ILKOM Jurnal Ilmiah Vol. 16, No. 2, August 2024, pp. 145-150
Accredited 2nd by RISTEKBRIN No. 200/M/KPT/2020; E-ISSN 2548-7779 | P-ISSN 2087-1716

Research Article Open Access (CC–BY-SA)

Improving Source Code Quality by Minimizing Refactoring Effort

Hayatou Oumarou[a,1,*], **Kabirrou Hamadou Tizi**[b,2]
[a]*Department of Computer Science, University of Maroua, Maroua, Cameroon.*
[b]*Department of Mathematics and Computer science, University of Maroua, Cameroon.*
[1] *oumarou.hayatou@univ-maroua.cm;* [2] *kabirroutizi@gmail.com*
* *Corresponding author*

Article history: Received October 21, 2023; Revised March 18, 2024; Accepted June 25, 2024; Available online August 20, 2024

Abstract
Software maintenance is a time-consuming and costly endeavor. As a part of maintenance, refactoring is aimed at enhancing quality. Due to project deadlines and limited resources, developers need to prioritize refactoring activities. In this paper, we present a livestock management-inspired approach for identifying and prioritizing classes to refactor within an object-oriented program. This approach empowers developers to enhance the time/quality ratio. The novelty of our approach lies in utilizing established metrics for detecting code defects to prioritize each class. To validate its effectiveness, the approach was tested on four distinct Pharo-based open source programs. The results demonstrate the approach's efficacy in improving software quality, reducing development time, and enhancing team productivity.

Keywords: Evaluation Model; Maintainability; Metric; Refactoring; Software Quality.

Introduction

In [1], Lehman asserts that a program used in a real-world setting must undergo continual change to remain effective and relevant. Consequently, software requires continuous maintenance to remain functional, with maintenance consuming up to 90% of the total cost of a typical software project [2]. Refactoring is a significant contributor to maintenance costs, as it allows developers to enhance the internal design structure of software while preserving its external behavior. According to Fontana [3], refactoring is crucial for reducing maintenance costs. Martin Fowler states in [4] that refactoring can be time-consuming and costly. On the other hand, Walkinshaw [5] shown that 80% of defects are found in only 20% of the code, much of which is rarely executed. This highlights the need to prioritize the maintenance of specific parts of the code, particularly in the refactoring process. The challenge lies in prioritizing these critical areas while minimizing refactoring efforts. The primary benefit for developers is a reduction in their workload, time savings, and the freeing up of resources.

Refactoring is a widely-recognized method of improving the internal structure of software while preserving its external behavior [6], [7]. It essentially aims to improve the (structural) quality and non-functional characteristics of a program by modifying its source code without altering its behavior [4]. It is a very interesting alternative for improving the quality of a given software [8].

More specifically, refactoring aims to reduce code smells. Vidal et al [9] proposed the SpIRIT tool for identifying and prioritizing bad smells on the basis of three parameters: code smell relevance, component modification history and modifiability scenarios. These code smells are then prioritized to help developers refactor the highest priority smells with the least effort. Yang et al. [10] use the CDT code clone detection tool to detect duplicate code from source code written in C language. They use machine learning algorithms to classify detected code clones according to history. Fontana and Zanoni [11] used machine learning algorithms to classify code smells according to their severity. Amandeep et al. [12] proposed a bio-inspired approach called DT-SOA for prioritizing five code smells on the basis of three parameters such as code smell relevance. All these techniques work on code smell instances, whereas we propose an approach that will aggregate these data at class level.

The first attempt at class prioritization was presented by Zhao and Hayes [13]. They prioritized classes according to a weighted maintainability ranking for each class containing bad odors, using different class characteristics such as size, complexity, etc. They considered several measures such as the size of the class, the complexity of the class, the size of the class and the complexity of the class. They generate a weighted ranking for classes or system packages that require urgent refactoring. Zhao and Hayes assume that the main task in identifying opportunities for refactoring is simply to locate code fragments that require more maintenance effort than others. This maintenance effort can be determined by computing the size and complexity of the source code. Khosla [14] proposed an approach for prioritizing classes requiring urgent refactoring. They introduced the Quality Depreciation Index (QDIR). The measure for each class is expressed as a function of the number of code bad smells contained in the class and the value of the object-oriented (OO) metric of the class itself. Classes are further organized according to their QDIR values to identify severely affected classes requiring immediate refactoring. Steidl and Eder [15] have worked on prioritizing classes for refactoring. Their approach is to prioritize code clones and long methods. They detect these two defects, then identify refactoring opportunities to eliminate them. Abha Chouchary and Singh [6] have proposed a three-stage approach. The first step is to analyze the history of refactoring performed on previous versions of the software. Then analyze the architectural relevance of the classes in the current version of the software. Finally, generate a ranking based on the intensity and frequency of code smells present in the selected classes. They apply the first two steps in parallel, the first working on all versions of the application and the second on the current version only. It generates a ranking based on the intensity and frequency of code smells present in the selected classes. Kosker et al. [16] proposed a code complexity-based class prioritization approach based on a weighted naive bayes algorithm to predict classes requiring urgent refactoring. Kosker et al. [16] analyze historical class information with the hypothesis that if class complexity decreases from the start of the project, then these classes are treated as refactored classes [4]. Rani and Chhabra [17] used a two-phase approach to prioritize malodorous classes. First, they used the jDeodorant tool to identify five code smells "Shotgun Surgery", "Parallel Inheritance", "Feature Envy" "Divergent Change" and "Blob" in Java EE software. In addition, they used change history as a fitness function to prioritize classes. Tarwani and Chung [18] used quality as a fitness function to prioritize malodorous classes.

In this paper, we introduce a method inspired by livestock park management to prioritize candidate classes for refactoring in a manner that minimizes the required effort. The rest of the document is organized as follows: Section 2 presents the proposed approach. Section 3, "Results and Analysis," scrutinizes and interprets the results obtained during the implementation of the method and threats to validity, while Section 4 presents conclusions and outlines potential future work.

Method

Inspired by the management of a local livestock park, we adapt its principles to software engineering. Here's how it works: similar to animals in a stockyard, some components of the software have issues that require attention. Often, there are limited resources available to address all the issues at once. Therefore, the components requiring attention are prioritized based on the type of issue and their history. Our analysis of this situation reveals four (4) important steps:

1) *The first step is to analyze each animal's health history;*
2) *The second step is to assess the current health of each animal;*
3) *The third determines the overall situation of the park;*
4) *The fourth step consists in prioritizing the animals according to the information obtained in the previous steps.*

We have applied this analogy to the software system, where classes are likened to animals and the software itself to a livestock park. Therefore, in order to prioritize the classes requiring immediate refactoring, we have established the following steps:

1) *The first step is to carry out a historical analysis of the system (Hc).*
2) *Considering the class as a unit, the second step is to detect faults in each class of the system.*
3) *The third step consists in determining the state of the system through the relationships between the various entities;*
4) *The fourth consists of prioritizing classes according to its score (Oc).*

Figure 1 shows the various stages in the approach including historical analysis; quality analyzer; system analyzer; and prioritizer.

We operate under the assumption that a class which undergoes frequent modifications is more likely to require future modifications. To identify classes that are more prone to future modifications, we analyze their historical data. Each class is then associated with an Hc score, defined as in Equation 1.

$$Hc = \sum_{i=1}^{n} a_i \tag{1}$$

Where a_i = 1 if the class has been modified, else a_i = 0 and i = version/commit

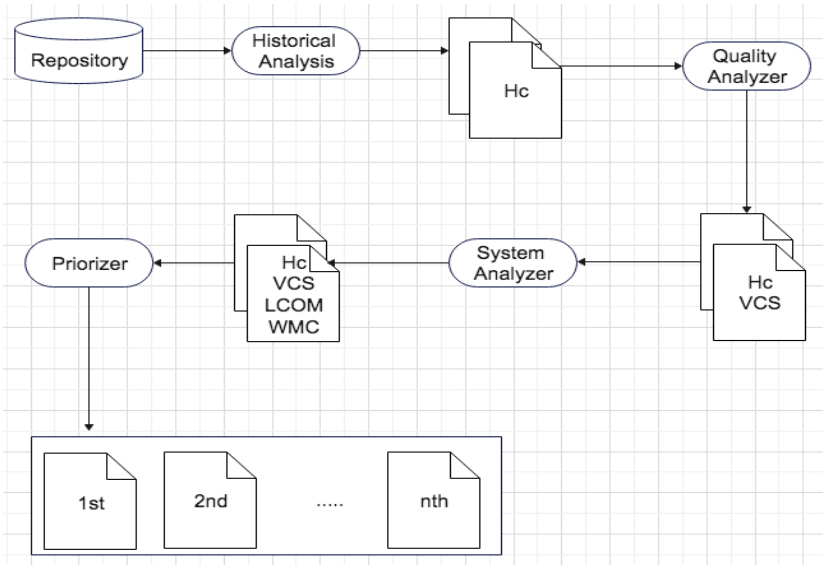

Figure 1. Illustration of the different stages of the method

The second step consist of class quality evaluation. This assessment heavily relies on the identification of code smells, which indicate the presence of issues. It's important to note that not all detected code smells carry the same weight. For instance, an "undeclared variable" does not pose the same severity as a "variable with a long name". In our evaluation, we will utilize the weights assigned by the detection tools. The class's code smell value (VCS) is then determined by Equation 2.

$$VCS = \sum_{i \in M} \lambda_i b_i \tag{2}$$

Where λ_i is the weight of a smells, b_i the Smells and M is the set of code smells.

In the step of system analyzer, we evaluate the status of the system using established metrics. The software engineering community has developed its own methods for characterizing software quality and it's evaluation [19]. In the context of object-oriented software, various metric suites have been introduced to assess its quality, including the Chidamber and Kemerer metric suite [20], the MOOD [21], the maintenance index [22] and others. Therefore, Object-Oriented software metrics aim to offer a quantitative perspective on how these principles have been implemented to attain specific objectives. For our project, we specifically utilize the WMC and LCOM metrics.

WMC is defined as in Equation 3.

$$WMC = \sum_{k=1}^{m} C_k \tag{3}$$

Where C_k represents the complexity of each method the class

WMC [23] is a measure of the complexity of a class. The higher the number, the more likely it is to be complexity-specific, limiting the possibility of reuse. In software development practice, it is recommended that WMC be kept low.

"Lack of Cohesion Metric" (LCOM) [24] tells us how cohesive a class is. But for our purposes, we use the Equation 4:

$$LCOM = 1 - \frac{\sum MF}{M \times F} \tag{4}$$

Where,

M is the number of methods in the class (includes static and instance methods, constructors, properties...). F is the number of instance fields. MF is the number of class methods calling a given field. $\Sigma(MF)$ is therefore the sum total of the value obtained by MF on all the fields in the class.

In the last step, we generate a rank to prioritize the class. The score (O_c) used to prioritize classes is determined by Equation 5.

$$O_c = 1 - \frac{e^{\left(WMC + Hc + \frac{1}{VCS}\right)}}{LCOM + e^{\left(\frac{1}{VCS}\right)}} \qquad (5)$$

Where Hc, VCS, WMC and $LCOM$ are obtained in previous steps.

After generating the scores for all classes, we proceed to sort these classes in descending order based on their Oc value. It's important to note that the closer the score is to 1, the higher the priority of the class.

Results and Discussion

A. Dataset and experimental SetUp

The context of the case study is Pharo, a real system for which we have access to software repositories, contributors and code analysis tools. The dataset comprises 4 open source Pharo systems available on GitHub at https://github.com/pharo-project. We considered these projects as ideal for our research due to Pharo's open source and immersive nature. Additionally, we have access to the developers and the analysis tool. Table 1 outlines the total number of entities in the dataset (packages, classes), which will greatly facilitate the analysis. For our analysis, we opted for Moose [25] as it offers a range of tools for metrics, clustering, querying, visualization, and interactive navigation within the source code. It is a specialized, extensible analysis tool designed for in-depth source code analysis and incorporates most of the fundamental elements. Our task will involve integrating these elements together. Moose is open source and immersive, allowing meta-programming and easy extensibility.

Table 1. Entities in evaluation dataset

	#versions	#classes in the last versions
Collection-sequenceable	261	13
Collection-Streams	194	10
Monticello	1164	92
Athens-Cairo	147	39

For this experiment, we consider the code smells in Table 2. This choice is guided by the large presence of these smells in the source code, their easy detection and easily appreciable correction.

Table 2. Description of Code Smells

Code smells	Description
God class	These code smells affect classes that do not follow the single responsibility principle.
Complex class	Instances of this smell affect classes with high cyclomatic complexity
Shotgun surgery	When a class makes a change (field or method) and triggers numerous small changes in other classes.
Spaghetti code	These smells affect classes that don't use object-oriented programming principles (e.g. inheritance), declare at least one long method without parameters and use instance variables.

B. Results presentations and observations

For the experiment, we selected four pairs of developers, each possessing extensive knowledge of the projects. Each expert within the pair had actively contributed to the maintenance of the projects, confirmed through commit log history. Each pair worked on 2 packages. The pairs dedicated 3 hours to refactoring one package without our approach and another package with our approach suggestions for 3 hours. Thus, for each package, we had 2 pairs: one utilizing our approach and one without. The group using our approach is referred to as the "control group." In order to assess the enhancement in quality, we defined the following metric:

$$Q1 = 1 - \frac{\sum O'c}{\sum Oc} \qquad (5)$$

Where Oc : the class score before modifications and
$O'c$: the class scores after modifications

We then run the QI to measure improvement on each package for each pair. Table 3 presents the results. We discuss results with experts to collect their appreciations.

Table 3. Improvement comparison.

Programs	Control Group (QI)	Other Group (QI)

Monticello	0.12004204	0.070212
Athens-Cairo	0.21951088	0.0794154
Collection-sequenceable	0.11617547	0.0861431
Collection-Streams	0.2665001	0.1165001

Observation 1: On average, we notice that over 46% of all classes are frequently refactored in previous releases. Generally, the percentage of refactored classes demonstrates a declining trend as we progress through the commits. This could be attributed to either the developers' waning interest in refactoring or the improving quality of the software product. As they face constraints such as strict deadlines and a limited budget, they allocate decreasing effort to applying refactoring techniques.

Observation 2: Through rigorous and analytical observation, we have determined that classes which have undergone multiple modifications exhibit a high Oc. For instance, in the Monticello program, the MCVersionMerger class has a higher score compared to other classes, indicating an urgent need for refactoring. This class has a significantly large number of historical changes.

Observation 3: We have observed that the control groups yield the best results. From our discussion, it is evident that our approach is effective and contributes to saving both quality and time.

The results demonstrate a notable enhancement when employing the approach. For instance, in the cases of Athens-Cairo, Monticello, Collection-Sequenceable, and Collection-Stream, we observed improvements of approximately 14%, 5%, 4%, and 15% respectively. The average percentage improvement amounts to 9.5%. There is a substantial advantage in using our approach compared to "blind" refactoring.

As with any empirical evaluation, the results of our experiments are subject to threats to validity. We have identified the following notable threats:

1. Pharo may not fully represent a larger population of systems, and the same holds true for applications from diverse domains or those written in different programming languages. Mitigating this challenge is difficult as there is limited information about the system property crucial for ensuring representativeness. Replicating the experiment for other systems is necessary. Nevertheless, we are confident that our experiment is not subject to this limitation.
2. The tools we have chosen may introduce bias. We also believe that these tools are credible, as they have been used by large companies for over 20 years in real, non-trivial projects.
3. The metrics used for evaluation may be inappropriate. We have used the metrics defined and applied in other work. We believe that these metrics are well established and do not pose a threat to validity.
4. Internal validity threats are linked to the implementation of our approach, as errors may inadvertently impact the accuracy of our results. To mitigate this threat, we conducted a manual examination of a subset of the results and did not identify any obvious errors. Additionally, biases in the developers' working methods can emerge. To minimize this risk, we opted for packages with multiple contributors, aiming to ensure that the developers represent a diverse population within the Pharo community.
5. The approach does not consider the needs of stakeholders. In discussions with experts, some have raised this issue. We don't think this threat affects our approach because it is focused on internal quality.

Conclusion

This paper presents a new method for prioritizing classes for refactoring based on their quality score. This score considers historical analysis, current source code quality, code complexity, and various metrics while also distinguishing between types of code smell. The study has shown that this prioritization leads to improved time efficiency for quality improvement. The empirical evaluation confirms the potential and practicality of the approach. The results of this research provide valuable insights for researchers and practitioners, enabling them to improve product quality and development process efficiency while reducing effort. In the short term, we plan to test our approach on multiple systems. In the medium term, we aim to extend the consideration of code smells in order to generalize our results.

References

[1] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry and W. M. Turski, "Metrics and laws of software evolution-the nineties," *International Software Metrics Symposium,* vol. 4, 1997.

[2] S. S. Ali, M. S. Zafar And M. T. Saeed, "Effort estimation problems in software maintenance–a survey," 2020.

[3] Z. R. e. A. Fontana, Software Clone Detection and Refactoring, University of Milano-Bicoco, Italy, 2019.

[4] K. e. a. Martin Fowler, Refactoring: Improving the Design of Existing code, Wesley, 1999.

[5] N. Walkinshaw And L. Minku, "Are 20% of files responsible for 80% of defects?," *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.,* vol. I, no. 12, pp. 1-10., 2018.

[6] A. ChaouldHary, Effective Prioritization of Classes for Reduced Refactoring Effort ", 31e conférence internationale IEEE/ACM sur le génie logiciel automatisé, Communiqué, 2018.

[7] G. Lacerda, F. Petrillo, M. Pimenta And G. Yann-Gaël, "Code smells and refactoring: A tertiary systematic review of challenges and observations.," vol. 167, 2020.

[8] o. Hachemane, évaluation de l'impact du refactoring basé sur les clones sur la qualité (maintenabillté-testabillté) des systèmes orientés objet, 2018.

[9] M. D.-P. Vidal, approach to prioritize code smells for refactoring, 2020.

[10] H. H. I. K. Yang, Filtering Clones for Individual User Based on Machine Learning Analysis, 2019.

[11] Z. Fontana, Code smell severity classification using machine learning techniques, 2019.

[12] A. Kaur, S. Jain And S. Goel, "Sandpiper optimization algorithm: a novel approach for solving real-life engineering problems.," *Applied Intelligence,* vol. 50, pp. 582-619, 2020.

[13] e. H. Zhao, Predicting classes in need of refactoring: an application of static metrics, 2006.

[14] Khosla, Priorisation des classes pour le refactoring : une étape vers l'amélioration de la qualité du logiciel, 2020.

[15] E. Steidl, Prioritizing maintainability defects based on refactoring recommendations, in: in Proceedings of the 22nd International Conference on Program Comprehension, 2018.

[16] T. e. B. Kosker, An expert system for determining candidate software classes for refactoring, 2019.

[17] J. K. C. Rani, Prioritization of smelly classes: A two phase approach (reducing refactoring efforts), 2017.

[18] A. C. Tarwani, Prioritization of code restructuring for severely affected classes under release time constraints, 2018.

[19] H. Oumarou, D. K. Moulla and Kolyang, "Using Quality Measures During the Software Development Process: Case Study of Cameroonian Software Industry.," *Indonesian Journal of Computer Science,* vol. 12, no. 3, 2023.

[20] S. R. C. a. C. F. Kemerer, A Metrics Suite for Object, IEEE Transaction on Software Engineering, 2020.

[21] C. a. N. Harrison, Evaluation of the MOOD Set of Object-Oriented Software Metrics, IEEE Transaction on Software, 2020.

[22] O. Hayatou, "A Source-Code Maintainability Evaluation Model for Software Products.," vol. 3, no. 2, 2023.

[23] R. Jangra, O. P. Sangwan and D. Nandal, "A Novel Approach for Software Effort Estimation using Optimized C&K Metrics," 2022.

[24] P. Mengal, "Métriques Et Critères D'évaluation De La Qualité Du Code Source D'un Logiciel," Revue d'un professionnel de l'industrie du logicie, Paris, 2019.

[25] N. Anquetil, A. Etien, M. Houekpetodji, B. Verhaeghe, S. Ducasse, T. Clotilde, D. Fatiha, S. Jerôme and D. Mustapha, "Modular Moose: a new generation of software reverse engineering platform.," *Reuse in Emerging Software Engineering Practices,* no. 19, 2020.