings of the 37th International Conference on Software Engineering (ICSE'15), Florence, Italy; 2015:393-402.
17. Silva D, Tsantalis N, Valente MT. Why we refactor? Confessions of GitHub contributors. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16), Seattle, WA, USA; 2016:858-870.
18. Dit B, Revelle M, Gethers M, Poshyvanyk D. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process*. 2013;25(1):53-95.
19. Sae-Lim N, Hayashi S, Saeki M. Context-based code smells prioritization for prefactoring. In: Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16), Austin, TX, USA; 2016:1-10.
20. Bohner SA, Arnold RS. *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press; 1996.
21. Gethers M, Dit B, Kagdi H, Poshyvanyk D. Integrated impact analysis for managing software changes. In: Proceedings of the 34th International Conference on Software Engineering (ICSE'12), Zurich, Switzerland; 2012:430-440.
22. Kohan A, Yamamoto M, Artho C. Automated dataset construction from web resources with tool Kayur. In: Proceedings of the Fourth International Symposium on Computing and Networking (CANDAR'06), Hiroshima, Japan; 2016:98-104.
23. Salton G, McGill MJ. *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, Inc.; 1983.
24. Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*. 1990;41(6):391-407.
25. Dit B, Wagner M, Wen S, Wang W, Linares-Vásquez M, Poshyvanyk D, Kagdi H. ImpactMiner: A tool for change impact analysis. In: Companion proceedings of the 36th International Conference on Software Engineering (ICSE'14), Hyderabad, India; 2014:540-543.
26. Dit B, Holtzhauer A, Poshyvanyk D, Kagdi H. A dataset from change history to support evaluation of software maintenance tasks. In: Proceedings of the Tenth Working Conference on Mining Software Repositories (MSR'13), San Francisco, CA, USA; 2013:131-134.
27. Keenan E, Czauderna A, Leach G, Cleland-Huang J, Shin Y, Moritz E, Gethers M, Poshyvanyk D, Maletic J, Huffman Hayes J, Dekhtyar A, Manukian D, Hossein S, Hearn D. TraceLab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In: Proceedings of the 34th International Conference on Software Engineering (ICSE'12), Zurich, Switzerland; 2012:1375-1378.
28. Dit B, Moritz E, Poshyvanyk D. A TraceLab-based solution for creating, conducting, and sharing feature location experiments. In: Proceedings of the IEEE 20th International Conference on Program Comprehension (ICPC'12), Passau, Bavaria, Germany; 2012:203-208.
29. Intooitus. inFusion. http://www.intooitus.com/products/infusion. (visited on 04/14/2015).
30. Brown WH, Malveau RC, McCormick HW, Mowbray TJ. *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York, NY, USA: John Wiley & Sons, Inc.; 1998.
31. Riel AJ. *Object-Oriented Design Heuristics*. Boston, MA, USA: Addison-Wesley; 1996.
32. Martin RC. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall; 2007.
33. Hunt A, Thomas D. *The Pragmatic Programmer: From Journeyman to Master*. Boston, MA, USA: Addison-Wesley; 2000.
34. Järvelin K, Kekäläinen J. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems*. 2002;20(4):422-446.
35. Croft B, Metzler D, Strohman T. *Search Engines: Information Retrieval in Practice*. 1st ed. Boston, MA, USA: Addison-Wesley Publishing Company; 2009.
36. Murphy-Hill E, Black AP. Refactoring tools: Fitness for purpose. *IEEE software*. 2008;25(5):38-44.
37. Fontana FA, Dietrich J, Walter B, Yamashita A, Zanoni M. Preliminary catalogue of anti-pattern and code smell false positives. Tech. Rep. RA-5/15, Poznan University of Technology; 2015. Available: http://www2.cs.put.poznan.pl/wp-content/uploads/2015/11/RA-5-2015.pdf
38. Fontana FA, Dietrich J, Walter B, Yamashita A, Zanoni M. Antipattern and code smell false positives: Preliminary conceptualization and classification. In: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16), Osaka, Japan; 2016:609-613.
39. Kersten M, Murphy GC. Mylar: A degree-of-interest model for IDEs. In: Proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD'05), Chicago, IL, USA; 2005:159-168.
40. Negara S, Vakilian M, Chen N, Johnson RE, Dig D. Is it dangerous to use version control histories to study source code evolution?. In: Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12), Beijing, China; 2012:79-103.
41. Marinescu R. Detection strategies: Metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, IL, USA; 2004:350-359.
42. Moha N, Guéhéneuc YG, Duchien L, Meur AFL. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*. 2010;36(1):20-36.

43. Tsantalis N, Chaikalis T, Chatzigeorgiou A. Jdeodorant: Identification and removal of type-checking bad smells. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08), Athens, Greece; 2008:329-331.
44. Munro MJ. Product metrics for automatic identification of "Bad Smell" design problems in java source-code. In: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05), Como, Italy; 2005:15-15.
45. Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H. A bayesian approach for the detection of code and design smells. In: Proceedings of the Ninth International Conference on Quality Software (QSIC'09), Jeju, South Korea; 2009:305-314.
46. Oliveto R, Khomh F, Antoniol G, Guéhéneuc YG. Numerical signatures of antipatterns: An approach based on b-splines. In: Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR'10), Madrid, Spain; 2010:248-251.
47. Sahin D, Kessentini M, Bechikh S, Deb K. Code-smell detection as a bilevel problem. *ACM Transactions on Software Engineering and Methodology*. 2014;24(1):6:1-6:44.
48. Fontana FA, Zanoni M, Marino A, Mantyla MV. Code smell detection: Towards a machine learning-based approach. In: Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM'03), Amsterdam, The Netherlands; 2013:396-399.
49. Fontana FA, Mäntylä MV, Zanoni M, Marino A. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*. 2016;21(3):1143-1191.
50. Palomba F, Panichella A, De L A, Oliveto R, Zaidman A. A textual-based technique for smell detection. In: Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16), Austin, TX, USA; 2016:1-10.
51. Palomba F, Bavota G, Di P M, Oliveto R, De L A, Poshyvanyk D. Detecting bad smells in source code using change history information. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13), Palo Alto, CA, USA; 2013:268-278.
52. Arcoverde R, Guimaraes E, Macia I, Garcia A, Cai Y. Prioritization of code anomalies based on architecture sensitiveness. In: Proceedings of the 27th Brazilian Symposium on Software Engineering (SBES'13), Brasilia, DF, Brazil; 2013:69-78.
53. Vidal SA, Marcos C, Díaz-Pace JA. An approach to prioritize code smells for refactoring. *Automated Software Engineering*. 2016;23(3):501-532.
54. Codabux Z, Williams BJ. Technical debt prioritization using predictive analytics. In: Proceedings of the 38th International Conference on Software Engineering Companion (ICSE'16), Austin, TX, USA; 2016:704-706.
55. Ratiu D, Ducasse S, Girba T, Marinescu R. Using history information to improve design flaws detection. In: Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR'04), Tampere, Finland; 2004:223-232.
56. Hayashi S, Saeki M, Kurihara M. Supporting refactoring activities using histories of program modification. *IEICE Transactions on Information and Systems*. 2006;E89-D(4):1403-1412.
57. Liu H, Guo X, Shao W. Monitor-based instant software refactoring. *IEEE Transactions on Software Engineering*. 2013;39(8):1112-1126.
58. Morales R, Soh Z, Khomh F, Antoniol G, Chicano F. On the use of developers' context for automatic refactoring of software anti-patterns. *Journal of Systems and Software*. 2017;128:236-251.

How to cite this article: Sae-Lim N, Hayashi S, Saeki M. Context-based approach to prioritize code smells for prefactoring. *J Softw Evol Proc*. 2018;30:e1886. https://doi.org/10.1002/smr.1886