Toward Proactive Refactoring: An Exploratory Study on Decaying Modules

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

***Abstract*—Source code quality is often measured using code smell, which is an indicator of design flaw or problem in the source code. Code smells can be detected using tools such as static analyzer that detects code smells based on source code metrics. Further, developers perform refactoring activities based on the result of such detection tools to improve source code quality. However, such 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 (e.g., low readability and understandability) 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.

***Index Terms*—code quality; code smell; refactoring;**

I. Introduction

Code smells were introduced by Fowler as an indicator of a design flaw or problem in the source code [1]. 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 [2]. Several studies have found that code smells are related to different aspects of software development such as maintainability [3], [4], [5]. 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 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 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. 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 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 becoming smelly. MDI is used to measure how bad a non-smelly module is, whereas *severity* [6] and *smell intensity index* [7] are used to measure how bad a smelly module is. The idea of decaying module 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 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.

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 characteristics of decaying modules.
3) We report an experiment on using a machine learning approach to predict decaying modules and show that developers' context can significantly improve the performance of the prediction model.

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 validity of this study. Section 6 discusses how decaying modules can be fit into refactoring prioritization. Section 7 presents the related works. Section 8 concludes this paper.

II. Decaying Module

A. Motivation

To describe the motivation behind the concept of decaying module, we use a dental metaphor because it has been used to explain the idea of software quality, e.g., floss and root-canal refactoring [8]. 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 the tooth decay in the future. Considering the aforementioned example, the concept of decaying module can be used to support proactive refactoring, i.e., developers can refactor decaying modules to prevent them from becoming smelly.

The main objective of decaying modules is to identify modules with a risk of becoming smelly in the future. This may be related with 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 [9]. 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.

B. 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, the strategy to detect God class defined by Lanza and Marinescu [2] can be reformulated as follows.

$$\text{God class} = (V_{\text{ATFD}} \geq T_{\text{ATFD}}) \land (V_{\text{WMC}} \geq T_{\text{WMC}}) \land (V_{\text{TCC}} \leq T_{\text{TCC}})$$

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.

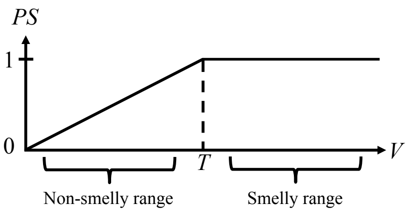


Fig. 1. Domain and range of percentage of symptom (PS).

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. Higher the ATFD, more likely 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. 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 becoming smelly.

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:

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

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, PS measured by ATFD of module x can be defined as

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

because the operator that determines its symptom is $\geq$. On the contrary, PS measured by the TCC of module x can be defined as

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

because the operator that determines its symptom is $\leq$.

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

2) Module Decay Index: Next, we define *MDI* as an indicator of how close a module is to becoming a code smell. A MDI is defined for each smell; MDI of smell s can be calculated by averaging the PS of each symptom measured by metric m where $M = \{\dots, m, \dots\}$ as:

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

For example, the MDI of God class of module x can be defined as

$$MDI_{\text{God class}}(x) = \frac{1}{3}\{PS_{\text{ATFD}}(x) + PS_{\text{WMC}}(x) + PS_{\text{TCC}}(x)\}.$$

Higher the MDI, closer is the module to becoming a code smell. MDI of one indicates that the module is a code smell.

MDI is also comparable to severity defined by Marinescu [6]. 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 severity of non-smelly modules.

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 decaying module when its current MDI has increased from the previous release.

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 [10]. 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 threshold such as data class or brain class defined by Lanza and Marinescu [2].

III. Decaying Module: Empirical Study

A. Motivation

This empirical study aims to conduct analyses on characteristics of a decaying module from different perspectives.

First, we empirically investigate the number of decaying modules that occur between releases. 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 be not 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.

Second, we study the characteristics of decaying modules, especially, their future characteristics. In this study, 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.

TABLE I
Dataset Information

Project	Period	No. of 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

B. Research Questions

The empirical study was conducted with the following research questions.

RQ1: How many decaying modules are present in each release compared to the modified modules?

RQ2: What are the future characteristics of the decaying modules compared to the non-decaying ones?

Details of each research question are explained later.

C. Experimental Setup

1) 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.

2) Data Collection: In this study, four open source projects: Accumulo[1], Ambari[2], Derby[3], and Hive[4] 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 I.

D. RQ1: How many decaying modules are present in each release compared to the modified modules?

1) Study Design: To answer this research question, we counted the number of modules that were modified and the number of decaying modules between each pair of releases. The modules that were modified between each pair of releases were generated using the `git log` command. 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.

2) Results and Discussion: Table II lists the result of our study. 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

[1]https://accumulo.apache.org/
[2]https://ambari.apache.org/
[3]https://db.apache.org/derby/
[4]https://hive.apache.org/

TABLE II
AVERAGES OF THE NUMBER OF DECAYING AND MODIFIED MODULES

Project	No. of modified classes	No. of decaying classes	Ratio
Accumulo	427.17	50.68	0.12
Ambari	394.34	98.47	0.25
Derby	450.78	72.48	0.16
Hive	627.94	132.17	0.21

TABLE III
RESULTS OF WILCOXON SIGNED-RANK TEST

Project	H_{01}	H_{02}	H_{03}
Accumulo	0.038	<0.001	<0.001
Ambari	0.130	<0.001	<0.001
Derby	0.571	<0.001	<0.001
Hive	0.046	<0.001	<0.001

and decaying classes. For example, in each release in the Accumulo project, 427.17 classes were modified 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, 0.25 for Ambari, 0.16 for Derby, and 0.21 for Hive. The average ratio in this study is approximately 0.19. In other words, compared to modified modules, 19% 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 19% of the number of modified modules were decaying modules in each release on average.

E. RQ2: What are the future characteristics of decaying modules compared to non-decaying ones?

1) Study Design: Similar to RQ1, 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). We excluded the releases without decaying modules, e.g., minor releases that have only few modifications, because we cannot compare the ratios of decaying and non-decaying modules.

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

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

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

Accordingly, the following alternative hypotheses corresponding to each null hypothesis can be defined as follows:

H_{a1}: The ratios of decaying modules that will be **modified** in later releases are *higher* than those of non-decaying modules.

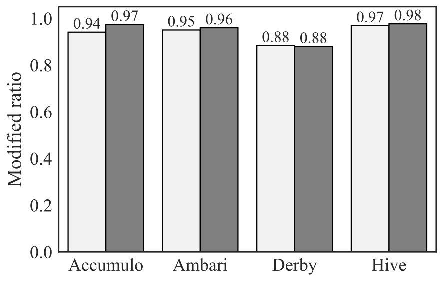
H_{a2}: The ratios of decaying modules that will get **decayed** in later releases are *higher* than those of non-decaying modules.

Then, we used the Wilcoxon signed-rank test, which is a non-parametric statistical hypothesis test, to determine any difference between the decaying and non-decaying modules.

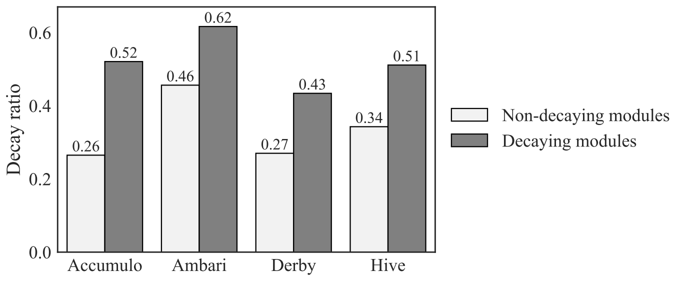
2) Results and Discussion: Figures 2a and 2b illustrate the results of our study. The bars represent the values of non-decaying and decaying modules. The values in Fig. 2a represent the ratio of modules that will be modified in later releases. We can observe only small, if any, differences in the projects. For example, the biggest difference is in the values of the Accumulo project indicating that approximately 94% and 97% of the non-decaying and decaying modules, respectively were modified in later releases. On the contrary, both the values of the Derby project are approximately 88% with slightly higher number of decaying-modules modified in the later release. Table III 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, while the p values of Accumulo and Hive are less than 0.05, the p values of Ambari and Derby are higher than 0.05. This means that for Ambari and Derby we fail to reject the null hypotheses (α = 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. For Accumulo and Hive, although we can reject the null hypotheses, the differences are very small.

Next, Fig. 2b illustrates the ratios of modules that decayed in later releases. For example, in the Ambari project, 46% of the non-decaying modules decayed in later releases. However, the number for decaying modules is as high as 62%. 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 decayed again in later releases. The results of the Wilcoxon signed-rank test listed in Table III indicate that the results are statistically significant (α = 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 is 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



(a) Modified ratio.



(b) Decay ratio.

Fig. 2. Comparison of averages of the number of decaying and non-decaying classes that will be modified and decay.

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 release was detected, decaying modules are more likely to get decayed in later releases.

IV. Decaying Class: Prediction

A. Motivation

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 improve. 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. Therefore, 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 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 [11]. Another example that is similar to this study is the work by Pantiuchina et al. [12] 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 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.

A scenario that is suitable to this case is the prefactoring phase, wherein developers refactor the source code to facilitate their future implementation [13]. 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 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 phase, wherein developers identify the code component that needs to be modified to satisfy change requirement [13]. We define this as *developers' context* [14]; 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" [15]. 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.

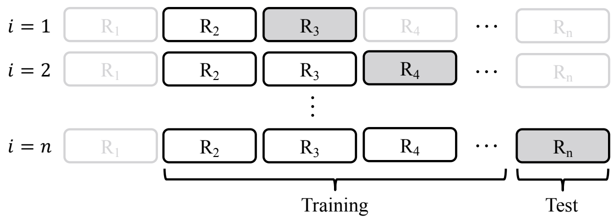


Fig. 3. Training and test data separation.

B. Research Question

In this section, we presented an empirical study with the following research questions.

RQ3: Can we use an existing technique to predict decaying modules?

RQ4: Does developers' context improve prediction performance?

Details of each research question will be discussed in the later subsections.

C. Experimental Setup

1) Baseline: We set a baseline model inspired by the work by Pantiuchina et al. that was proposed to predict modules that will become smelly [12]. 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.

2) 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 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*. 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 as response variables.

3) Data Separation: We separated data into training and test sets as shown in Fig. 3. 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.

4) Data Preparation: Next, we performed 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. Totally, we removed 16, 19, 15, and 19 variables for the projects: Accumulo, Ambari, Derby, and Hive, respectively.

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. For each training set, we randomly selected non-decaying modules equal to the number of decaying modules to balance the classes of response variables.

5) Model Construction: Finally, we constructed prediction models using the random forest method of the scikit-learn library [16] and calculated 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.

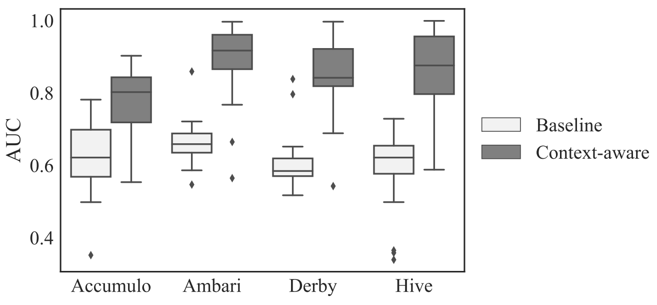


Fig. 4. AUC values of baseline and context-aware models.

6) Data Analysis: In this study, we use 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.

D. RQ3: Can we use an existing approach to predict decaying modules?

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.

2) Results and Discussion: Figure 4 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.

E. RQ4: Does developers' context improve prediction performance?

1) Study Design: We compare the performance of two models: baseline and context-aware. 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. 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. It is noteworthy that this way of representing developers' context may not be practical as it assumes perfect knowledge of developers. However, our main goal of this preliminary study 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

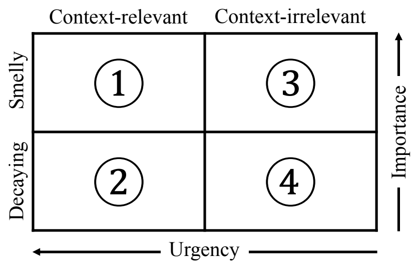


Fig. 5. Quadrant analysis of refactoring target prioritization.

of using developers' context. Then, we plan to conduct a study with more practical settings, e.g., using automated approaches such as impact analysis to estimate developers' context, in the future.

To confirm if the results are statistically significant, we conduct the Wilcoxon signed-rank test with the following null hypothesis:

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

Therefore, an alternative hypothesis can be defined as:

H_{a3}: Developers' context *improves* the performance of prediction model.

2) Results and Discussion: Figure 4 shows the result of our experiment. The box plot shows the values of the performance of the baseline and context-aware 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 Accumulo, Ambari, Derby, and Hive, 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 III. 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.

V. 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 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.

VI. Discussion

In this study, we propose the idea of decaying module 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 [17], [18]. This phenomenon may be comparable to the fact that static analysis tools are not used well by developers owing to the large number of detected warnings [19]. 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 [20], [21], [22]. 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. 5 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 bad effect on the system while decaying modules are yet to have such an effect; therefore, they can be considered to be less important. The horizontal axis represents the urgency of the modules, 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 [20]. 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.

VII. Related Work

Murphy-Hill and Black used the terms *floss refactoring* and *root-canal refactoring* to refer to different refactoring tactics [8]. 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.

One of the work aligned with the idea of proactive refactoring is *just-in-time refactoring* proposed by Pantiuchina et al. [12]. 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* [6] and *smell intensity index* [7] 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" [23]. 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.

VIII. Conclusion

In this study, we propose the idea of decaying module 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 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 use of developers' context can improve the performance of the prediction model.

Acknowledgments

This work was partly supported by JSPS Grants-in-Aid for Scientific Research Numbers JP18K11238, JP15K15970, JP15H02683, and JP15H02685.

References

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

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

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

[4] ——, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proc. ICSE*, 2013, pp. 682–691.

[5] Z. Soh, A. Yamashita, F. Khomh, and Y.-G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?" in *Proc. SANER*, vol. 1, 2016, pp. 393–402.

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] J. Pantiuchina, G. Bavota, M. Tufano, and D. Poshyvanyk, "Towards just-in-time refactoring recommenders," in *Proc. ICPC*, 2018, pp. 312–315.

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

[14] 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.

[15] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proc. FSE*, 2006, pp. 1–11.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, no. Oct, pp. 2825–2830, 2011.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.