Context-Based Code Smells Prioritization for Prefactoring

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

***Abstract*—To find opportunities for applying prefactoring, several techniques for detecting bad smells in source code have been proposed. Existing smell detectors are often unsuitable for developers who have a specific context because these detectors do not consider their current context and output the results that are mixed with both smells that are and are not related to such context. Consequently, the developers must spend a considerable amount of time identifying relevant smells. As described in this paper, we propose a technique to prioritize bad code smells using developers' context. The 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 which smells are associated with the context. Consequently, our approach can provide developers with a list of prioritized bad code smells related to their current context. Several evaluations using open source projects demonstrate the effectiveness of our technique.**

I. Introduction

Code smell is an indicator of a design flaw or problem in the source code which can affect important maintainability aspects [1]. If applied properly, it can be solved by refactoring, which is a technique for improving the software structure without changing its functionality [2]. Code smells of many types are summarized as a smell catalog with their names [2], [3], [4]. Code smells are often introduced when implementing new features and often by developers with high workloads [5]. Several factors, such as bad design decisions and low priority assigned to software quality, can influence code smells [6]. A code smell detection strategy using a logical condition of code metrics has been proposed [4]. Code smell detectors of various types have been proposed by detecting these code smells automatically [7].

In an issue-driven software development project, development teams tend to adopt an issue tracking system such as Jira or Bugzilla to manage their lists of issues. The issue can be anything related to software change requests, e.g., bug fixing or feature implementation. When developers apply refactoring techniques to source code before implementing changes according to issues, we call this stage *prefactoring* phase [8]. Many studies have highlighted the importance of this prefactoring phase. Meng et al. [9] found 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. [10] supported this statement by asserting the possibility that the only goal of refactoring, for developers, is to prepare source code for future changes. Given that situation, modules within the focus of the developers can be regarded as their context. Such modules are likely to have higher priority to fix code smell and improve code quality over others because fixing code smells in them might contribute to support of future implementation, i.e., improving the understandability and extendibility of the source code. However, 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 not related to the context of the developer, e.g., the severity of each smell. Therefore, the time-consuming process of identifying relevant outputs is left to developers, which is similar to the fact that static analysis tools are not used well by developers because of numerous detected warnings [11].

A clear need for a context-aware approach for code smell detection is apparent in recent research in this area. Most respondents to work done by Yamashita et al. [12] state that developers need code smell detectors that support context-sensitive or domain specific strategies. Work done by Bavota et al. [10] suggests that the developer's perspective should be considered when proposing refactoring recommendations.

As described in this paper, 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 extendibility and understandability by refactoring the source code before implementing a feature. Using the proposed technique, change descriptions in an issue tracking system that are going to be implemented using a particular milestone are regarded as the developer context. Such changes are going to be implemented by modifying or extending existing modules in the source code. These existing modules can be located using impact analysis [8], [13]. We regard such modules as relevant modules because they are likely to be the location for implementation of the new features in the change descriptions. 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

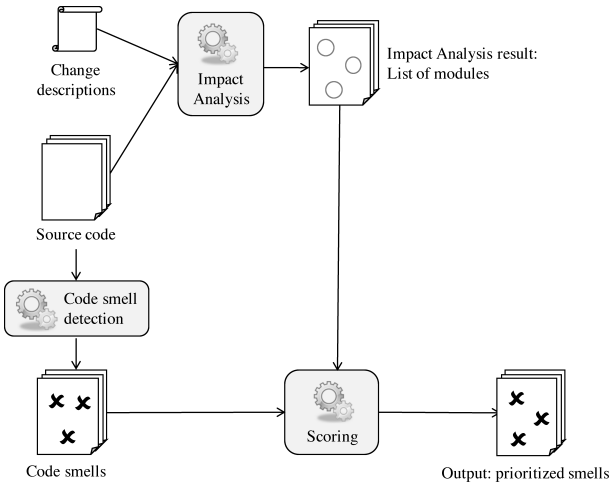


Fig. 1. Overview of the proposed technique.

to the context.

We have evaluated our technique using four open source projects. The results demonstrate the effectiveness of our technique.

The main contributions of this paper are the following:

1) showing the relevance between developers' context and code smells can be a useful criterion for prioritizing code smells for prefactoring,
2) presenting a technique to use an issue tracking system to estimate the context of developers using impact analysis, and
3) presenting an empirical study related to context-based code smell prioritization.

The remainder of this paper is organized as follows. The next section describes our approach and its automated toolchain. Section 3 presents our evaluation. Section 4 presents a description of related work. Then Section 5 concludes this paper.

II. Proposed Technique

As described in this paper, we specifically examine support of an issue-driven software development project adopting an issue tracking system to manage their lists of issues. Assuming that such a project has a list of issues that must be implemented before a particular milestone, i.e., releasing major or minor versions, then such a list of issues is useful to estimate the developers' context. We regard the modules that are likely to be modified as our estimation.

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 issue change descriptions obtained from the issue tracking system and the source code of the targeted project. The output is the prioritized list of code smells based on the relevance to the developer context. Our approach

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.

Fig. 2. Example portion of an issue in issue tracking system.

first uses impact analysis to obtain a list of modules that are likely to be the targeted modules of each change description. Next, we generate a list of code smells by application of an existing code smell detector with the source code of the focused project. Then, for each code smell in the list, we calculate the *Context Relevance Index* (*CRI*) based on the relevance of the prior result from impact analysis. Finally, we output the prioritized list of code smells ordered by the *CRI* value.

Our approach uses the existing impact analysis and code smell detection techniques. The following subsections present an explanation of these techniques and their application.

A. Using Impact Analysis

During software development process, developers can modify some modules to satisfy a change request, e.g., bug fixing or feature implementation. Developers might first use their experience, system knowledge, or technique such as feature location [13] to identify at least one 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], [14].

Impact analyses of various types take different inputs, e.g., natural language query, execution scenario, or source code artifact [13]. Gethers et al. [15] proposed a combination technique to perform impact analysis depending on the source of contextual information available to each software project, e.g., information retrieval, mining software repository, or dynamic analysis.

Figure 2 presents an example of an issue in issue tracking system of jEdit project. We consider the information in *Summary* together with *Description* field as a change description. In our technique, impact analyses that take a change description d and source code C as inputs and provide a set of modules $M = \{\ldots, m, \ldots\}$ with their probability as outputs were the types that we chose because we specifically examine support of issue-based software development projects in which developers tend to implement features or fix bugs by following the change descriptions in an issue tracking system. In this situation, we assume that such a change description describes the new behavior of an existing feature, i.e., fixing a bug or improving functionality. These existing features can be located using impact analysis. Therefore, the located modules of these features can be candidates to be modified to achieve the change. Consequently, applying the

TABLE I
EXAMPLE 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

prefactoring technique to these modules is likely to support the developers' implementation, i.e., improving understandability or extendibility of the source code. The impact analysis of this type suits our needs.

As described in this paper, we input a set of change descriptions $D = \{d_1, \ldots, d_n\}$ and source code C to the impact analysis and obtain a series of sets of modules $\{M_1, \ldots, M_n\}$. As described herein, modules can be either classes or methods.

B. Using Code Smell Detection Technique

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, e.g., lines of code (LOC). The input is the source code that we want to analyze. The output is a list of smells. Each smell consists of $\langle type, entity, granularity, severity \rangle$, 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 subsystem, class, and method, and *severity* is an integer value representing the strength of the smell, for example, Marinescu defined in [16] as: "Severities are computed by measuring how many times the value of a chosen metric exceeds a given threshold". Table I presents an example of code smell result from inFusion [17], a code smell detector that we used in this research. For example, the second row shows the Blob Class smell in `org.gjt.sp.jedit.Buffer` class with severity 7. We have omitted the package and class name of the method level smell.

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

C. Context Relevance Index

To assign the priority of each smell, we define the *Context Relevance Index* (*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 the predicate $match(m, entity)$ holds when module m equals to or belongs to *entity* of each smell, and where $w(m)$ stands for the probability of each module. If m is a method, and *entity* having 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 code smell is a class, $match(m, entity)$ holds when m is in *entity* class.

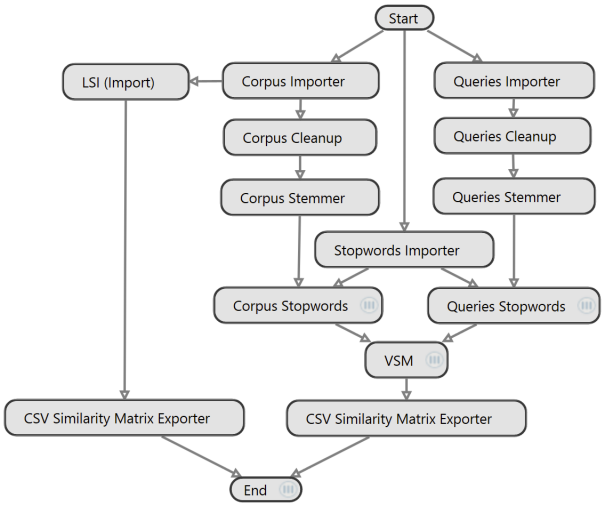


Fig. 3. TraceLab configuration.

III. EMPIRICAL STUDIES

A. Research Questions

We conducted an empirical study to validate our approach with the following research questions.

RQ 1: *Which code smell granularity provides a better ranking: Coarse-grained or fine-grained?*

RQ 2: *Does the accuracy of impact analysis affect quality of the ranking?*

RQ 3: *Does context-based smell prioritization provide more relevant results than the severity-based smell prioritization?*

The aim of these research questions is to validate the approach of using the context of developers to prioritize code smells. Details of the motivations of respective RQ are explained later.

B. Experimental Setup

1) Experimental Implementation: We have implemented an automated tool for use with the proposed technique. The chain is designed to connect with an existing impact analysis tool. When executed, our tool trigger a code smell detector to generate a list of smells and to calculate the *CRI* of each smell based on the relevance of the result from impact analysis.

For impact analysis, we use tools proposed by Dit et al. [18] together with a TraceLab-based solution [19], [20] to compute the vector space model and latent semantic indexing. Then, we filter out the unrelated modules using the execution trace available in the dataset. These techniques were chosen because they are the impact analysis that take a text document such as a change description as a query and find a relevant code based on statistical methods. Figure 3 portrays our configuration in TraceLab which is based on [20].

For smell detection, we use inFusion ver. 1.9.0 [17] because 1) it can detect code smells of 24 types such as Blob Class, Data Class, or Feature Envy, 2) all detected smells are associated with the severity score, and 3) its detection process can be automated. These characteristics suit our approach.

TABLE II
DATASET INFORMATION

Project	Version	Size (LOC)	# Issues	# Method level smells	# Class level smells
ArgoUML	0.22–0.24	309,468	91	412	61
JabRef	2.0–2.6	72,555	39	60	37
jEdit	4.2–4.3	140,592	150	184	51
muCommander	0.8.0–0.8.5	88,455	92	44	41

2) Data Collection: In this evaluation, four open source projects, ArgoUML[1], Jabref[2], jEdit[3], and ArgoUML[4], active open source projects, were our subjects because their data are available through the benchmark dataset [13] of Dit et al. The dataset includes the list of *Summary* and *Description* information, the list of executed methods, and the gold set methods of each issue during analyzed period. Table II presents information of our datasets including the size of the source code of the earlier version, the number of issues that we used in between two versions, and the number of smells detected by inFusion ver. 1.9.0 [17].

We first defined the oracle as a set of code smells that occur in the modules that were modified by developers during two releases according to the data in the benchmark dataset because these code smells are relevant to the developers' context as we discussed in the previous section. As for ArgoUML, we prepared the oracle by first applying the source code at version 0.22 to the code smell detector and obtained the result. Next, we used the gold set methods, the methods that were actually modified to solve extractable issues in jEdit's issue tracking system, from the benchmark dataset during version 0.22 and 0.24. Finally, we intersected these two sets to obtain a list of smells that are actually related to the developer context. We applied the same process to JabRef ver. 2.0–2.6, jEdit ver. 4.2–4.3 and muCommander ver. 0.8.0–0.8.5.

Regarding the baseline of our evaluation, we used the original result from inFusion sorted by the severity of respective smells by applying a stable sort to the original result.

3) Data Analysis: To evaluate the results of our approach, we used Normalized Discounted Cumulative Gain (nDCG) [21], [22], which is the normalization of Discounted Cumulative Gain (DCG), as a criterion. DCG is a popular measure for evaluating the quality of ranking documents with the assumption that the relevant documents appearing in the higher position of the list are more useful than those appearing in the lower position of the list. Furthermore, each relevant item can be graded according to the degree of relevance using numerical number. Actually, DCG is calculable using the following formula:

$$DCG_p = rel_i + \sum_{i=1}^{p} \frac{rel_i}{\log_2(i)}$$

[1]http://argouml.tigris.org/
[2]http://www.jabref.org/
[3]http://www.jedit.org/
[4]http://argouml.tigris.org/

where rel_i is the graded relevance of the result at rank i and 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, the maximum DCG value we can obtain from the ranking result.

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. Because our assumption is that solving a code smell related to multiple issues is more useful than solving one unrelated or related to only one issue, a code smell that relates to many issues is expected to have a higher degree of relevance when calculating nDCG.

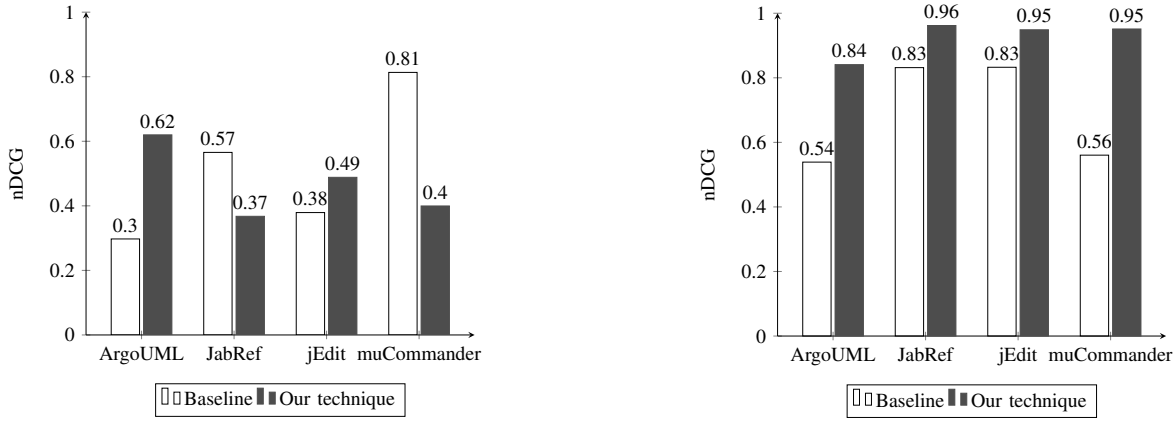
Therefore, because our technique involves rearranging the result from a code smell detector and assigning a higher rank to relevant code smells, the nDCG of the result from our technique is expected to be higher than the baseline of our evaluation, i.e., the original result from the code smell detector.

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

C. RQ 1: Which code smell granularity provides a better ranking: Coarse-grained or fine-grained?

1) Motivation: Code smell detection tools often classify smell granularity as coarse-grained or fine-grained, e.g., class level and method level. Solving code smells of both types is likely to improve the source code quality. If developers solve code smells in the module that they are going to modify, then it would support their 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 the understanding of how to solve it is easier. Therefore, we suspect that, in context-based prioritization, coarse-grained and fine-grained granularity code smells would yield different ranking quality.

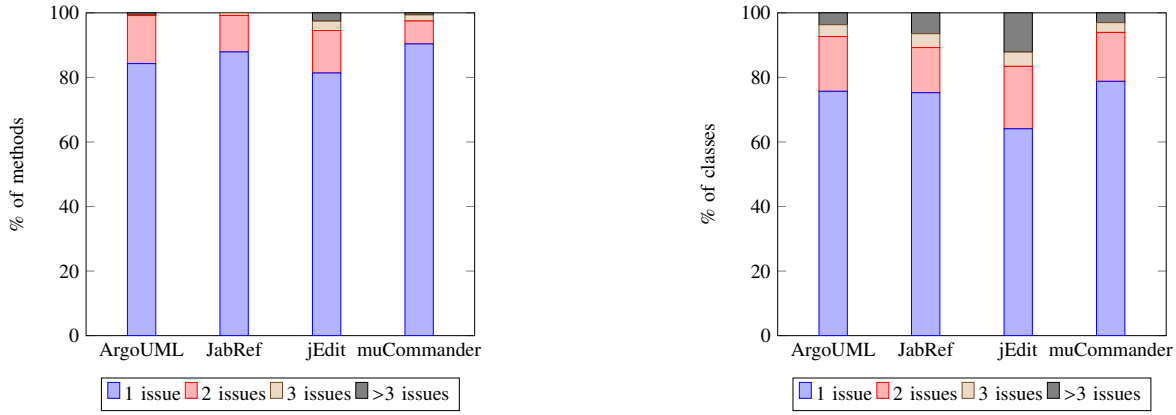
2) Study Design: To answer this question, we conducted two independent experiments. We applied our technique for fine-grained (method level) code smells in the first experiment and coarse-grained (class level) code smells in the second one. We used method level impact analysis to prioritize method



(a) Fine-grained (Method) level.

(b) Coarse-grained (Class) level

Fig. 4. Comparison of the nDCG value between results of baseline and our approach.



(a) Fine-grained (Method) level.

(b) Coarse-grained (Class) level.

Fig. 5. Comparison of the commonality between method and class level.

level code smells, whereas we used class level impact analysis for prioritizing class level code smells.

3) Results and Discussion: Figures 4a and 4b present results of our experiments. In the fine-grained case, our technique can provide ranking qualities with the minimum nDCG 0.37 and maximum nDCG 0.62. However, in the coarse-grained case, our technique can provide ranking qualities with the minimum nDCG 0.84 and maximum nDCG 0.96. As comparison of these two cases shows, the coarse-grained granularity code smell can yield a better ranking. Note that we used the vector space model with dynamic analysis (see detail in the next subsection) as an impact analysis because it produced the best result.

When comparing the ranking quality between baseline and the results obtained using our technique, in 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 ArgoUML and jEdit projects, but fails to do so for JabRef and muCommander projects. We investigated the results and found that many smells have zero *CRI*, but they are related with 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. Because our technique relies solely on the result of impact analysis, the accuracy of impact analysis can also affect the accuracy of our technique. That is to say, the impact analysis might have failed to locate the correct module that is the target of a change description. Consequently, our method incorrectly predicted that this smell is not related 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 sub section. The reason that this is not the case for class level smells is that module *m* from impact analysis result must equal to or belong to *entity* of each smell *s* when we calculate the *CRI* value of each smell. Therefore, the 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 indicates that our technique is more appropriate for use with coarse-grained level code smells.

We conducted further investigation into the reason for this phenomenon by analyzing the commonality of issues in the

issue tracking system. As Figs. 5a and 5b show, most of the method were modified by only one issue, with the average issues per method of 1.17, 1.13, 1.28, and 1.13, respectively, for ArgoUML, JabRef, jEdit, and muCommander. However, there are many classes that were modified by more than one issue with the average issues per class of 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 in terms of the commonality of issues, it is difficult to specify very 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 would contribute more to issue implementation than solving the code smell at the method level.

D. RQ 2: Does the accuracy of impact analysis affect quality of the ranking?

1) Motivation: In this study, we limit the input of the technique to only the source code and change descriptions to reflect real-world circumstances in which information availability is limited. Therefore, in our technique, the only criteria that would impact the ranking quality of our result are results from impact analysis. Investigating how different impact analyses affect the ranking quality can enable us to find the appropriate impact analysis for context-based code smell prioritization.

2) Study Design: To understand the impact of the accuracy of impact analysis, we selected some impact analyses from work by Gethers et al. [15] to observe differences related to accuracy among them. Their work's aim is to integrate different impact analysis to improve the overall accuracy of each independent technique, but also showed that different impact analysis approaches yield different accuracies of the result. They discussed differences among combinations of these three techniques: information retrieval (IR), dynamic analysis (Dyn), and mining software repository (MSR). Actually, IR and the combination of IR and Dyn were our candidates because they require only the change description and execution trace, which are common during the processes of iterative software development. We excluded the technique of MSR because we want to limit the input of the technique to the source code and change descriptions. Regarding IR techniques, we use the vector space model and latent semantic indexing because they are both vector-based techniques but are applied with different mechanisms. Our assumption is that different techniques can be expected to influence the result of our approach, even though they are IR-based techniques. The next paragraphs present overviews of the respective techniques we used for this study.

Vector Space Model (*VSM*) is a model used mostly for information retrieval 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. The results then are the relevance between source code and change requests.

Latent Semantic Indexing (*LSI*) [24] is based on VSM, with the intention to handle the situation of synonymy, a group of words that share similar meanings, and polysemy, words that have multiple meanings. Actually, LSI is used for assessment of the similarity between documents from the common words two documents contain rather than simple terms. The more common words they have, the more similar they are. Details of the techniques we used for this study can be found in an earlier report of the literature [15].

Dynamic Analysis (*Dyn*) uses the execution traces of the given program. 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 run-time information is useful to filter out the result from IR technique because the modules that are not in execution trace are unlikely to be affected by the change request.

According to work by Gethers et al. [15], different cut points in the same technique also contribute to different accuracy of impact analysis. As described in the paper, the combination of different impact analysis might decrease the accuracy of impact analysis at certain cut points. Therefore, we also include different cut points of impact analysis into our assumption.

3) Results and Discussion: Figure 6 presents a comparison of nDCG value of different impact analysis. As the graph shows, each impact analysis and each cut point provide different nDCG values. In most cases, VSM+Dyn yields better results than VSM; LSI+Dyn yields better results than LSI. Additionally, higher cut points tend to generate better results than lower cut points. It is noteworthy that our intent is not finding the technique that can provide best result, but analyzing the impact of the accuracy of each technique.

We conclude that, when using different impact analyses, either IR-based alone or the combination of IR-based and dynamic analysis, one can expect that different techniques in the same IR-based category (VSM or LSI) or even different cut points in the same technique contribute to the quality of the ranking from our technique. Because the main characteristic that would differ among respective techniques is the accuracy, e.g., precision, recall, or F-1 measure, it would be natural to consider the relation between the quality of our result and the accuracy of impact analysis.

We considered the relation between nDCG and the accuracy of impact analysis using Spearman's correlation coefficient to evaluate the association between two variables. By considering Fig. 7c, we can see the relation between nDCG and the F-1 measure of each technique. The value of Spearman's correlation is approximately 0.37, indicating a weak but positive relation between the nDCG and F-1 measure. The reason underlying this weak relation might be the low value of F-1 measures.

Figure 7a presents similar results to those of the F-1 measure. The Spearman's correlation value is approximately 0.24, also indicating a weak but positive relation between

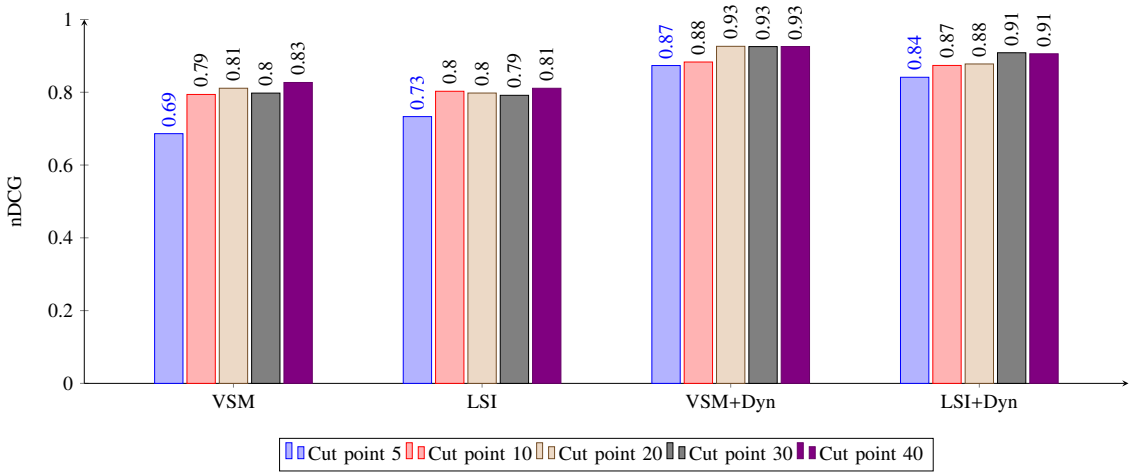
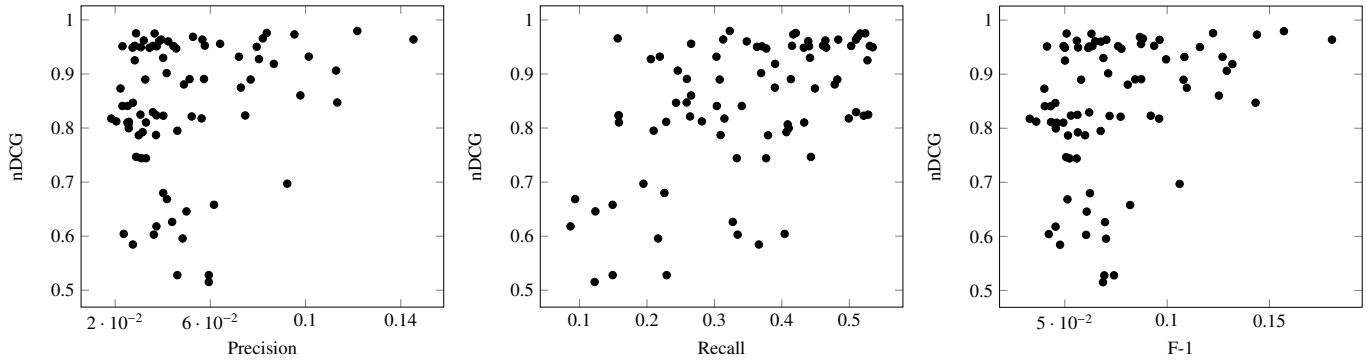


Fig. 6. Comparison of the nDCG value between results obtained using different impact analysis approaches.



(a) Precision (r = 0.24). (b) Recall (r = 0.48). (c) F-1 measure (r = 0.37).

Fig. 7. Relation between the accuracy of impact analysis and the ranking quality of our technique.

nDCG and the precision value. We also suspect that the reason behind this relation is the low precision value.

However, when considering Fig. 7b which represents the results of nDCG value at different recall values, the relation between two variables can be understood. The calculated Spearman's correlation value is approximately 0.48, indicating a moderate positive correlation. A tendency exists for high recall values to go with high nDCG values (and vice versa).

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 ranking our technique can provide. Moreover, because the correlation coefficient between nDCG and recall is higher than those between nDCG and precision, and nDCG and recall, it can be said that recall affects our technique the most. Therefore, our technique is more suitable with high-recall impact analysis.

E. RQ 3: Does context-based smell prioritization provide more relevant results than the severity-based smell prioritization?

1) Motivation: The main objective of this study is to propose that context-based smell prioritization can be an efficient method for the support of the prefactoring phase. Therefore, we answer this research question to ascertain whether context-based smell prioritization produces more suitable results for supporting the prefactoring phase than the severity-based approach.

2) Study Design: We applied our technique with the dataset at the class level because it can provide a better ranking, as discussed in RQ 1. We used the combination of VSM and Dyn at the cut point 40 items because this configuration could produce the best results according to RQ 2 in Fig. 6.

3) Results and Discussion: As Fig. 4b shows, the nDCG value of the results from our technique is higher than the nDCG of the baseline. Therefore, after prioritizing the list of smells using our technique, smells that are related to the developers' context were on the higher rank of the list. As a result, the developers can specifically examine the top rank smells directly without specifying which smell is or is not related to their context.

Table III presents a comparison of the top ten ranking between the baseline and our approach. Each row displays the rank of the smell, type of smell, name of module having the smell, and number of issues actually modified in the module. We highlighted the code smell that is actually *relevant* to developers' context, i.e., smells in our oracle. It is apparent that

TABLE III
COMPARISON OF THE RANKING ITEMS BETWEEN THE BASELINE AND OUR APPROACH

(a) ArgoUML: Baseline

Rank	Smell Type	Class Name	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	7.90	3
2	God Class	ProjectBrowser	4.04	7
3	Blob Class	ProjectBrowser	4.04	7
4	Schizophrenic Class	StylePanel	2.43	1
5	God Class	FigNodeModelElement	2.18	4
6	God Class	UMLMutableGraphSupport	1.54	
7	Blob Class	GeneratorCSharp	1.04	
8	God Class	FigEdgeModelElement	0.94	3
9	God Class	ExtensionMechanismsHelperMD	0.91	1
10	God Class	CoreFactoryMDRImpl	0.80	1

(c) JabRef: Baseline

Rank	Smell Type	Class Name	Severity	#Issues
1	God Class	BasePanel	10	4
2	God Class	JabRef	10	7
3	God Class	EntryEditor	10	6
4	God Class	JabRefFrame	10	5
5	Data Class	GUIGlobals	10	
6	God Class	Util	5	8
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	JabRef	6.37	7
2	Blob Class	JabRef	6.37	7
3	God Class	EntryEditor	3.99	6
4	Blob Class	EntryEditor	3.99	6
5	God Class	Util	3.04	8
6	God Class	BasePanel	2.81	
7	God Class	ImportFormatReader	2.73	
8	God Class	JabRefFrame	2.64	5
9	God Class	MainTable	1.84	2
10	God Class	BibtexDatabase	1.73	

(e) jEdit: Baseline

Rank	Smell Type	Class Name	Severity	#Issues
1	God Class	jEdit	10	10
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	5
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	16.83	12
2	Blob Class	Buffer	16.83	12
3	God Class	jEdit	14.43	10
4	God Class	BeanShell	11.66	
5	God Class	GUIUtilities	7.88	7
6	God Class	View	7.09	12
7	Blob Class	SearchAndReplace	6.23	9
8	Blob Class	JEditTextArea	6.08	5
9	God Class	TextAreaPainter	5.46	3
10	God Class	Gutter	5.28	5

(g) muCommander: Baseline

Rank	Smell Type	Class Name	Severity	#Issues
1	God Class	JnlpTask	10	
2	God Class	StatusBar	7	1
3	God Class	WindowManager	6	2
4	God Class	FileTable	6	14
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	4

(h) muCommander: Our approach

Rank	Smell Type	Class Name	CRI	#Issues
1	God Class	FileTable	12.01	14
2	God Class	FolderPanel	7.85	7
3	God Class	AbstractFile	7.44	2
4	Schizophrenic Class	LocalFile	6.34	7
5	God Class	StatusBar	3.98	1
6	Schizophrenic Class	StatusBar	3.98	1
7	God Class	WindowManager	3.78	2
8	God Class	FileFactory	3.74	3
9	Schizophrenic Class	CommandManager	3.66	1
10	God Class	CommandManager	3.66	1

the result of the baseline tends to be mixed between relevant and irrelevant smells while our approach tends to put relevant smells to the top of the list. Moreover, the number of related issues of each smell in the baseline tends to be scattered, i.e., 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 high number of related issues to the top of the list because our assumption is that solving code smell with the high number of related issues would be more beneficial to developers, i.e., improving understandability and extendibility of the source code, than solving the code smell with the low number of related issues.

We further analyzed 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 `org.gjt.sp.jedit.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 with many issues that must be resolved by the developers. We confirmed it by investigating the actual changes made

during revision 4.2–4.3. Results showed that, out of 150 issues, 12 issues were implemented in this class which contains this God Class smell. Therefore, if the developers realized the importance of this smell and fixed it, then it might be able to facilitate their implementation, i.e., improving the understandability of source code for 12 issues.

In contrast, when considering the 1st rank of muCommander project in the baseline that is the God Class smell of the class `com.mucommander.ant.jnlp.JnlpTask` was ranked 27th in the result from our technique. This is also true because our technique predicted that this smell is not related in any way with any issue in the issue tracking system. Our technique assigned zero *CRI* to this smell. We also investigated the actual change and found no issue 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 developers' context can support developers' implementation more than the original order such as severity.

F. Threats to Validity

One biggest threat to construct validity is the oracle dataset that we used. Because the oracle was extracted based on the changes in version control repositories, it might lack some important modules that were not modified by solving the given issues but should be refactored. A typical example is the modules that were needed to understand to make changes; refactoring them contributes to improve understandability. Also, one can improve the extendibility of a module without refactoring it directly, but refactor another related module instead. However, it is not easy to extract such modules in an empirical way. Version control and issue tracking repositories do not contain such information in a formal way. Although the use of fine-grained interaction history of developers such as Mylyn logs [25] might be useful to confirm the context of developers more explicitly, collecting the histories of such type is another challenge [26].

IV. Discussion

A. Context vs. Severity: Which is Better in General?

Many studies have been proposed to detect or prioritize code smells based on the severity, e.g., how bad the smells are. Such severity-based techniques are very useful and appropriate when the development team wants to improve the overall quality of the software. Solving code smells that contribute most to the decay of the source code is likely to be more useful than solving those which have little effect on the system.

However, in the real world situation, development teams have limited time for delivery the product, not to mention improved quality of the source code. Many reports have described that developers refactor their code only when they must modify source code [9]. In this situation, solving the most severe code smell might not be the best option because solving such smells might not contribute to supporting developers' implementation directly. 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.

No clear distinction exists to indicate which smell prioritization type is the best approach. Developers can select the most appropriate approach depending on their situation.

B. Can Our Approach Predict the Smells to be Refactored?

The aim of our technique is not to *predict* the smells that will be refactored by developers. One might consider conducting an empirical study to confirm whether or not the provided ranking of smells by the proposed technique fits the modules to be refactored. For example, Bavota et al. [10] investigated the relation between the quality of a software product and related refactoring activities and provided a dataset of the *refactored* class level smells. We measured the accuracy of rankings of ArgoUML ver. 0.22–0.24 using the number of refactorings applied to smells as the oracle extracted from this dataset provided by Bavota et al. [10]. As a result, all of the techniques produced the rankings of poor quality. We obtained the nDCG values of 0.32, 0.34, 0.45, 0.23, and 0.22, respectively, for the baseline (severity-based technique), VSM, LSI, VSM+Dyn, and LSI+Dyn (context-based techniques). Because developers in the current development style do not solve smells frequently [10], most recommended smells on the rankings were not actually refactored even if they *should be* refactored. Our aim is not to predict the smells that current software developers are solving but to recommend the smells that have enough impact on their development.

V. Related Work

Because code smell detectors tend to generate a huge number of smells, many techniques have been proposed to reduce the number of code smell detection results. Fontana et al. [27] proposed a technique to reduce the number of code smell detection results by application of strong and weak filters, but the method limits the technique to code smells of five types. Arcoverde et al. [28] presented four heuristics to prioritize code smells based on the relevance of potential contribution to software architecture degradation. Ratiu et al. [29] used historical information to filter out the entities that might not have a negative effect from the original results detected using 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. Fontana et al. [30] also introduced Code Smell Intensity index as a criterion to prioritize code smells. The approach devotes attention specifically to the most severe smell instances, but it is limited to code smells of seven types, whereas our approach specifically examines the more context-related smell instances. Moreover, it is not limited to specific smells. The novel point of our approach compared to them 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.

Some existing techniques also use the context of developers to detect code smells [31], [32], but they can be regarded as supporting the *postfactoring* phase because such techniques detect code smells during the source code editing process used of developers.

Vidal et al. [33] proposed a technique to prioritize code smells based on three criteria: historical information of component modification, relevance of type of code smell from developers' 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 manual work specifying scenarios and related source code components. In contrast, our technique processes this step automatically. Consideration of other factors such as past modification of components or code smell type preferences of developers remains as a subject of our future work.

VI. Conclusion

As described in this paper, we proposed a technique for prioritizing code smell detection results by consideration of developers' current context. The result of our technique is a list of prioritized smells based on the relevance to the developer context. The more relevant to the developer context, the higher the rank that smell is placed on the list. Therefore, our approach can assist the developers 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. Our preliminary evaluation indicated that our technique can be useful.

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

Acknowledgment

This work was partly supported by a JSPS Grant-in-Aid for Scientific Research (Nos. 15H02685 and 15K15970).

References

[1] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Proc. ICSM*, 2012, pp. 306–315.

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

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

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

[5] M. Tufano, F. Palomba, G. Bavota, and R. Oliveto, "When and why your code starts to smell bad," in *Proc. ICSE*, 2015, pp. 404–414.

[6] T. Vale and I. S. Souza, "Influencing Factors on Code Smells and Software Maintainability: A Cross-Case Study," in *2nd Workshop on Software Visualization, Evolution and Maintenance*, 2014.

[7] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," *Journal of Software Maintenance and Evolution*, vol. 23, no. 3, pp. 179–202, 2011.

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

[9] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" in *Proc. ICSE*, 2015, pp. 393–402.

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

[11] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. ICSE*, 2013, pp. 672–681.

[12] A. Yamashita and L. Moonen, "Do developers care about code smells? An exploratory survey," in *Proc. WCRE*, 2013, pp. 242–251.

[13] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[14] S. A. Bohner and R. S. Arnold, *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[15] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *Proc. MSR*, 2012, pp. 430–440.

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

[17] Intooitus, "inFusion," http://www.intooitus.com/products/infusion.

[18] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. Kagdi, "A dataset from change history to support evaluation of software maintenance tasks," in *Proc. MSR*, 2013, pp. 131–134.

[19] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. Huffman Hayes, A. Dekhtyar, D. Manukian, S. Hossein, and D. Hearn, "Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions," in *Proc. ICSE*, 2012, pp. 1375–1378.

[20] B. Dit, E. Moritz, and D. Poshyvanyk, "A TraceLab-based solution for creating, conducting, and sharing feature location experiments," in *Proc. ICPC*, 2012, pp. 203–208.

[21] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of IR techniques," *ACM Transactions on Information Systems*, vol. 20, no. 4, pp. 422–446, 2002.

[22] B. Croft, D. Metzler, and T. Strohman, *Search Engines: Information Retrieval in Practice*, 1st ed. Addison-Wesley Publishing Company, 2009.

[23] G. Salton and M. J. McGill, *Introduction to modern information retrieval*. McGraw-Hill, Inc., 1983.

[24] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[25] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proc. AOSD*, 2005, pp. 159–168.

[26] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?" in *Proc. ECOOP*, 2012, pp. 79–103.

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

[28] R. Arcoverde, E. Guimaraes, I. Macia, A. Garcia, and Y. Cai, "Prioritization of Code Anomalies Based on Architecture Sensitiveness," in *Proc. 27th Brazilian Symposium on Software Engineering*, 2013, pp. 69–78.

[29] D. Ratiu, S. Ducasse, T. Girba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proc. CSMR*, 2004, pp. 223–232.

[30] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a Prioritization of Code Debt : A Code Smell Intensity Index," in *Proc. IEEE 7th International Workshop on Managing Technical Debt*, 2015, pp. 16–24.

[31] S. Hayashi, M. Saeki, and M. Kurihara, "Supporting refactoring activities using histories of program modification," *IEICE Transactions on Information and Systems*, vol. E89-D, no. 4, pp. 1403–1412, 2006.

[32] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1112–1126, 2013.

[33] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, pp. 1–32, 2014, DOI: 10.1007/s10515-014-0175-x.