PAPER

Supporting Proactive Refactoring: An Exploratory Study on Decaying Modules and Their Prediction*

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

SUMMARY Code smells can be detected using tools such as a static analyzer that detects code smells based on source code metrics. Developers perform refactoring activities based on the result of such detection tools to improve source code quality. However, such an approach can be considered as *reactive refactoring*, i.e., developers react to code smells after they occur. This means that developers first suffer the effects of low-quality source code before they start solving code smells. In this study, we focus on *proactive refactoring*, i.e., refactoring source code before it becomes smelly. This approach would allow developers to maintain source code quality without having to suffer the impact of code smells. To support the proactive refactoring process, we propose a technique to detect *decaying modules*, which are non-smelly modules that are about to become smelly. We present empirical studies on open source projects with the aim of studying the characteristics of decaying modules. Additionally, to facilitate developers in the refactoring planning process, we perform a study on using a machine learning technique to predict decaying modules and report a factor that contributes most to the performance of the model under consideration.

key words: *code quality, code smells, refactoring*

1. Introduction

Code smells were introduced as an indicator of a design flaw or problem in the source code [3]. Their definitions were presented in a descriptive language; therefore, several studies have interpreted them in a formal manner. For example, Lanza and Marinescu use source code metrics to form conditions and combine each condition with logical operations to detect code smells [4]. Several studies have found that code smells are related to different aspects of software development, such as maintainability [5]–[7]. Therefore, it is advisable to remove code smells by a refactoring operation that can improve the quality of the source code and avoid undesirable consequences.

However, such an approach can be considered as *reactive refactoring*. We term it reactive because developers basically react to code smells after they occur in the system. An advantage of this approach is that it allows developers to focus on the most problematic part of the source code (smelly code) rather than handling every part of the source code, which may be impractical in real life. However, an important disadvantage of such an approach is that, by the time developers are warned of the code smells by the tools, they have already suffered the bad effect of the code smells, e.g., low readability and understandability. In other words, developers cannot prevent code smells from occurring in the system.

To deal with the problem, in this study, we shift our focus to *proactive refactoring*, which is the action of refactoring source code before it becomes smelly. Both proactive and reactive refactorings are the same in a sense that developers apply a tool to source code and perform refactoring. However, the difference is the target of both types, i.e., reactive refactoring focuses on the target that affects maintainability. In contrast, proactive refactoring focuses on the target that has not affected maintainability but is likely to do so in the future. Similar to the manner in which code smells are considered as candidates for performing reactive refactoring, we propose the idea of *decaying modules* as candidates for performing the proactive refactoring. A decaying module is a non-smelly module that is about to become smelly. We measure the quality index of a module by calculating the *module decay index* (*MDI*) that indicates the closeness of a module to be becoming smelly. MDI is used to measure how bad a non-smelly module is, whereas *severity* [8] and *smell intensity index* [9] are used to measure how bad a smelly module is. The idea of decaying modules can be mainly used in two ways. First, it can be used to warn developers of the modules that are getting close to becoming smelly so that developers can take preventive measures. Second, it can be used to indicate the overall quality of the entire system so that developers can view the overall status of a project, and not just the smelly modules. This would allow developers to develop more proactive strategies for controlling software quality. In other words, developers can focus on preventing code modules from becoming smelly rather than waiting until they become smelly and then resolve the code smells.

This paper is revised based on our previous studies [1], [2]. The main contributions of this study are as follows.

1. We propose the concept of decaying modules by observing MDI.
2. We present empirical studies regarding the characteristics of decaying modules.
3. We report experiments on using a machine learning approach to predict decaying modules and show that

Manuscript received January 1, 2015.
Manuscript revised January 1, 2015.

†The author is with School of Computing, Tokyo Institute of Technology, Tokyo, 152–8550 Japan.
††The author is with Department of Software Engineering, Faculty of Science and Technology, Nanzan University, Nagoya, 466–8673 Japan.

*This paper is revised based on [1] and [2], which appeared in proceedings of the third International Workshop on Refactoring and proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution, © 2019 IEEE.

a) E-mail: hayashi@c.titech.ac.jp

DOI: 10.1587/trans.E0.??.1

Copyright © 200x The Institute of Electronics, Information and Communication Engineers

developers' context, even if it is estimated by automated impact analysis techniques, can improve the performance of the prediction model.

4. We present an investigation on how we can improve the impact analysis technique to enhance the performance of the decaying module prediction model further.

The remainder of this paper is organized as follows. Section 2 presents the definition of decaying modules. Section 3 presents our empirical studies on decaying modules. Section 4 presents an experiment on the prediction approach. Section 5 discusses the threats to the validity of this study. Section 6 discusses how decaying modules can be fit into refactoring prioritization. Section 7 presents the related work. Section 8 concludes this paper.

2. Concept of Decaying Modules

2.1 Motivation

To describe the motivation behind the concept of decaying modules, we use a dental metaphor because it has been used to explain the idea of software quality, e.g., floss and root-canal refactoring [10]. We draw a parallelism between the process of removing code smells and tooth decay treatment, i.e., developers handle code smells after they occur; this is similar to the act of patients taking care of their tooth decays after they happen. On the contrary, the process of preventing code smells is comparable to the process of brushing one's teeth every day, i.e., developers prevent code smells from occurring by refactoring modules that are not yet smelly; this is similar to the act of brushing one's teeth to remove small food particles. However, it is impractical to refactor every module in the system; therefore, we define the concept of *decaying module* to represent a module without code smell but with progressively worsening quality. If a tooth without tooth decay is building up plaque, then the plaque becomes the primary cause of tooth decay in the future. Considering the aforementioned example, the concept of decaying modules can be used to support proactive refactoring, i.e., developers can refactor decaying modules to prevent them from becoming smelly.

The main difference between code smells and decaying modules is that decaying modules do not directly affect the maintainability of the source code. Additionally, while both code smells and decaying modules are the targets of refactoring, the underlying reasons are different. Code smells require refactoring because they tend to reduce the maintainability of the system, whereas decaying modules need refactoring to prevent such effects from occurring.

The main objective of detecting decaying modules is to identify modules with a risk of becoming smelly in the future. This may be related to an empirical study by Tufano et al., which reported that modules that are likely to be affected by code smells are characterized by specific metrics' trends [11]. Accordingly, we suspect that observing the distance between the metric values and the thresholds indicating that the module will be considered as smelly would enable us to generate a set of modules that are likely to be affected by code smells.

2.2 Definition

We define a decaying module as a module that is getting closer to becoming smelly during a certain period. One way to detect code smells is to use a metric-based strategy. Such a strategy detects code smells by considering whether particular metrics exceed their corresponding thresholds. In this case, we can use a formula to detect a decaying module. A decaying module would be a module whose metric values have not exceeded their thresholds. Therefore, we refer to code smell detection strategies that use multiple symptoms (or conditions) to detect code smells. Each symptom is determined by whether a source code metric, e.g., lines of code (LOC), exceeds a specific threshold. Logical operations are then used to combine all the symptoms and identify code smells. For example, a metric-based strategy to detect God Class [4] can be reformulated as follows:

$$\begin{aligned}\text{God Class} = {} & (V_{\text{ATFD}} \geq T_{\text{ATFD}}) \wedge \\ & (V_{\text{WMC}} \geq T_{\text{WMC}}) \wedge (V_{\text{TCC}} \leq T_{\text{TCC}}),\end{aligned}$$

where V_{ATFD}, V_{WMC}, and V_{TCC} are the values of access to foreign data (ATFD), weighted method count (WMC), and tight capsule cohesion (TCC), respectively. Similarly, T_{ATFD}, T_{WMC}, and T_{TCC} are the thresholds of ATFD, WMC, and TCC, respectively.

This strategy includes three symptoms, where each symptom is determined by a corresponding metric. For example, the metric ATFD measures how many foreign attributes are used by a class. The higher the ATFD is, the more likely it is a class to be a God Class. Therefore, the operation $\geq$ is used to determine if the value of ATFD exceeds the threshold; if the former is true, the symptom holds. On the contrary, the metric TCC represents the degree of cohesiveness of a class. A lower TCC value indicates a less cohesive class; this means that the class is more likely to be a God Class. Here, the operation $\leq$ is used to determine if the value of TCC is less than the threshold; if the former is true, the symptom holds. All the symptoms are then conjunctively combined using *and* operations to determine the God Class code smell.

Then, we define the following metrics to measure the closeness of a module to be becoming smelly.

2.2.1 Percentage of Symptom

First, we define a *percentage of symptom* (*PS*) to measure how close the current value of a specific metric is to its threshold. PS is measured by metric m of module x, which is defined as:

$$PS_m(x) = \begin{cases} \min\{\,1, V_m(x)/T_m\,\}, & \text{if comparator is} \geq \\ \min\{\,1, T_m/V_m(x)\,\}, & \text{if comparator is} \leq, \end{cases}$$

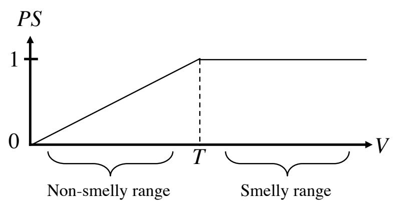


Fig. 1 Domain and range of percentage of symptom (PS).

where V_m is the value of metric m and T_m is the threshold value of metric m. The min function sets the maximum value of PS to one. For example, PSs measured by ATFD and TCC of module x can be defined as

$$PS_{\text{ATFD}}(x) = \min\{\,1, V_{\text{ATFD}}(x)/T_{\text{ATFD}}\,\},$$
$$PS_{\text{TCC}}(x) = \min\{\,1, T_{\text{TCC}}/V_{\text{TCC}}(x)\,\}$$

because the operator that determines their symptom are $\geq$ and $\leq$, respectively.

Figure 1 shows the domain and range of PS. The higher the PS is, the closer it is to complete the condition of the symptom. PS of one indicates that the symptom is completed.

2.2.2 Module Decay Index

We define MDI as an indicator of how close a module is to becoming a smell. A MDI is defined for each smell; MDI of smell s is calculated by averaging the PS of each symptom measured by metric m where $M_s = \{\ldots, m, \ldots\}$ as

$$MDI_s(x) = \frac{1}{|M_s|} \sum_{m \in M_s} PS_m(x).$$

For example, the God Class MDI of module x is defined as

$$MDI_{\text{God Class}}(x) = \frac{PS_{\text{ATFD}}(x) + PS_{\text{WMC}}(x) + PS_{\text{TCC}}(x)}{3}.$$

The higher the MDI is, the closer the module is to become a smell. MDI of one indicates that the module is a smell.

MDI is also comparable to the severity defined by Marinescu [8]. Severity is a metric ranged $[1, 10]$, and it is used to measure how bad a smelly module is. MDI, however, ranges from $[0, 1)$ with the purpose of measuring how bad a non-smelly module is. In other words, MDI can also be considered as the severity of non-smelly modules.

2.2.3 Decaying Module

In general, a decaying module can be defined as a module whose MDI has increased over a period of time. In this paper, we refer to a module as a decaying module when its current MDI has increased from the previous release. More precisely, a decaying module m regarding a smell s at release n is defined as

$$Decaying_s(m_{@n}) = MDI_s(m_{@n}) > MDI_s(m_{@n-1})$$

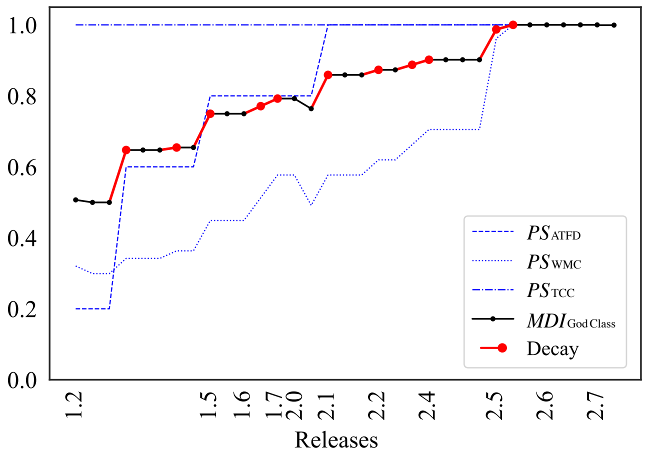


Fig. 2 The decaying process of *AmbariServer*.

where $m_{@n}$ denotes a module m at release n. We also use $Decaying(m_{@n})$ with omiting the smell type s if it is obvious.

As a first step in defining decaying modules, we use God Class as a subject owing to its simple detection strategy and the fact that it is a code smell that is often studied in this research area [12]. However, this approach is also applicable to other types of code smells that are determined by using logical operations to combine all symptoms, where each symptom holds if particular metrics exceed their thresholds such as Data Class or Brain Class defined by Lanza and Marinescu [4].

Example. Figure 2 shows the evolution of *AmbariServer* class module in Apache Ambari project[†]. We plotted the values of PS_{ATFD}, PS_{WMC}, PS_{TCC}, and $MDI_{\text{God Class}}$ across 33 releases. Also, we highlighted the decays of the module, i.e., the increases of MDI value, in red. This module went through 11 decays between releases during its evolution and eventually became God Class. Of the metrics used in detecting God Class, the TCC value was lower than the threshold ($T_{\text{TCC}} = 1/3$) from the beginning, and its PS value (PS_{TCC}) was always 1 in this plot. The ATFD and WMC value gradually increased as the module evolves, and the MDI value gradually increased accordingly. Finally, in release 2.5.1, the module met the conditions of three metrics and was detected as a God Class. In this way, the fact that a module decays continuously means that it becomes gradually closer to a smell, and it may finally become a smell.

Smells are related to not only maintainability issues but also faults, which adversely affects software quality [13], [14]. This fact may lead to a hypothesis that refactoring modules that have already been smelly might be too late. Also, removing smells is tough. Literature shows developers have difficulties to devote dedicated resources to removing smells outside of their normal development activities, and it is common to refactor modules during modifying them in their activities [15]. To adapt to such a style, proactive refactoring is effective to apply refactorings preventively be-

[†]https://github.com/apache/ambari/blob/trunk/ambari-server/src/main/java/org/apache/ambari/server/controller/AmbariServer.java

Table 1 Dataset information

Project	Period	# releases
Accumulo	2012/03/27–2018/07/16	30
Ambari	2013/02/04–2018/08/22	33
Derby	2005/08/01–2018/05/05	28
Hive	2010/10/27–2018/05/18	33

fore target modules become smelly, rather than traditional reactive refactoring.

We think that decaying modules can be refactored by applying the same refactoring patterns to code smells. Decaying modules can be considered as weak instances of code smells, whose metric values do not exceed thresholds to be regarded as problematic. Just as refactoring for code smells reduces the complexity, which leads to bringing the metric values below the thresholds, refactoring for decaying modules also reduces the complexity to make the metric values away from the thresholds and prevent the future generation of smells. For example, for God Class smells, we mitigate concentrated responsibilities by extracting classes. Decaying in terms of God Class can be considered a sign of concentrated responsibilities or bloated individual responsibilities. Therefore, extracting classes should also be appropriate refactoring operations for them before a problematic situation is happening.In addition, there may be cases where applying refactorings is difficult because of the difficulty in splitting responsibilities, as is the case with normal refactorings of code smells.

3. Empirical Study of Decaying Modules

This empirical study aims to conduct analyses on the characteristics of a decaying module from different perspectives. The empirical study was conducted with the following research questions (RQs). Their details are explained later.

RQ$_1$: How many decaying modules are present in each release compared to the modified modules?
RQ$_2$: What are the future characteristics of the decaying modules compared to the non-decaying ones?

3.1 Experimental Setup

Experimental Implementation. In this study, as a first step in studying decaying modules, we limit the target of the code smell to be the God Class. As discussed earlier, God Class is considered one of the most common code smells studied in this research area. To detect the decaying module, we first used inFusion ver. 1.9.0 as a static analysis tool to calculate the metrics of each module. Then, we calculated the *PS* of each metric and *MDI* of each module. Finally, we classify those modules as decaying modules whose *MDI*s have increased from the earlier release.

Data Collection. In this study, four open source projects: Accumulo[†], Ambari[††], Derby[†††], and Hive[††††] were our subjects. They were selected from a list of active open source projects of The Apache Software Foundation, which is commonly used as a subject for open source software study. The dataset information can be found in Table 1.

3.2 RQ$_1$: How many decaying modules are present in each release compared to the modified modules?

Motivation. We expect that such information can be used as a first step to understand the characteristics of decaying modules. The number of modules is different depending on the size of the project; therefore, analyzing absolute numbers may not be useful. We analyze the ratio of decaying modules to the number of modules that are modified by developers in each release. This is because module modification is the main activity during software development, and it is relatively easy to understand as a comparator.

3.2.1 Study Design

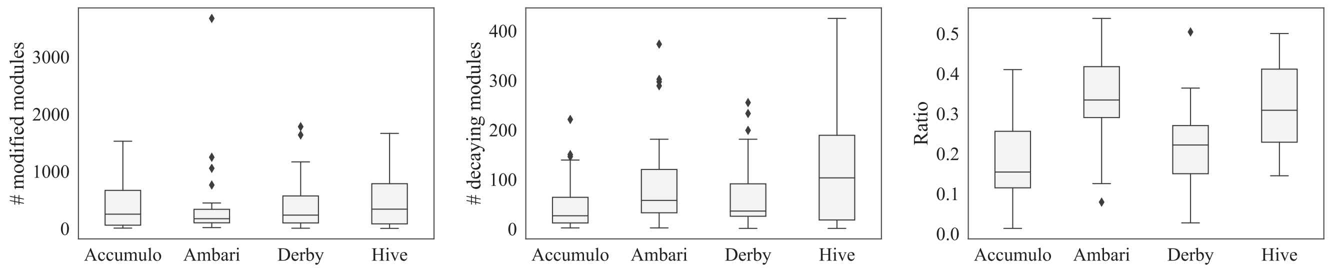
To answer this RQ, we counted the modules that were modified and the decaying modules between each pair of releases. The modules that were modified between each pair of releases were generated using the `git-log` command. We excluded modified modules after they become smelly because we focus on the comparison between modified modules and decaying modules, and our definition of decaying modules does not capture further decaying after modules become smelly. Then, we ran our tool that was explained previously to detect the decaying modules between each pair of releases. Finally, we calculated the ratio between the number of modified and decaying modules.

3.2.2 Results and Discussion

Figure 3 represents the number of modified classes, the number of decaying classes, and the ratio of them, respectively. For the number of modified classes, we can observe similar distributions for all projects except for Ambari, which tends to have the lowest number of modified classes among all projects. For the number of decaying classes, we can see that Hive has the largest distribution among the four projects. For the ratio, we can observe the similar distributions between Accumulo and Derby, and between Ambari and Hive.

To simplify the results, we summarize the averages of each value in Table 2. The second and third columns show the average numbers of the modified and decaying modules, respectively. The last column shows the ratio of the number of modified and decaying classes. For example, in each release in the Accumulo project, 415.45 classes were modified,

[†]https://accumulo.apache.org/
[††]https://ambari.apache.org/
[†††]https://db.apache.org/derby/
[††††]https://hive.apache.org/



(a) # modified modules. (b) # decaying modules. (c) Ratio.

Fig. 3 The number of decaying modules, modified modules, and the ratio between them.

Table 2 Averages of the number of decaying and modified modules

Project	# modified modules	# decaying modules	Ratio
Accumulo	415.45	50.68	0.12
Ambari	369.63	98.47	0.27
Derby	403.59	72.48	0.18
Hive	578.84	132.17	0.23

and 50.68 were decayed. This yields a ratio of 0.12, i.e., for every 100 modified classes, 12 classes were decayed. The ratios are varied for different projects: 0.12 for Accumulo, 0.27 for Ambari, 0.18 for Derby, and 0.23 for Hive. The average ratio in this study is approximately 0.20. In other words, compared to modified modules, 20% will become decaying modules. This result suggests that the decaying modules are not rare problems and may be worth considering as essential problems that the developers should handle by considering that almost 20% of the modified modules will have lower code quality.

In conclusion, approximately 20% of the number of modified modules were decaying modules in each release on average.

3.3 RQ$_2$: What are the future characteristics of decaying modules compared to non-decaying ones?

Motivation. We study the characteristics of decaying modules, especially their future characteristics. We consider two characteristics of decaying modules, i.e., the decaying modules that will be modified and will decay in the future. If the number of decaying modules that will decay in the future is higher than that of non-decaying modules, it may be a sign that decaying modules are important problems that are worth handling.

3.3.1 Study Design

Similar to RQ$_1$, we counted the decaying modules with the following two characteristics: 1) the decaying modules that will be modified in later releases, and 2) the decaying modules that will again get decayed in later releases. Then, we computed the averages of all releases and calculated the ratio of each. For comparison, similar steps were applied to the modules that were modified but did not become decaying modules, i.e., non-decaying modules. Similar to RQ$_1$, we excluded modified modules after they become smelly. We excluded the releases without decaying modules, e.g., minor releases that have only a few modifications, because we cannot compare the ratios of decaying and non-decaying modules.

More precisely, the sets of decaying modules and non-decaying modules at release n used in this study are defined as follows:

$$\begin{aligned} M^{dec}_{@n} &= \{\, m_{@n} \mid \mathit{Decaying}(m_{@n}) \,\}, \\ M^{dec*}_{@n} &= \{\, m_{@n} \mid \neg\mathit{Decaying}(m_{@n}) \wedge \\ &\qquad \mathit{Modified}(m_{@n}) \wedge \neg\mathit{Smelly}(m_{@n-1}) \,\} \end{aligned}$$

where $\mathit{Modified}(m_{@n})$ and $\mathit{Smelly}(m_{@n})$ respectively denote that $m_{@n}$ is modified and smelly at release n. For the given set of target modules $M_{@n}$ at release n, which is either $M^{dec}_{@n}$ or $M^{dec*}_{@n}$, the ratios for 1) and 2) are computed as

$$\begin{aligned} R^{mod}_{@n} &= \frac{|\{\, m_{@n} \in M_{@n} \mid \exists k > n \cdot \mathit{Modified}(m_{@k}) \,\}|}{|M_{@n}|}, \\ R^{dec}_{@n} &= \frac{|\{\, m_{@n} \in M_{@n} \mid \exists k > n \cdot \mathit{Decaying}(m_{@k}) \,\}|}{|M_{@n}|}. \end{aligned}$$

To confirm whether the results are statistically significant, we conducted statistical tests with the following null and alternative hypotheses:

H$_{01}$ / H$_{a1}$: The ratios of decaying modules that will be **modified** in later releases are *not higher* / *higher* than the ones of non-decaying modules.

H$_{02}$ / H$_{a2}$: The ratios of decaying modules that will get **decayed** in later releases are *not higher* / *higher* than the ones of non-decaying modules.

Then, we used the Wilcoxon signed-rank test, which is a non-parametric statistical hypothesis test, to determine any difference between the ratios of decaying and non-decaying modules. We did not use any parametric tests because we did not assume a specific distribution for the ratios.

3.3.2 Results and Discussion

Figure 4 illustrates the results of our study. The bars represent the values of non-decaying and decaying modules. The values in Fig. 4a represent the ratio of modules that will be modified in later releases. We can observe only small, if

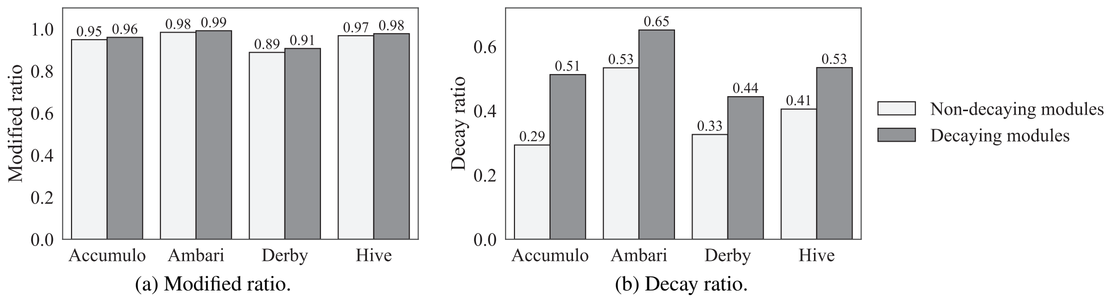


Fig. 4 Averages of the number of decaying and non-decaying classes that will be modified and decay.

Table 3 Results of Wilcoxon signed-rank test

Project	H_{01}	H_{02}	H_{03}
Accumulo	0.300	0.013	<0.001
Ambari	0.204	<0.001	<0.001
Derby	0.070	<0.001	<0.001
Hive	0.074	<0.001	<0.001

any, differences in the projects. For example, the biggest difference is in the values of the Derby project, indicating that approximately 89% and 91% of the non-decaying and decaying modules were respectively modified in later releases. Table 3 shows the result of Wilcoxon signed-rank test. The cells with p values less than 0.05 are highlighted in gray. It can be seen that the p values of all four projects are higher than 0.05. This means that we fail to reject the null hypotheses ($\alpha = 0.05$) that the ratios of decaying modules that will be modified in later releases are not higher than the ones of non-decaying modules.

Figure 4b illustrates the ratios of modules that decayed in later releases. For example, in the Ambari project, 53% of the non-decaying modules decayed in later releases. However, the number of decaying modules is as high as 65%. The ratios are clearly higher than those of their non-decaying counterparts. In other words, decaying modules are more likely to get decayed again in later releases. Additionally, it can be observed that more than half of the decaying modules in each project, except for Derby, decayed again in later releases. The results of the Wilcoxon signed-rank test listed in Table 3 indicate that the results are statistically significant ($\alpha = 0.05$). In such cases, we can reject the null hypotheses that the ratios of decaying modules that will get decayed in later releases are not higher than the ones of non-decaying modules.

With these pieces of evidence, we can conclude that, although we could not observe the difference of the ratios of decaying and non-decaying modules that will be modified in later releases, the ratios of decaying modules that will get decayed in later releases are higher than those of non-decaying modules. In other words, we can conclude that decaying modules are the modules that have got closer and will likely get more close to becoming smelly modules. Therefore, we argue that they are significant problems that should be handled before they affect source code quality.

In conclusion, while no difference of the ratios of decaying and non-decaying modules that will be modified in later releases was detected, decaying modules are more likely to get decayed in later releases.

4. Predicting Decaying Modules

As discussed in the previous section, decaying modules are important problems that are worth considering. However, a decaying module only represents past and present information, i.e., whether a module has decayed from the last release. As shown in the previous section, although decaying modules are more likely to get decayed again in the future, they also have a chance of not getting decayed in the next release, i.e., their quality may be improved. Therefore, knowing that a decaying module will still be a decaying module in the next release (its quality will get even lower) can support developers to determine whether it should be refactored. In other words, the prediction approach can help developers prioritize decaying modules for refactoring.

When considering modules for refactoring, one possible way is to refactor modules with high MDI value. However, as mentioned earlier, modules that tend to affect the maintainability of the system are the modules with code smells, i.e., smelly modules. In contrast, modules without code smells, i.e., clean modules, do not have such an effect. Modules with high MDI value, while they are close to becoming smelly, are not smelly. Therefore, we do not consider them as the target of refactoring regarding the problems of source code. On the other hand, modules whose MDI value is increasing are more likely to become smelly modules in the future. Consequently, such modules will require refactoring. Therefore, it is recommended to refactor modules that MDI value is getting higher rather than focusing on the modules with high MDI value.

In this study, we consider the use of a machine learning approach to predict modules that will become decaying modules in the next release. Such an approach can be implemented by considering the characteristics of each module, e.g., code quality metrics, as predictor variables and whether a module will get decayed in the next release (*True* or *False*) as a response variable. This technique is widely used for defect prediction, where the characteristics of each module are used to predict whether the module is defective [16]. Since an example that is similar to this study is the work by Pantiuchina et al. [17] that proposes to predict code smells,

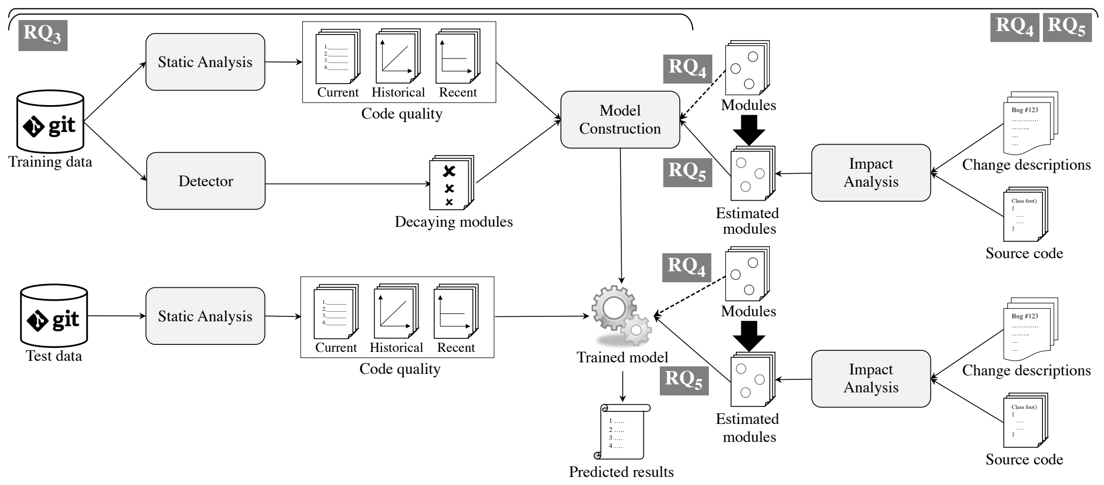


Fig. 5 Decaying modules prediction approach.

we can use it as a baseline approach.

A scenario that is suitable to this case is the prefactoring phase, wherein developers refactor the source code to facilitate future implementation [18]. In this scenario, developers can use the prediction result of the modules that will get decayed in the next release to plan their refactoring strategy. In the prefactoring phase, developers usually have an idea of the changes that they plan to make, i.e., they know the modules that they are going to change. Such information is obtained from the concept location and the impact analysis phases, wherein developers identify the code component that needs to be modified to satisfy change requirement [18]. We define this as *developers' context* [19]; it is similar to the *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" [20]. Predictions made by the trained model can alert developers to specific modules. If a future decay is predicted for modules to be modified or related to the ongoing tasks, developers examine whether such decay can actually occur based on their future implementation plan. Based on the examination, the module may be included in the prefactoring process to prevent a future generation of smells. Perfect prediction is difficult in machine learning, and predictions may include falses. Rather than blindly accepting all predictions, refactoring activities can be facilitated by using predictions as a supplement to improve developers' awareness of potential smells. In addition in this case, we suspect that developers' context may contribute to improving the performance of prediction models. The underlying reason is that the modules that will be modified are more likely to get decayed than the modules that will not be affected by any changes.

Also, we investigate the possibility of the use of automated impact analysis techniques to estimate such a developers' context for the decaying module prediction. This investigation is aimed to meet a realistic situation that the developer who uses the system may not have perfect knowledge of the locations of the changes.

In this section, we present an empirical study with the following RQs. Details of each RQ will be discussed in the later subsections.

RQ$_3$: Can we use an existing technique to predict decaying modules?

RQ$_4$: Can developers' context improve prediction performance?

RQ$_5$: Can developers' context estimated by IR-based impact analysis technique improve prediction performance?

RQ$_6$: How can we further improve the performance of the prediction model?

4.1 Experimental Setup

The idea of this approach is to use the information of the current release to predict modules that are going to decay in the next release. Figure 5 shows an overview of the approach.

Baseline. We set a baseline model inspired by the work by Pantiuchina et al. that was proposed to predict modules that will become smelly [17]. The left part of Fig. 5 represents an overview of the baseline approach. Three sets of variables are used as predictor variables: current code quality, historical code quality trend, and recent code quality trend. The response variable is whether the module will get decayed in the next release, i.e., after implementing changes that are planned for the release.

Variables Construction. We calculated three types of predictor variables: current code quality, historical code quality trend, and recent code quality trend. Historical code quality trend represents the trend of each module's quality from its creation until the current release, whereas recent code quality trend represents the trend of each module's quality from the earlier release until the current release. These two types of variables can be used to supplement each other in case when the quality of a module decreased in the past but increased during recent activities. For the variable: current code quality, we use inFusion ver 1.9.0 as a static analysis tool to calculate 34 metrics in addition to the proposed *MDI*.

Table 4 Metrics used in this study

Label	Metric Name
ALD	Access to Local Data
AMW	Average Method Weight
ATFD	Access to Foreign Data
BOVR	Base-class Overriding Ratio
BUR	Base-class Usage Ratio
CBO	Coupling Between Objects
CPFD	Capsules Providing Foreign Data
CRIX	Criticality Index
CW	Class Weight
DIT	Depth of Inheritance Tree
MDI	Module Decay Index
HIT	Height of Inheritance Tree
ICDO	Incoming Coupling Dispersion for an Operation
LCC	Loose Capsule Cohesion
LCOM	Lack of Cohesion of Methods
LDA	Locality of Data Accesses
LOC	Lines of Code
NAS	Number of Added Services
NOA	Number of Attributes
NOACCM	Number of Accessor Methods
NOAM	Number of Abstract Methods
NOCHLD	Number of Children
NOM	Number of Methods
NOPRTA	Number of Protected Attributes
NOPRTM	Number of Protected Methods
NOPUBA	Number of Public Attributes
NOPUBM	Number of Public Methods
NOVRM	Number of Overriding Methods
OCDO	Outgoing Coupling Dispersion for an Operation
PNAS	Percentage of Newly Added Services
RFC	Response for a Class
SPIDX	Specialization Index
SUM_LOC	Sum of Lines of Code
TCC	Tight Capsule Cohesion
WOC	Weighted Operation Count

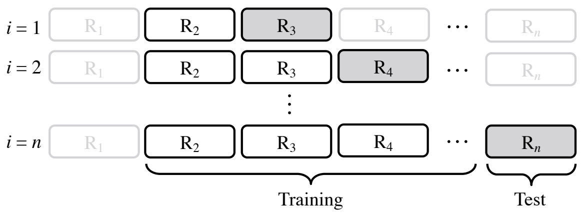


Fig. 6 Training and test data separation.

The metrics used in this study are shown in Table 4. For the variable: historical code quality trend, we compute the regression slope line fitting the value of each metric from the first to the current release. Finally, for the variable: recent code quality trend, we compute the regression slope line fitting the value of each metric from the earlier to the current release. As a result, we have a total of 105 (35 + 35 + 35) predictor variables. Consequently, we used the technique proposed in the previous section to detect decaying modules and record them as response variables.

Data Separation. We separated data into training and test sets as shown in Fig. 6. Initial releases of the source code were skipped because there is insufficient information to calculate the slope of each metric. In the first iteration (i = 1), we trained the model on release 1 and performed a prediction test on release 2. Then, in the next iteration (i = 2), we trained the model on release 1–2 and performed a prediction test on release 3. The iterations were repeated for the whole dataset. This strategy simulates the situation where developers use all of the available data to train the model. Such data may be fewer at the beginning of a project but would increase along with the development process.

Table 5 Metrics removed in each project

Metric	Accumulo	Ambari	Derby	Hive
ALD	✗	✗	✗	✗
CPFD	✗	✗	✗	✗
LCC	✗	✗	✗	✗
LDA	✗	✗	✗	✗
NOPUBM	✗			
OCDO	✗	✗	✗	✗
PNAS	✗	✗	✗	✗
RFC	✗	✗	✗	✗
SPIDX	✗	✗	✗	✗
SUM_LOC	✗	✗	✗	✗
WOC	✗	✗	✗	✗
Historical CPFD Trend				✗
Historical LCC Trend	✗	✗	✗	✗
Historical OCDO Trend	✗	✗	✗	✗
Historical PNAS Trend				✗
Historical SUM_LOC Trend		✗	✗	✗
Historical WOC Trend		✗		✗
Historical RFC Trend		✗		
Recent LCC Trend	✗	✗		✗
Recent OCDO Trend	✗	✗	✗	✗
Recent SUM_LOC Trend	✗	✗	✗	✗
Recent WOC Trend		✗		
# removed variables	16	19	15	19

Data Preparation. Next, we performed a correlation-based feature selection technique to minimize collinearity among the predictor variables. For each pair of variables having Spearman ρ higher than 0.8, we removed one of the variables. The technique was repeatedly conducted until there was no pair of variables that met the criteria. The removed variables are shown in Table 5. Additionally, the dataset that we use in this study can be considered as imbalanced, i.e., the number of decaying modules is only a small proportion of all the modules. To avoid the problem of imbalanced data affecting the performance of the prediction models, we applied a sub-sample technique to the dataset. Note that we did not use any modules in our training and test sets after they become smelly, similar to RQ_1 and RQ_2.

Model Construction. Finally, we constructed prediction models using the random forest and calculated the performance of the prediction models by applying them to the data in the test sets. We kept default values for the model parameters. Optimizing and analyzing such parameters remains our future work.

Data Analysis. In this study, we use the area under the curve (AUC) of the receiver operating characteristic (ROC) plot, which is commonly used for evaluating and comparing the performance of the machine learning model. AUC ranges from 0 to 1. Higher AUC indicates better performance of the prediction model. Its value above 0.5 indicates that the model performs better than random guessing.

4.2 RQ_3: Can we use an existing approach to predict decaying modules?

Motivation. An example that is similar to this study is the work by Pantiuchina et al. [17] that proposes to predict code smells. Their work uses source code quality to predict whether a module is likely to be affected by code smells in

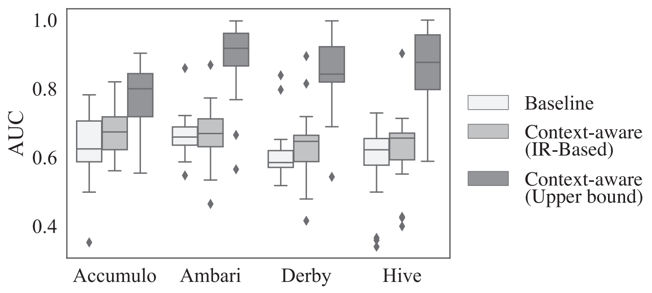


Fig. 7 AUC values of baseline and context-aware models.

the future. Therefore, in this study, our aim is to investigate whether their approach is also applicable to predicting decaying modules. If such an approach can successfully apply to predicting decaying modules, we can use its result to support developers when selecting targets for refactoring.

4.2.1 Study Design

We use the baseline model explained in the previous subsection to measure prediction performance. As aforementioned, AUC value above 0.5 indicates that prediction performs better than random guessing. Therefore, if the median of AUC measured by the baseline is greater than 0.5, we infer that an existing approach can be used to predict decaying modules.

4.2.2 Results and Discussion

Figure 7 shows the performance of the baseline model of each project. The median values are 0.62, 0.66, 0.58, and 0.62 for Accumulo, Ambari, Derby, and Hive, respectively. Median values of all the projects are greater than 0.5; therefore, we can conclude that the existing approach may be suitable for predicting decaying modules as well.

To conclude, an existing approach may be applicable to predict decaying modules.

4.3 RQ_4: Can developers' context improve prediction performance?

Motivation. As mentioned earlier, developers' context refers to modules that developers need to modify. For supporting the prefactoring phase, we suspect that developers' context may contribute to improving the performance of prediction models. The main reason is that modules that developers are going to make changes are more likely to decay than modules that are not going to be impacted by any changes. In this RQ, we conduct a study under the assumption that developers have perfect knowledge of their context. It is noteworthy that even in this situation, the prediction approach is still necessary because we cannot identify decaying modules only from developers' context. The underlying reason was shown in the result of RQ_1, where only approximately 19% of the modified modules were decaying modules in each release. However, assuming perfect knowledge of developers may not be a realistic setting because the situation that developers know every module to be modified beforehand is rare in large-scale projects. Nevertheless, the main purpose of this RQ is to examine the potential of using developers' context to improve the prediction model. In other words, we want to know the upper bound performance of using the developers' context rather than its practical performance.

4.3.1 Study Design

We compare the performance of two models: baseline and context-aware (upper-bound). The baseline model uses the variables described previously as predictor variables, whereas the context-aware model uses the developers' context as an extra predictor variable. As aforementioned, we regard developers' context as the modules that the developers intend to modify for the next release, which is shown in the right part of Fig. 5. We use `git-log` command to obtain a list of modules that are modified between two releases. Then, we mark a variable as *True* if the module is modified, and *False* otherwise. To confirm if the results are statistically significant, we conduct the Wilcoxon signed-rank test with the following null and alternative hypotheses:

H_{03} / H_{a3}: Developers' context *does not improve* / *improves* the performance of prediction model.

4.3.2 Results and Discussion

Figure 7 shows the result of our experiment. The box plot shows the values of the performance of the baseline and context-aware (upper-bound) models. As reported in the earlier RQ, median values of the AUC values for the baseline model are 0.62, 0.66, 0.58, and 0.62 for Accumulo, Ambari, Derby, and Hive, respectively. For the context-aware model, the median values are 0.80, 0.92, 0.84, and 0.88 for them, respectively. It can be seen that the context-aware model performs better for every project.

The results of the Wilcoxon signed-rank test are shown in Table 3. It can be seen that the results are statistically significant ($\alpha = 0.05$). Therefore, we can conclude that developers' context can help improve decaying module prediction performance significantly.

In conclusion, developers' context can improve the performance of the decaying module prediction model.

4.4 RQ_5: Can developers' context estimated by IR-based impact analysis technique improve prediction performance?

Motivation. To mitigate the gap between the real situation and the study in RQ_4, which was conducted based on the assumption that the developer has to input perfect information to the system manually, we propose an alternative approach that does not rely on perfect knowledge of developers. By using the results of an automated impact analysis technique to represent developers' context, we suspect that it can also improve the performance of the model without the need of

Table 6 Optimized parameters of IR-based impact analysis techniques

Project	Technique	Cut point
Accumulo	BM25	20
Ambari	BM25	40
Derby	VSM	30
Hive	BM25	10

perfect knowledge from developers. However, since such techniques do not have perfect accuracy, it is sensible that such improvement is smaller than using the perfect knowledge of developers.

4.4.1 Study Design

We obtain a list of issues of each project from their issue tracking systems[†]. Then, we extract the summary, description, and the release that the issue was implemented. For each issue, we applied an impact analysis to generate locations that are likely to be changed to complete the issue. In this study, we focus on IR-based impact analysis because it requires minimum information to perform, i.e., it takes only the change descriptions and source code as inputs of the technique. The IR-based impact analysis works by calculating the textual similarity between an issue and source code. We consider the similarity score determined by the impact analysis technique as a probability that a particular module will be modified. Then, we calculate the Context Relevance Index (CRI), which was proposed in our prior work [19]. The CRI of a module can be calculated by the summation of the similarity score in all issues that contains the module. The CRI represents the relevance of each module to the context of developers. A higher value of CRI means higher relevance to the context, i.e., more likely to be modified. We then create a new variable representing the CRI value.

In order to calculate the CRI value, we need to specify two parameters: the technique used by IR-based impact analysis and the cut point when calculating CRI. The technique used by IR-based impact analysis determines how the data is represented and how the similarity is calculated while the cut point determines the number of modules used when calculating CRI. In this study, we adopt four fundamental IR-based impact analysis techniques which are often studied in impact analysis research: Vector Space Model (VSM), Latent Semantic Indexing (LSI), Latent Dirichlet allocation (LDA), and Okapi BM25 (BM25). For the cut point, we use 10, 20, 30, and 40, which are usually used in the previous research [19], [21]. For each project, we try all combinations of each impact analysis technique and cut point, e.g., VSM with 10 cut point or VSM with 20 cut point. We then finally select the best combination of each project to use in this study. The underlying reason is that we want to simulate the real-world setting where parameters of the technique are optimized for each project before utilizing the technique. The best combination of each project can be found in Table 6.

Finally, similarly to RQ_3, we compare the accuracy of the prediction model between the baseline and the model with CRI value. Similarly to previous RQs, we conduct the Wilcoxon signed-rank test with the following null and alternative hypotheses to confirm the statistically significance.

[†]https://issues.apache.org/jira/secure/Dashboard.jspa

Table 7 Results of Wilcoxon signed-rank tests and the Cliff's delta effect size tests

Project	***p* value**	**Cliff's delta**
Accumulo	<0.001	0.293 (small)
Ambari	0.237	0.067 (negligible)
Derby	0.007	0.382 (medium)
Hive	0.002	0.218 (small)

H_{04} / H_{a4}: Developers' context estimated using IR-based impact analysis technique *does not improve* / *improves* the performance of prediction model.

In addition, we calculate Cliff's delta (d) as a measure of the magnitude of the improvement. The Cliff's delta is interpreted based on the threshold by Romano et al. [22]: *negligible* for $|d| < 0.147$, *small* for $0.147 \leq |d| < 0.33$, *medium* for $0.33 \leq |d| < 0.474$, and *large* for $0.474 \leq |d|$.

4.4.2 Results and Discussion

Figure 7 shows the accuracy of the prediction model between the baseline model, the context-aware model using IR-based impact analysis techniques to estimate developers' context, and the context-aware upper bound models. When comparing the accuracy between the baseline and IR-based models, we can observe that the IR-based model tends to have higher performance than the baseline model. Specifically, the median of the AUC values have been improved from 0.62 to 0.67 for Accumulo, from 0.66 to 0.67 for Ambari, from 0.58 to 0.65 for Derby, and from 0.62 to 0.66 for Hive.

Table 7 shows the results of Wilcoxon sign-rank tests and Cliff's delta effect size tests. The cells with p values less than 0.05 and Cliff's delta higher than 0.147 (not negligible) are highlighted in gray. The results of the Wilcoxon signed-rank test shows that the results are statistically significant at $\alpha = 0.05$ except for Ambari project. Therefore, for the projects other than Ambari, we can reject the null hypothesis and conclude that developers' context estimated by IR-based impact analysis technique can improve the performance of the prediction model. Furthermore, Cliff's delta values show that the result has a small effect for Accumulo and Hive, medium effect for Derby, and negligible effect for Ambari. So, we can see that the improvements are not negligible for all projects except for Ambari.

However, when we compare the improvement of the performance of the prediction model between IR-based and upper bound models, we can see that the improvements of IR-based models are much lower than the ones of the upper bound model. The result is as expected because the improvement of the upper bound model assumes perfect knowledge of developers, but the impact analysis technique relies on change descriptions to estimate the context. We can conclude that the IR-based impact analysis technique has a potential of representing developers' context, although using only textual change descriptions and source code as inputs.

In conclusion, developers' context estimated by IR-based impact analysis technique can improve the performance of the decaying module prediction model.

4.5 RQ$_6$: How can we further improve the performance of prediction model?

Motivation. As discussed in earlier sections, while the context estimated by existing IR-based impact analysis techniques can help improve the performance of a prediction model, the improvements are still low comparing to the situation of perfect knowledge of developers. One reason for such small improvement may be the low accuracy of the IR-based impact analysis, i.e., the high number of false positives and the low number of true positives. If the impact analysis technique results in many false positives, they will become noises that may obstruct the prediction model instead of helping them identify decaying modules. Furthermore, if the number of true positives is low, the impact analysis techniques can predict only a part of the correct answers and fail to detect the rest and, therefore, provide insufficient information to the prediction model. To this end, many approaches have been proposed to improve the accuracy of impact analysis techniques, such as combining an IR-based approach with extra information [23]. Nevertheless, even the state-of-the-art approach is still far from being perfect. Although the research community has been working on improving the accuracy of impact analysis techniques, it is still unclear whether high accuracy impact analysis techniques can improve the decaying module prediction model. Thus, to obtain empirical evidence, we artificially tune the accuracy of impact analysis techniques and observe the relationship between the accuracy of an impact analysis technique and a decaying module prediction model. We expect the result to be useful for the future direction of impact analysis research.

4.5.1 Study Design

We conducted an analysis under the assumption that mitigating the problems of high false positives and low true positives can improve the performance of the prediction model. We artificially modified the result of an IR-based impact analysis technique in two steps, which are inspired by the task input generation approach of a feature location study [24]. First, we decrease the number of false positives by randomly removing false positives from the result. Second, we increase the number of true positives by randomly adding false negatives to the result. We refer to the ratio that we decrease the number of false positives as False Positive Decrement Ratio (FPDR) and to the ratio that we increase the number of true positives as True Positive Increment Ratio (TPIR). Both of the ratios are from 0.0 to 1.0 with the step of 0.1. After performing the modification, we recalculate the CRI and use it as an exploratory variable of the prediction model in the same way as RQ$_5$. Finally, we calculate the performance of each model for comparison.

4.5.2 Results and Discussion

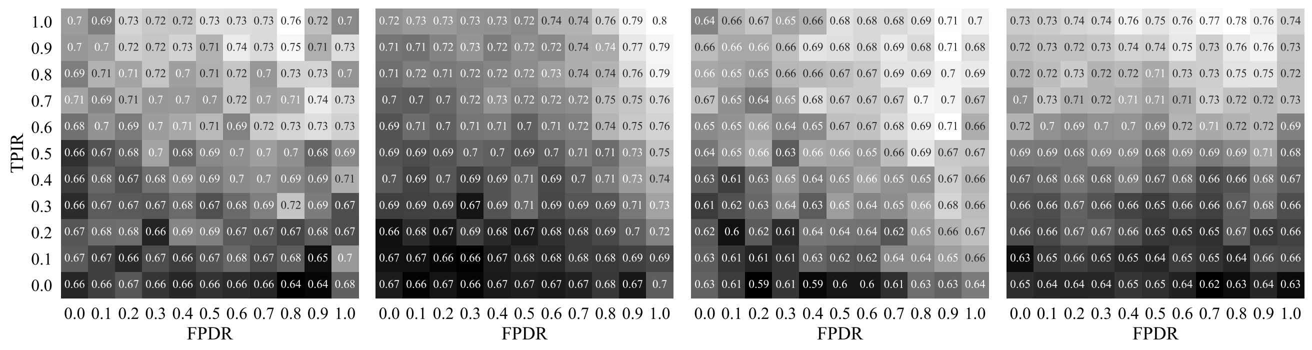
Figure 8 represents the heat map of the AUC of the prediction model. The vertical axis represents the values of TPIR, while the horizontal axis represents the values of FPDR. Each cell represents the AUC value of each setting. The brighter color shows a higher AUC, while the darker color shows the lower AUC values. In general, we can observe that AUC values tend to have a higher value in the top right corner of the heat map (e.g., in Ambari project). This result suggests that the more we decrease the number of false positives, and the more we increase the number of true positives, the higher AUC values become. Moreover, when we observe the value with low TPIR, we can see that increasing FPDR does not significantly improve the AUC values. This may indicate that increasing the number of true positives should be given higher priority than decreasing the number of false positives.

Technically, decreasing the number of false positives may be accomplished by complimenting an IR-based approach with other approaches such as a dynamic analysis approach. For example, we can use execution trace to filter irrelevant modules from the result of an IR-based approach, which may result in a lower number of false positives [21]. On the other hand, increasing the number of true positives can be done by combining the IR-based approach with a technique such as mining software repositories (MSR). For instance, MSR approach can adopt association mining rules to detect modules that were often modified together in the past and use that information to detect the modules that may not be found by only IR-based approach [21]. While combining both techniques together have been shown to improve the accuracy of impact analysis techniques [21], one downside is that it requires extra information which may or may not be available depending on the projects.

To sum up, whereas improving the accuracy of the impact analysis techniques by decreasing the number of false positives and increasing the number of true positives can improve decaying modules prediction, we should give higher priority to increasing the number of true positives.

5. Threats to Validity

In this study, we use four open source projects as our subjects. Therefore, the results of this study may not generalize to other types of projects. Additionally, we conducted the experiment on the prediction model using only the random forest method without performing any parameter optimization. Therefore, the result may differ in different models and different parameter settings. It is noteworthy that the primary goal of this study is not to find the highest performance of the prediction model but to show that existing techniques from a different research area can also be applied to this problem and that developers' context can improve the performance significantly. Moreover, replicating this study on a larger scale may be beneficial.



(a) Accumulo. (b) Ambari. (c) Derby. (d) Hive.

Fig. 8 Comparison of the prediction model performance at different TPIR and FPDR.

In addition, in this study, as we use the module's name as a primary key when conducting analyses, the case of renaming is not considered. This may impact the number of decaying modules in our study.

One significant threat to validity when using change descriptions to estimate developers' context lies in software repositories. For example, developers may make some modifications unrelated to any issue in the issue tracking system. In this situation, impact analysis techniques will fail to include such modifications in the estimated list. In addition, as we rely on commit messages to identify the true positives of each issue, if developers do not put the details of the issue to the commit messages, our technique will fail to identify the true positives. We mitigated this threat by filtering the projects that have a high ratio of commits that include issue ID (higher than 80%) based on the list by Miura et al. [25]. This can ensure that most of the changes were related to the issues in the issue tracking system.

6. Discussion

In this study, we propose the idea of decaying modules that can be used to target non-smelly modules to support proactive refactoring. However, several studies have shown that developers normally do not handle code smells [26], [27]. This phenomenon may be comparable to the fact that static analysis tools are not used well by developers owing to a large number of detected warnings [28]. In other words, the number of code smells is too high for developers to consider refactoring, thereby resulting in several studies focusing on filtering and prioritizing code smells [29]–[31]. Here the following simple question arises: Why should developers handle non-smelly modules when they already have more than enough smelly modules to handle.

In this context, we propose a guideline presented in Fig. 9 using quadrant analysis. The vertical axis represents the importance of the modules, i.e., smelly modules are more important than decaying ones. This is because smelly modules have a bad effect on the system [13], [14] while decaying modules are not yet found to have such an effect; therefore, they can be considered to be less important. The horizontal axis represents the urgency of the modules,

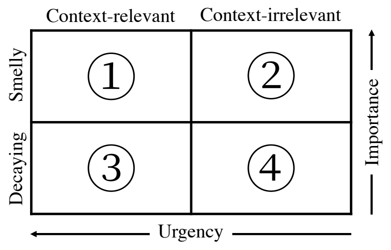


Fig. 9 Quadrant analysis of refactoring target prioritization.

i.e., context-relevant modules are more urgent than context-irrelevant ones. The main reason is that although solving code smells that are irrelevant to developers' context (e.g., modules developers plan to modify) may improve the overall quality of the system, it does not support the developers' current activities. This statement is also supported by our previous study on professional developers showing that developers tend to refactor the code smells related to their context [29]. Therefore, context-irrelevant modules can be postponed until they become relevant to developers' context.

Considering importance and urgency together, it is obvious that smelly context-relevant modules should have the highest priority, and decaying context-irrelevant modules should have the lowest priority. However, between decaying context-relevant and smelly context-irrelevant modules, we argue that decaying context-relevant modules should be given higher priority than smelly context-irrelevant modules. As aforementioned, solving context-irrelevant modules, irrespective of their quality, is not likely to facilitate planned implementation. On the contrary, in addition to preventing modules from being affected by code smells, solving decaying context-relevant modules can help developers get ready for their implementation.

7. Related Work

Murphy-Hill and Black used the terms *floss refactoring* and *root-canal refactoring* to refer to different refactoring tactics [10]. They use frequency and how developers mix refactoring with other kinds of program changes to categorize the two types of refactoring. Floss refactoring refers to frequent refactoring that is mixed with other types of program changes, whereas root-canal refactoring refers to infrequent refactoring that may not be mixed with other types of changes. On the contrary, in this study, we use the timing and purpose of refactoring to categorize the two types of refactoring. We term it reactive refactoring if the operation is applied after a code smell occurs in the source code with the purpose of removing the code smell and proactive refactoring if the operation is applied before a code smell occurs in the source code with the purpose of preventing a code smell. Therefore, floss refactoring and root-canal refactoring can be considered to be both reactive and proactive depending on when and why the developers perform the refactoring. In the case of root-canal refactoring, when developers reserve time specifically for refactoring, they can apply decaying module prediction at the same time when they detect code smells as targets for refactoring. In the case of floss refactoring, when developers make a change on a given module, the tool can warn the developers if the module is a decaying module.

One of the work aligned with the idea of proactive refactoring is *just-in-time refactoring* proposed by Pantiuchina et al. [17]. They proposed an approach to predict code components that will be affected by God and complex classes' code smells within a specific time. The approach allows developers to prevent code smells by refactoring source code right before they are introduced to the system. On the contrary, the idea of decaying module and module decay index (MDI) proposed in this study can not only be used to prevent the introduction of code smells but also to represent the status of the source code quality of non-smelly modules using the context of code smells.

The MDI proposed in this study can also be compared to *severity* [8] and *smell intensity index* [9] which were proposed to measure how bad a code smell is. Such metrics can be used to prioritize code smells, i.e., more severe code smells should be refactored first if developers want to improve the overall quality of systems. MDI can be used in a similar manner; however, for non-smelly modules. In other words, it can be used to measure the quality of non-smelly modules so that developers can notice the quality of the whole system, and not only smelly modules. Such information may allow developers to develop new strategies of refactoring by looking at the whole system and preventing modules from decaying rather than only reactively remove code smells from the system.

The similar term, *code decay*, is defined by Eick et al. in their work as "Code is decayed if it is more difficult to change than it should be" [32]. They use effort, interval, and quality as the keys to identify code decay. However, in this study, our definition is closely related to code smells, i.e., decaying modules are modules that are getting closer to becoming smelly. *Code decay indices* (*CDIs*) are also defined to quantify symptoms as risk factors. Although CDIs are mostly computable directly from a version control system, MDI defined in this study is computed from each module metrics value and the threshold of the symptoms of the code smell.

Moreover, in addition to representing the location that should be refactored, code smells are also used to describe characteristics of source code. For instance, Takahashi et al. successfully used code smell information to improve the accuracy of bug localization [33], [34]. Nevertheless, it is obvious that such a technique cannot be used on non-smelly modules as there is no such metric to indicate the quality of non-smelly modules. Thus, in addition to being used for supporting proactive refactoring, we expect MDI to be used to describe characteristics of source code that can be applied to other research fields as well.

8. Conclusion

In this study, we propose the idea of decaying modules that can be used to support proactive refactoring that can prevent code smells from occurring. A decaying module can be detected by measuring the module decay index (MDI). MDI can also function as a quality indicator of non-smelly modules. We conducted empirical studies on decaying modules and found that 19% of the number of modules that are modified in each release becomes decaying modules. Additionally, compared to non-decaying modules, decaying modules have a higher tendency to get decayed again in the future. Finally, we studied the use of a machine learning technique to predict decaying modules in the next release. We found that the use of developers' context can improve the performance of the prediction model.

Our future work includes the definition of MDI and decaying modules for other types of code smell. We also plan to conduct a study on the effect of decaying modules on the maintainability of the source code. In addition, using other time units for measuring decaying modules also remains as our future work.

Acknowledgments

This work was partly supported by JSPS Grants-in-Aid for Scientific Research Numbers JP18K11238 and JP21H04877.

References

[1] N. Sae-Lim, S. Hayashi, and M. Saeki, "Toward proactive refactoring: An exploratory study on decaying modules," Proc. third International Workshop on Refactoring (IWOR'19), pp.1–10, 2019.
[2] N. Sae-Lim, S. Hayashi, and M. Saeki, "Can automated impact analysis techniques help predict decaying modules?," Proc. 35th IEEE International Conference on Software Maintenance and Evolution (ICSME'19), pp.541–545, 2019.
[3] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[4] M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice, Springer, 2006.
[5] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?," Proc. 28th IEEE International Conference on Software Maintenance (ICSM'12), pp.306–315, 2012.
[6] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," Proc. 35th International Conference on Software Engineering (ICSE'13), pp.682–691, 2013.
[7] Z. Soh, A. Yamashita, F. Khomh, and Y.G. Guéhéneuc, "Do code

smells impact the effort of different maintenance programming activities?," Proc. 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16), pp.393–402, 2016.

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

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

[10] E. Murphy-Hill and A.P. Black, "Refactoring tools: Fitness for purpose," IEEE Softw., vol.25, no.5, 2008.

[11] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," IEEE Trans. Softw. Eng., vol.43, no.11, pp.1063–1088, 2017.

[12] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," J. Softw. Maint. Evol., vol.23, no.3, pp.179–202, 2011.

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

[14] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," Empir. Softw. Eng., vol.23, no.3, pp.1188–1221, 2018.

[15] D. Silva, N. Tsantalis, and M.T. Valente, "Why we refactor? Confessions of GitHub contributors," Proc. 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16), pp.858–870, 2016.

[16] V.R. Basili, L.C. Briand, and W.L. Melo, "A validation of object-oriented design metrics as quality indicators," IEEE Trans. Softw. Eng., vol.22, no.10, pp.751–761, 1996.

[17] J. Pantiuchina, G. Bavota, M. Tufano, and D. Poshyvanyk, "Towards just-in-time refactoring recommenders," Proc. 26th IEEE/ACM International Conference on Program Comprehension (ICPC'18), pp.312–315, 2018.

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

[19] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based approach to prioritize code smells for prefactoring," J. Softw. Evol. Proc., vol.30, no.6:e1886, pp.1–24, 2018.

[20] M. Kersten and G.C. Murphy, "Using task context to improve programmer productivity," Proc. 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06), pp.1–11, 2006.

[21] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," Proc. 34th International Conference on Software Engineering (ICSE'12), pp.430–440, 2012.

[22] J. Romano, J.D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys," Annual meeting of the Florida Association of Institutional Research, pp.1–33, 2006.

[23] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," J. Softw. Evol. Proc., vol.25, no.1, pp.53–95, 2013.

[24] T. Ishio, S. Hayashi, H. Kazato, and T. Oshima, "On the effectiveness of accuracy of automated feature location technique," Proc. 20th Working Conference on Reverse Engineering, pp.381–390, 2013.

[25] K. Miura, S. McIntosh, Y. Kamei, A.E. Hassan, and N. Ubayashi, "The impact of task granularity on co-evolution analyses," Proc. 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'16), pp.1–10, 2016.

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

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

[28] B. Johnson, Y. Song, E.R. Murphy-Hill, and R.W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," Proc. 35th International Conference on Software Engineering (ICSE'13), pp.672–681, 2013.

[29] N. Sae-Lim, S. Hayashi, and M. Saeki, "An investigative study on how developers filter and prioritize code smells," IEICE Trans. Inf. Syst., vol.101, no.7, pp.1733–1742, 2018.

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

[31] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," Softw. Qual. J., vol.23, no.2, pp.323–361, 2015.

[32] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," IEEE Trans. Softw. Eng., vol.27, no.1, pp.1–12, 2001.

[33] A. Takahashi, N. Sae-Lim, S. Hayashi, and M. Saeki, "A preliminary study on using code smells to improve bug localization," Proc. 26th IEEE/ACM International Conference on Program Comprehension (ICPC'18), pp.324–327, 2018.

[34] A. Takahashi, N. Sae-Lim, S. Hayashi, and M. Saeki, "An extensive study on smell-aware bug localization," J. Syst. Softw., vol.178, no.110986, pp.1–17, 2021.

Natthawute Sae-Lim received a B.E. degree in computer engineering from King Mongkut's Institute of Technology Ladkrabang in 2012. He received M.E. and Ph.D. degrees in computer science from Tokyo Institute of Technology in 2016 and 2019, respectively. He was a postdoctoral researcher in School of Computing at Tokyo Institute of Technology. His research interests include empirical software engineering and mining software repositories.

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

Motoshi Saeki received D.E. degree in computer science from Tokyo Institute of Technology, in 1983. He was a professor in School of Computing at Tokyo Institute of Technology. He is currently a professor in Department of Software Engineering at Nanzan University. His research interests include requirements engineering, software design methods, and computer supported cooperative work (CSCW).