Reducing Maintenance Efforts of Developers by Prioritizing Different Code Smells

Abstract: An architectural problem associated with a software system constantly affects the evolving system. These architectural problems are symptoms of different code smells that must be removed using refactoring. The refactoring efforts are directly associated with the maintenance cost of the software system. This cost can be minimized or optimized by prioritizing different code smells. Prioritization helps tackle only a subset of code smells and hence minimize the maintenance cost. This makes code smells ranking and prioritization an important research area and is tackled in this paper. This paper proposes a new approach that is capable of ranking the existing code smells. This prioritization is based on the newly proposed metric based on three identified key criteria. Firstly, the severity of a code smells based on the change-history of a software system. Secondly, the association of the code smells with the improvement in the understandability of the software system. Thirdly, the importance of a developer’s feedback for a given code smells associated with a class in the software system. The feasibility of the proposed approach is tested and evaluated on an open-source Java software system.


I. INTRODUCTION

In software engineering, software evolution is a continuous and long-lasting activity associated with the software development life cycle. Generally, a big amount of cost (upto 90%) is only associated with only maintenance and evolution of a large and complex software system [2]. This maintenance and evolution are associated with bug fixing and requirement change. These activities of the developers may create architectural problems introduced in the system. This poor underlying architecture may be a symptom of the problem associated with the source code of the system. These symptoms are identified as the code smells and are the main reason for the poor code quality [4]. Code-smells, also known as the code that is bad for the developer and which affects the maintenance of the software in future when there is a need to change the software [8]. One quick fix technique to overcome these code smells is the refactoring. A refactoring activity helps to improve the underlying software architecture and is directly associated with the code smell type.

The authors in Fowler et al. identified and classified different code problems into 22 code smell types [4]. However, the authors in [15] are of the opinion that not all these code smells are equally important. This is because these code smells are not directly linked with the code quality. In a given situation, a particular code smell may affect the code quality more as compared to other code smells present in the system. Therefore, it is important to prioritize these identified code smells and only remove those code smells (by applying suitable refactoring) that affects more to the improvement of underlying system architecture. This motivates us to propose a new technique for code smell prioritization. This paper does not focus on identifying new smell in a system, rather, we aim to rank already present code smells in the system.

Many approaches are already proposed that focuses on code smell filtering and prioritization [12], [3], [14], [1], [16]. These approaches are based on textual-based, smell intensity based on source code metrics, architecture, and context-based properties of the software system. However, all these techniques missed out two important factors, such as the developer’s knowledge and the impact of change-history of the underlying software system, while prioritizing different code smells. This becomes the research gap for our paper.

This paper proposes an approach for ranking/prioritization of different code smells associated with the software system. The code smells prioritization helps obtain the optimal balance between the applied maintenance efforts and the improvement in code quality. The proposed approach is based on three criteria namely severity of a code smell, understandability, and the user importance for a particular code smell. Here, the severity of a code smell is measured based on the available change-history of the software system and it measures the past importance of a code smell. A code smell that is tackled more in past is generally more severe and must be presently handled. This measurement in our proposed approach is done using a newly proposed metric known as Severity based on change-history of a software system. The understandability criteria are also considered in our proposed approach and any code smell whose mitigation increases the understandability of the underlying code is considered more important. The understandability is measured using the Cognitive Complexity metric. Finally, the user importance is considered based on developer’s knowledge about a particular code smell. This paper ultimately proposes a new prioritization metric known as Rank that assign a rank value to a given code smell.

The rest of this paper is structured in the following sections: first, different relevant literature work is summarized. Next, the proposed approach is discussed in detail. Further, the experimentation and results obtained are provided. Finally, conclusion and future work are presented.

II. LITERATURE SURVEY

This section in the research paper gives information about the work already carried out in the...
field of Prioritizing different code smells. The study of code smells prioritization strategies is always remains a subject of recent studies in the literature. Code smells being a dynamic concept for researchers throughout the globe and thus different verifiable studies have been done to get perception into the matter.

Code smell is also prioritized by severity which describes that code smells is very strong and it is used as one of the important aspect. [6] defined the term severities which can be computed by the value which is chosen from the given metric larger than the given threshold and also the number of times the value exceeds. The authors in [5] finds that the source code which contains the bad smells are change very frequently and also becomes very defective. Fontana et al. proposed Intensity index which is calculated by the distribution of metrics of the software[3]. They defined and also represented the range which goes from Very low to Very high with five levels of intensity and it can be represented by the value from 1-10 as the index value. Similarly, Yamashita and Moonen proposed the factors of the code developers to prioritize the bad smells in the code that in the future affects the maintenance part of the software but partly[17]. Context Relevance Index also defined by them by to prioritize the code smells and also list the problems as an issues lists by using a tracking system.

Besides these, user perception regarding dealing with a code smell is also studied. According to Arcoverde et al. different code smells can be prioritized based on four factors namely error density, software architecture, error density and role of evolutionary information of a software system [1]. The authors in [11] studied how different developers understand different code smells. They found that the way with which the developer handles code smell is different for different type of code smells. Similarly, the authors in [16] studied the code smell at the component level. Based on the study, they proposes a semi-automated technique to prioritize the code smells. Their approach is based on the three factors namely change history of the component, modifiability scenario of the component, and the user consideration of the code smell.

### III. PROPOSED APPROACH

In this section of our paper, we describe our proposed approach in detail. The proposed approach aims at prioritizing different code smells present in a system by ranking them. The ranking of different code smells is based on three criteria namely 1) severity of the particular code smell, 2) needed maintenance efforts, and 3) User feedback for the particular code smell. The severity of a code smell is determined based on the change-history of the underlying software system. The maintenance efforts are measured in terms of the smelly class code’s cognitive complexity. Similarly, the user feedback is received in order to receive the user’s perception who is involved in the maintenance team of the software system. Based on these three mentioned criteria, finally, different identified code smells are prioritized. Figure-1 gives details of the proposed approach of this paper.

#### A. Code Smells Identification

Before prioritizing different code smells, it is important that we have a list of actual code smells present in the system. As our paper does not target identifying code smells, so, we rely on standard tools for identifying different kinds of code smells present in a software system. Different tools used in this paper for this purpose includes Deodorant\(^1\), InFusion\(^2\), and JSpIRIT\(^3\). These tools are available as eclipse-plugins and are capable of analyzing object-oriented Java programs. These software return list of possible code smells present in the system. Here, multiple tools are selected because no single tool is capable of detecting all the possible code smells present in a software system. This is based on our manual inspection of these tools when they are applied on a sample software system. Two or more tools identify code smells present in the underlying software system, so in order to prepare a final list; we simply merged all the lists together to have a common overview of the code smells belonging to the system based on the expert (developers) judgment.

#### B. Code Smells Prioritization

This phase of the proposed approach aims at ranking/prioritization different identified code smells such that the overall maintenance efforts are minimized. The proposed prioritization is based on the above mentioned (in the first paragraph of this section–3) three criteria. These different criteria and overall function used to rank different code smells are discussed in detail in the following sub-sections of this paper.

#### C. Change-History based Severity

The change-history of a software system depicts how the software adapts to the bugs identified and changed requirements in the past. It depicts details of changed classes in order to remove bugs and/or code smells in the past. Moreover, the authors in [9] conclude that not every code smell is equally relevant to the maintenance efforts associated with software. For example, some big classes (possible god class code smell) are equally considered important in software like compilers and they are negatively associated with the maintenance of the system. Therefore, we conclude that different code smells have different severity and hence, developers devote different maintenance efforts. The change-history of a software system is a direct indicator of the maintenance efforts devoted by developers with a class in past based on two criteria namely bug fixing and requirement change.

---

1. https://marketplace.eclipse.org/content/paedorant
3. https://sites.google.com/site/santiaaguaral/project/jsprite

**Retrieval Number:** H10380688S319/190BEIESP
It means a smelly class that is changed more in past presently needs more developer’s efforts in order to reduce its future maintenance efforts. These extra efforts are measured as the severity of a code smell associated with the class using the newly proposed metric as shown in the equation (1).

$$\text{Severity } (C_j) = \frac{T_C}{T_N}$$

Here, \(T_N\) is the total number of transactions present in the change-history of a software system. The \(T_C\) is the total number of transactions in which the class \(C\) associated with the code smell \(I\) undergo changes. It is important to note here that a higher severity valued class’s code smell is given more importance and hence undergo more change as compared to other classes. Before applying this proposed metric, the change-history of the system is pre-processed in order to effectively utilize them for severity measuring. These steps are as follows:

1. The change-history contains a lot of change information associated with the software overtime. However, not all the change information is directly relevant from the code smell point of view. Hence, we consider only those changes that are directly linked with the feature change of the software system.
2. Those transactions that contain large number of changed classes associated with them are considered and given more importance. This is done because such transactions are associated with the quality of the system such as refactoring activities etc.
3. Multiple developers may be engaging with the same issue. Therefore, we have combined such multiple transactions into a single commit/transaction. This helps obtain a single transaction view of the same issue.
4. Duplicate transactions are simply ignored.

D. Cognitive Complexity

The cognitive complexity is a technique that helps measure the current understandability of different classes associated with the software system. A higher value always represents a class that that difficult to understand logic. The authors in [7] proposed a suite of cognitive complexity metrics. In this paper, we used the class complexity (CC) metric from the proposed suite. Equation (2) represents the formula for the CC metric.

$$CC = \sum_{p=1}^{s} \left( \sum_{j=1}^{q} \sum_{k=1}^{m} W_c (j, k, l) \right) (2)$$

Here, \(W_c\) is the cognitive weight associated with the three basic control structures (iteration, branch, and sequence) present inside different methods belonging to the class. This metric is chosen as it helps to measure the current maintenance efforts that must be devoted in order to increase understandability of the associated class and hence the underlying software system. Any smelly class that possesses a higher value of the cognitive complexity must be tackled with higher priority in order to increase the understandability of the software system. In this paper, the cognitive complexity of a smelly class is measured using a Sonarqube tool.

E. Importance of the User Feedback

The intelligence of developers can never be neglected and it can play an important role in the accurate detection and mitigation of a code smell. Therefore, in this paper, we have considered the human intelligence in the form of the feedback received based on different important parameters is considered. The various parameters that are considered important from the code smell prioritization point of view are as follows:

1. Smell Relevance. As per the authors in [10] different tools have a different number of smells identified for the same software system. This is due to the fact that each such software system is based on different detection strategy. Hence, we considered this parameter (smell relevance) important and considered user feedback on it.  
2. Refactoring Cost. Here, based on the possible refactoring associated with a code smell, the corresponding cost incurred is predicted. This cost can be high or low and directly affects the
priority of the underlying code smell from the actual maintenance team.

3. **Quality.** This parameter is chosen in order to consider the improvement in the understandability, readability and maintainability of the system.

4. **Localization of multiple code smells.** This parameter is designed to consider the user feedback for the situation in which a given class is affected by multiple code smells simultaneously.

In this paper, a team of expert developers is selected randomly and each of the team’s member is presented with the designed parameters. The team members analyze different code smells on these parameters and assign a number in the range $[1 \ldots 5]$ with 5 as the highest priority number and 1 denoting the least priority. Each individual feedback on each of the code smell is received and finally, the importance of different code smells is obtained by averaging the individual feedback.

**F. Code Smell Prioritization**

This is the last phase of the proposed approach and it uses a newly proposed ranking metric known as RankCS that assigns priority to different code smells. The proposed ranking metric is defined as the multiplicative product of the three criteria as explained in above subsections. Equation (3) represents the proposed ranking metric.

$$\text{RankCS} = \text{Severity} \times CC \times \text{UserFeedback} \quad (3)$$

The proposed rank metric RankCS computes the overall priority score for a given code smell based on the severity, cognitive complexity of the system, and the user understanding.

**IV. EXPERIMENTATION AND RESULTS**

As the experimentation, helps evaluate the proposed approach and confirms its feasibility. Therefore, in this paper, we considered evaluating our proposed approach against a standard object-oriented software system named HospitalAutomationWithJavaEE. We considered this software system because [13] has recently used it and it helps us to compare our results of the proposed approach with the rival similar approach. The considered software system helps us automate the Hospital functionality and consist of 49 classes and 61 transactions (from 03/2015 to 02/2018). The said system is analyzed with multiple code smell detection tools and consolidated list of code smells identified in the system is presented in the table-1. Total 18 classes are identified as affected with code smells, which must be dealt by the developers. These smelly classes are uniquely assigned a code smell ID as indicated in the table. The table also gives detail about the class name and the possible affected code smells. As per the proposed approach, the severity of a code smell is identified from the change history of the studied system and it is mined using SVN mining tool. The pre-processing of the identified commits from the identified change-history is done using a tool specially designed by us.

Table-2 shows the final code smell prioritization score (RankCS) along with the important user factor as recorded based on the questions presented to them. From the table-2, the smelly classes are prioritized in the following order as SC9, SC6, SC1, SC2, SC7, SC3, SC4, SC5, SC12, SC13, SC8, SC16, SC15, SC18, SC14, SC11, and SC17. The SC9 is the highly prioritized and SC17 is the least prioritized class. Further, it is the opinion of the authors that these smelly classes can be further shortlisted the obtained priority index based on some threshold. In figure-2, that shows the prioritization index for different smelly classes, the dotted line is used as the threshold and all classes whose score is above this line are considered more relevant.

**Table-1: Code Smells List in the studied System.**

<table>
<thead>
<tr>
<th>Smelly Class ID</th>
<th>Smelly Class Name</th>
<th>Code Smell Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC1</td>
<td>SaveAppointmentService.Java</td>
<td>FE (Feature Envy)</td>
</tr>
<tr>
<td>SC2</td>
<td>GetAvailableAppointments.java</td>
<td>FE, God Class (GC)</td>
</tr>
<tr>
<td>SC3</td>
<td>GetDoctors.java</td>
<td>FE</td>
</tr>
<tr>
<td>SC4</td>
<td>GetHospitals.java</td>
<td>FE, GC</td>
</tr>
<tr>
<td>SC5</td>
<td>GetPatientInformation.java</td>
<td>GC</td>
</tr>
<tr>
<td>SC6</td>
<td>SaveAppointments.java</td>
<td>GC</td>
</tr>
<tr>
<td>SC7</td>
<td>GetDistricts.java</td>
<td>FE</td>
</tr>
<tr>
<td>SC8</td>
<td>GetAppointmentsOfPatients.java</td>
<td>GC</td>
</tr>
<tr>
<td>SC9</td>
<td>AvailableAppointmentService</td>
<td>Duplicate Code (DC)</td>
</tr>
<tr>
<td></td>
<td>Service.java</td>
<td>DC</td>
</tr>
<tr>
<td>SC10</td>
<td>GetClinicPlaces.java</td>
<td>DC</td>
</tr>
<tr>
<td>SC11</td>
<td>DoctorService.java</td>
<td>DC</td>
</tr>
<tr>
<td>SC12</td>
<td>Klinikyerleri.java</td>
<td>DC</td>
</tr>
<tr>
<td>SC13</td>
<td>Hastaneler.java</td>
<td>DC</td>
</tr>
<tr>
<td>SC14</td>
<td>Doktorlar.java</td>
<td>DC</td>
</tr>
<tr>
<td>SC15</td>
<td>Hecler.java</td>
<td>DC</td>
</tr>
<tr>
<td>SC16</td>
<td>Iller_.java</td>
<td>DC</td>
</tr>
<tr>
<td>SC17</td>
<td>Randevususalteri.java</td>
<td>DC</td>
</tr>
<tr>
<td>SC18</td>
<td>DistrictService.java</td>
<td>DC</td>
</tr>
</tbody>
</table>

**Table-2: Prioritization of different identified code smells.**

<table>
<thead>
<tr>
<th>Smelly Class ID</th>
<th>User Feedback Score</th>
<th>Smell Rank/ Prioritization (RankCS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC1</td>
<td>4.14</td>
<td>0.45</td>
</tr>
<tr>
<td>SC2</td>
<td>2.79</td>
<td>0.34</td>
</tr>
<tr>
<td>SC3</td>
<td>3.25</td>
<td>0.27</td>
</tr>
<tr>
<td>SC4</td>
<td>2.25</td>
<td>0.22</td>
</tr>
<tr>
<td>SC5</td>
<td>1.87</td>
<td>0.19</td>
</tr>
<tr>
<td>SC6</td>
<td>3.82</td>
<td>0.64</td>
</tr>
<tr>
<td>SC7</td>
<td>3.73</td>
<td>0.30</td>
</tr>
<tr>
<td>SC8</td>
<td>0.97</td>
<td>0.14</td>
</tr>
<tr>
<td>SC9</td>
<td>4.23</td>
<td>0.72</td>
</tr>
<tr>
<td>SC10</td>
<td>1.17</td>
<td>0.12</td>
</tr>
</tbody>
</table>

---

*https://github.com/ilknune/HospitalAutomationWithJavaEE*
Figure-2: Prioritization plots for different smelly classes.

V. CONCLUSION AND FUTURE WORKS

This paper presents a novel approach that helps prioritize different code smells present in a software system. Removing code smells helps improve the underlying software quality by applying different refactoring operations. Different refactoring is associated with different maintenance efforts. Therefore, this paper targets prioritizing different code smells associated with smelly classes. This prioritization is based on three criteria namely Severity, understandability, and user importance for the particular smelly class. The severity is determined using the change-history and efforts to improve understandability is measured using the Cyclomatic complexity metric. The proposed approach considers the user feedback as important and finally measure the overall code smell rank using a new metric proposed in this paper. This proposed metric is obtained as a multiplicative product of the used three criteria.

The future work associated with the proposed approach is many. Firstly, other factors can be determined that are directly associated with the code maintenance effort and their feasibility can be checked for prioritizing code smells. Secondly, the proposed approach can be extended to provide visualization support for prioritization. Finally but not the last, the proposed approach can be tested for prioritizing refactoring associated with the identified code smells.

REFERENCES

Authors Profile

Randeep Singh is a Research Scholar in Department of Computer Science & Engineering, M. M. Engineering College, M. M. (Deemed to be University) Mullana, Ambala, Haryana, India. Randeep Singh received M. Tech from Kurukshetra University. Randeep Singh is in teaching and Research & Development since 2008. He has published about 10 research papers in International, National Journals and Refereed International Conferences. His current research interests are in Software Engineering.

Dr. Amit Kumar Bindal is an Associate Professor in Department of Computer Science & Engineering, M. M. Engineering College, M. M. (Deemed to be University) Mullana, Ambala, Haryana, India. Dr. Bindal received Ph.D from Maharishi Markandeshwar University, M. Tech (Computer Engineering) from Kurukshetra University and B. Tech. in Computer Engineering from Kurukshetra University Kurukshetra. Dr. Bindal is in teaching and Research & Development since 2005. He has published about 60 research papers in International, National Journals.
and Refereed International Conferences. His current research interests are in Wireless Sensor Networks, Underwater Wireless Sensor Networks, Sensors and IOT etc.

Dr. Ashok Kumar is an Ex-Professor in Department of Computer Science & Engineering, M. M. Engineering College, M. M. (Deemed to be University) Mullana, Ambala, Haryana, India & former Professor of Kurukshetra University. Dr. Kumar is in teaching and Research & Development from more than 40 years. He has published many research papers in International, National Journals and Refereed International Conferences. His current research interests are in Software Engineering, Digital Image Processing.