A User Feedback Centric Approach for Detecting and Mitigating God Class Code Smell Using Frequent Usage Patterns

Randeep Singh, Amit Bindal, and Ashok Kumar

***Abstract*— Code smells are the fragments in the source code that indicates deeper problems in the underlying software design. These code smells can hinder software evolution and maintenance. Out of different code smell types, the God Class (GC) code smell is one of the many important code smells that directly affects the software evolution and maintenance. The GC is commonly defined as a much larger class in systems that either know too much or do too much as compared to other classes in the system. God Classes are generally accidentally created overtime during software evolution because of the incremental addition of functionalities to it. Generally, a GC indicates a bad design choice and it must be detected and mitigated in order to enhance the quality of the underlying software. However, sometimes the presence of a GC is also considered a good design choice, especially in compiler design, interpreter design and parser implementation. This makes the developer's feedback important for the correct classification of a class as a GC or a normal class. Therefore, this paper proposes a new approach that detects and proposes refactoring opportunities for GC code smell. The proposed approach makes use of different code metrics in combination along with utilizing user feedback as an important aspect while correctly identifying the GC code smell. The proposed approach that considers combined use of code metrics, is based on two newly proposed code metrics in this paper. The first newly proposed metric is a new approach of measuring the connectivity of a given class with other classes in the system (also termed as coupling). The second newly proposed code metric is proposed to measure the extent to which a given classes make use of foreign member variables. Finally, the proposed approach is also empirically evaluated on two standard open-source commonly used software systems. The obtained result indicates that the proposed approach is capable of correctly identifying the GC code smell.**

***Keywords*— Code Smell, God Class, Refactoring, Software Evolution, Maintenance, Quality.**

I. INTRODUCTION

In software engineering, software maintenance is a continuous and mandatory activity that helps implement the corrective, adaptive, perfective, and preventive steps in a software development life cycle. These steps are the result of a change in user requirement, environmental change, bug fixing need, quality improvement of the underlying software system, or other maintenance activity that is carried out on the system. Moreover, the long-term modification in the underlying code of software may weaken the underlying design of the system. This may be due to the lack of knowledge of the software developer, market demands or other careless activities during the software development process. The degraded design results in the decreased quality and the increased maintenance cost of the system. The issues that result in the poor underlying design and also worsen the software maintainability are termed as code smells [9]. Therefore, it is mandatory to detect and mitigate these smells in order to improve software durability. The process of mitigating the identified code smell is termed as refactoring. Refactoring helps us to boost various quality parameters of the code/design like maintainability, extensibility, and understandability. The refactoring aims at altering the underlying structure of the system without modifying the actual working of the system.

The author in [9] identifies and categories different poor symptoms present in the source-code into different types of code smells. Out of these different code smells, some of the commonly perceived, more concerned, and highly rated as sever from the developer's perspective are Complex Class, Spaghetti Code, Long Method, and God Class [20]. Out of these important code smells, we target identifying God Class (GC) code smell in this paper. Further, the identified GC code smell is resolved by suggesting Extract Class refactoring suggestions to the software maintainer team after considering team feedback [9][30].

Several studies in literature targets detection and refactoring of the code smells present in the source code of a software system [15, 9, 6, 1, 13]. These studies targets using various traditional code metrics and machine learning algorithms to detect code smells.

Manuscript received February 24, 2019; revised April 1, 2019. Date of publication July 25, 2019. Date of current version July 25, 2019. The associate editor prof. Tihana Galinac Grbac has been coordinating the review of this manuscript and approved it for publication.

Authors are with the Department of Computer Science & Engineering, Maharishi Markandeshwar (Deemed to be University), India.

E-mails:randeeppoonia@gmail.com,amitbindal@mmumullana.org, mailtodr.ashok@rediffmail.com).

Digital Object Identifier (DOI): 10.24138/jcomss.v15i3.720

Besides these, various automated and semi-automated tools are proposed and compared in literature for the purpose of code smell detection and removal [8, 16, 24, 19]. To the best of author's familiarity, these various approaches already proposed in the literature make use of different metrics in a combination that is defined in an ad-hoc manner. The suitability of such a combination of different metrics needs to be determined for the correct identification of the code smells. Moreover, it is necessary to determine if a new combination of code metrics can improve the accuracy of code-smells detection. Further, even if the same set of metrics are used the threshold limit used by them may be different. The constant threshold limit used may not be feasible always. This is more explained in the study design section of the paper. These approaches also miss out the user intelligence (available in the form of feedback) during GC identification process. It is the opinion of the author's in this paper that this feedback can play an important role in the correct identification of the GC code smell. Moreover, due to the abstract definition of various kinds of smells in [9], ambiguity is always present and different developers are having different perspectives on these code smells [23]. This makes the user knowledge (mainly developer engaged with the maintenance team) important that can highly deviate the overall accuracy in determining correct code-smell. User feedback can help determine more accurate code-smells and thus help in reducing the overall maintenance cost of the underlying software system (due to less number of classes on which the maintenance team need to work). These identified limitations in the literature become the research gap for this paper.

This paper targets filling these identified gaps by incorporating the user feedback (of mainly developers) during the process of GC code smell detection. The proposed approach in this paper recommends and proposes a new set of metrics that together can increase the accuracy while detecting GC code smell. Moreover, the threshold limit that is used in the proposed approach is also not kept constant and it can be easily changed as per the comments provided by the feedback team. The proposed approach differs from the others in that 1) it makes combined use of two new more robust and dynamic metrics; 2) it considers user feedback as important and it can help reduce the overall maintenance cost by correctly identifying GC code smell, and 3) the used threshold are not kept constant, instead, they are dynamic in nature. The main contribution of this paper can be summarized as follows:

1. To propose a new coupling metric that can measure how much a given class is dependent on the rest of the classes in the system.
2. To propose a new metric for measuring the extent to which a given class makes use of the foreign class member variable.

The whole paper is divided into the following six sections. Section II gives details about the literature work that is devoted to code smell detection and mitigation. Section III describes the proposed approach in detail and section IV provides details about how the whole study designed and conducted in order to evaluate the proposed approach. Section V gives discussion details about the obtained results as part of the experimental evaluation of the proposed approach. Finally, section VI provides the concluding remarks and possible subsequent work directions.

II. Literature Survey

This section in the research paper gives information about the work already carried out in the field of code smell detection and mitigation using refactoring. The study of code smells detection strategies is always remains a subject of recent studies in the literature. Most of these studies are based on utilizing the knowledge obtained from the underlying source-code and usually uses different metrics in combinations [4, 5, 14, 12, 25].

Fowler categorized and classified different symptoms of poor design into 22 types (code smells) and gives details about the possible refactoring opportunities for different code smells [9]. The authors in [2] specified a total of 40 anti-patterns present in the source code that may give rise to worse design for the underlying system. The authors in [11] studied the code smell detection as a distributed optimization problem using parallel evolutionary algorithms. The process of code smell detection is also targeted by applying different machine learning algorithms [8]. The authors performed the largest experimental evaluation by considering 16 different machine learning algorithm which is applied to a total of 74 software systems belonging to different domains. Similarly, the authors in [13] present an approach named SMURF for detecting anti-patterns using the machine learning algorithms and considering the practitioner's feedback into account. The authors in [29] studied different versions of the same software system for determining when and why code smells are introduced during development. The authors carried out a large investigation of 200 open source systems.

The literature work also focusses on determining the relationship between identified code smells and different qualities of the software system. Yamashita et al. perform a detailed investigation for determining the extent to which the maintainability of software can be predicted based on the known code smells [25]. The authors in [7] perform an empirical study on a software belonging to different domains aiming at identifying different code smells and determining a possibility of correlation among code smells that affects each of the domains. The authors in [12] presented details about various object-oriented (OO) metrics and show how these metrics can be efficiently and effectively used in code-smell detection. This use is based on the combined use of various OO metrics. The authors also presented various visualization techniques that help us easily understand complex software systems.

Several semi/ fully- automatic tools are also proposed in the literature that aims at detecting underlying code smells and applying the refactoring in order to improve the underlying code quality [26, 15, 27]. The authors in [28, 6] surveyed different tools and also performed a comparison of different tools in order to determine their capability for code smell detection.

There are basically two main differences between our

proposed approach in this paper and the approaches proposed by various researchers in literature. First, different approaches in literature do not take user feedback during code smell detection. However, different developers have different perception about a code smell. Therefore, considering user feedback can highly deviate the obtained results for a code smell. Secondly, different researchers have used a combination of different metrics for detecting code smell and they have used different thresholds for the metric values. The choice of the

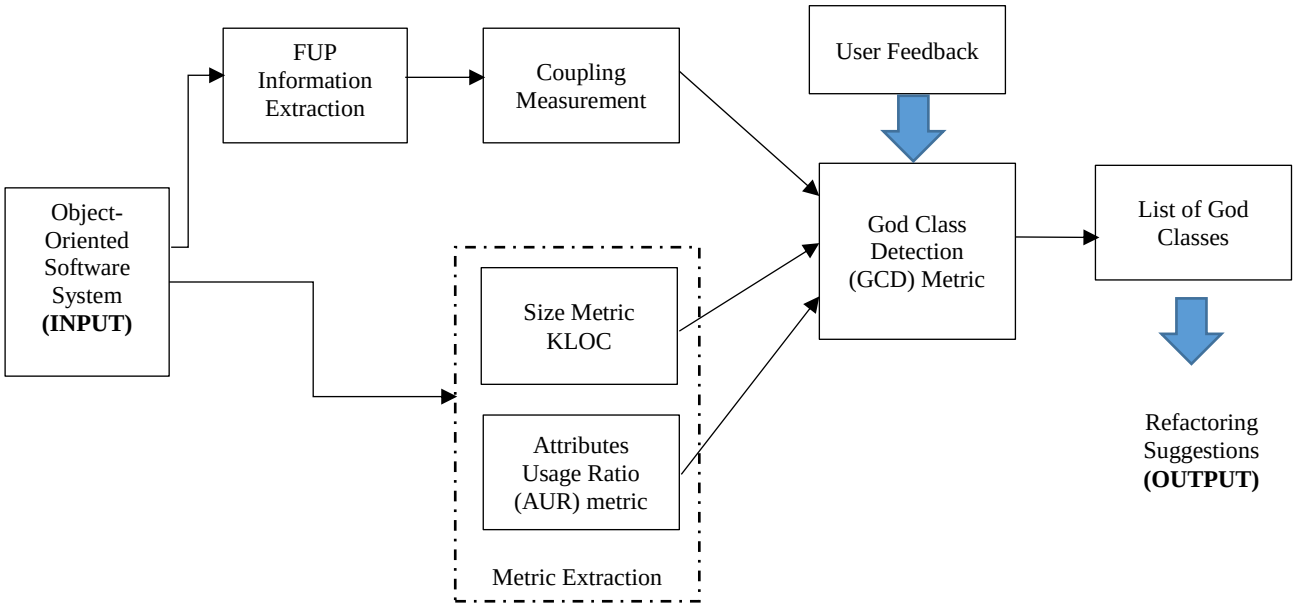

Fig. 1. Proposed Approach for Detecting and Mitigating God Class (GC) Smell.

threshold value is arbitrary and/ or is based on some specific software under study. Therefore, it is our belief that the same threshold may or may not hold on other software too. These identified two research gaps are targeted in this paper.

III. Proposed Methodology

In this part of the paper, we describe the details of the newly proposed approach, which is designed for detecting the GC code smell. The newly proposed approach aims at detecting the GC code smell from an object-oriented software system. After detecting the GC code smell, the proposed approach mitigates it by proposing an extract class refactoring to the software maintainer team. In the proposed approach of this paper is based on the two newly proposed code metrics. The first newly proposed metric (COUPGC) measures the coupling among different classes of a software system. This newly proposed metric is based on utilizing the Frequent Usage Patterns (FUPs) for measuring more accurate coupling. These FUPs are available in the form of member variable usage patterns in the source code of a software system. The second newly proposed metric is the Attributes Usage Ratio (AUR). This metric measures the extent to which a given class uses the attributes of other classes in the system. Moreover, we also propose a new combination of code metrics that can detect the GC presence with more accuracy. Finally, a new GC detection metric is proposed based on the previous code metric combination and the feedback of the developers. Figure-1 shows the overview of the working of the proposed approach.

The proposed approach considers utilizing user feedback while detecting the GC code smell. This feedback is obtained from the experts in the maintenance team or the special experts specially hired for this purpose. Based on the manual inspection of the various code-smell lists (various code-smell detection tools), it is observed that the lists differ from each other. This gives rise to the possibility of false detection by different tools. Further, tackling all these smells together results in increased maintenance cost of the underlying software system. At this stage, the expert knowledge can help in reducing maintenance cost by eliminating the false detections. The detailed working of the proposed approach is given in the following sub-sections:-

A. FUP Information Extraction

This step of the proposed approach aims at extracting FUP information by parsing the source-code of different classes belonging to the software system. All those member variables, associated with different classes of the software, which are either directly or indirectly used within the class constitutes the FUP information for the said class. This information is available as sets of member variable names. Here, the indirect usage refers to the member variable usage due to a function call to the same or different classes. The scope of the identified FUP set is the whole software system and it may contain names of foreign member variables (member variables belonging to other classes). This FUP information is further represented in the form of a vector called FUP vector. The size of this vector is N and it represents the count of distinct member variables belonging to the whole software system. In the vector representation, the index denotes the unique member variable in the system and the value at that index represents the frequency (count) of the usage for the corresponding member

variable. Here, if a class does not use any of the member variables then its corresponding frequency is zero.

B. FUP Information Extraction

This step of the proposed approach aims at extracting FUP information by parsing the source-code of different classes belonging to the software system. All those member variables, associated with different classes of the software, which are either directly or indirectly used within the class constitutes the FUP information for the said class. This information is available as sets of member variable names. Here, the indirect usage refers to the member variable usage due to a function call to the same or different classes. The scope of the identified FUP set is the whole software system and it may contain names of foreign member variables (member variables belonging to other classes). This FUP information is further represented in the form of a vector called FUP vector. The size of this vector is N and it represents the count of distinct member variables belonging to the whole software system. In the vector representation, the index denotes the unique member variable in the system and the value at that index represents the frequency (count) of the usage for the corresponding member variable. Here, if a class does not use any of the member variables then its corresponding frequency is zero.

Figure-2 shows a typical representation of a FUP vector for a system containing three classes A, B, and C having two member variables each. Here, the frequency for the usage of member variable B.M2 (second member variable of class B) is zero. It denotes that class A is not using the member variable M2 of class B.

Index →	A.M1	A.M2	B.M1	B.M2	C.M1	C.M2
A	5	9	1	0	7	6

Fig.2. A typical FUP vector representation used for the denoting the FUP set of class 'A'.

B. FUP Based Coupling Measurement

This phase of the proposed approach aims to measure the coupling strength of a given class with the rest of the other classes in the system. The extracted FUP information is the key in this step and these are used as input in this phase. For measuring the coupling strength, a new metric called $COUP_{GC}$ is proposed. This proposed metric is defined in equation (1) of this paper.

$$COUP_{GC}[i] = \begin{cases} \bigvee_{j=1;j\neq i}^{N} \left(\frac{\sum Ref_{Intra}[j]}{Ref_{Total}[i]}\right) \\ 0; \quad Otherwise \end{cases} \tag{1}$$

Here, $Ref_{Intra}[j]$ is the total number of non-zero references made for member variable of class j within the class i. The $Ref_{Total}[i]$ is the total number of non-zero references made by the class i for the rest of the classes in the system. These two values are computed from the FUP vector representation of FUP information of the class i. The coupling of the class i with itself is zero as indicated within the formula in equation (1). The value of the $COUP_{GC}$ is always in the range [0…1]. The measured coupling value is stored in a square matrix of dimension N x N. here, N represents the count of the classes in the software.

In this paper, we propose a new coupling metric and in literature there are a number of other coupling metrics such as CBO, RFC (in CK-metric suite), Afferent and Efferent (in Martin metric suite), etc. Our proposed coupling metric differs from the already existing coupling metrics in the following ways:-

1. The existing coupling metrics consider the interaction among different classes based on the direct use of the method and member variables. These metrics lack considering dependencies arising due to the indirect calling of methods between classes.
2. Based on our manual inspection, it was determined that dependencies arising due to method calls are not always harmful because sometimes methods present in a class are only informative in nature (methods that prints some message/ information and do not access/ change any member variables). This kind of dependencies is avoided in our proposed approach by considering FUP relations.

C. Size Metric Extraction

This part depicted in the approach proposed in figure-1 aims at extracting the size of different classes using a well-known size metric called KLOC. The size metric in our proposed metric is used to measure another dimension of the complexity of the underlying classes. In our proposed approach, the size metric is computed for each of the class belonging to the software. This metric is chosen because generally, a large class is difficult to understand and modify and hence increased maintenance cost.

D. Attributes Usage Ratio (AUR) Metric

This phase of the proposed approach measures the extent to which a given class uses foreign member variables. For this purpose, a new metric has been proposed as shown in equation (3). The proposed metric is computed as an average and is based on the usage factor $Usage_i\,[j]$. The usage factor measures the extent to which a given class, say i, uses the member variables of other class say j. The definition of this metric is as shown in equation (2).

$$Usage_i\,[j] = \frac{Number\ of\ Member\ Variables\ of\ j\ used\ by\ i}{Total\ number\ of\ Member\ Variables\ in\ j} \tag{2}$$

$$AUR_{GC}\,[i] = \sum_{\mathrm{j}\,=\,1;\,\mathrm{j}\,\neq\,\mathrm{i}}^{N} Usage_i\,[j] \Big/ N \qquad (3)$$

E. God Class Detection (GCD) Metric

This phase of the proposed approach aims at detecting the presence of God Class (GC) in a software system. In our proposed approach, a GC is detected based on the following three factors:

1. Lower inner-class cohesion and higher coupling. The proposed $COUP_{GC}[i]$ metric is used to measure the degree of coupling of a class with other classes. The overall coupling of a given class is computed as the average of the coupling values of the classes on which it is dependent. A threshold value is used to select the minimum coupling value that distinguish between a GC and a normal class.
2. A high complexity. In our proposed approach, the underlying complexity of a class is measured using the standard KLOC metric proposed in the literature. This metric is found to be a good indicator for measuring program complexity [10]. This is because the authors in [10] found strong empirical evidence of having a strong linear relationship between Cyclomatic Complexity and Lines of Code metric. We have chosen this standard metric to measure complexity because it is easy to measure and is also used by other researchers to detect code smells.
3. High usage for foreign member variables. The proposed $AUR_{GC}\,[i]$ metric is used to measure the extent to which a given classes uses the foreign member variables. A class showing high usage to member variables of other classes gives an indication towards the presence of GC code smell.

Based on the above mentioned three factors, the proposed GCD metric for identifying god classes is presented as follows in equation (4):-

$$GCD = \bigvee_{i=1}^{N} (\,COUP_{GC}[i] > \alpha) \text{ AND } (KLOC > \beta) \text{ AND } (AUR_{GC}\,[i] > \gamma) \qquad (4)$$

The proposed GCD metric makes use of three constraints $\alpha, \beta, and\ \gamma$ for the three considered metrics. These constraints are used as a threshold and above this threshold value, the corresponding metric value indicates towards the GC smell presence. The three constraints consolidates the user knowledge and are user-specific (developer's feedback) In this paper, their current values are taken as the average of the corresponding metrics over the software level rather than keeping them as constant. In the equation (4), the process of detecting GC is dependent on the user feedback, which is the maintenance team understanding for the presence/ absence of GC code-smell in the specified class of the software system. As the user feedback is obtained based on the expertise of the maintenance team and different members can have different feedback on the GC code-smell.

Therefore, in order to reduce the bias, we collected their independent feedback and then obtained the final feedback score by averaging their independent scores.

F. Refactoring Suggestions

This final part of the approach depicted in figure-1 aims at providing refactoring suggestions. After the GC list is identified in the above step using a metric combination and user feedback, the GCs are refactored using the *Extract Class* method. The proposed approach does not automatically refactor the underlying software system, instead, it only gives suggestions to the maintainer team regarding extract class opportunities.

IV. STUDY DESIGN

In this part of the paper, we provide details about the experiment conducted in order to empirically evaluate the proposed approach for detecting GC code smell. This section is split into the following sub-sections:

A. Target Systems

In order to empirically evaluate the proposed approach, we consider two Object-Oriented software systems whose details are presented in the Table-I. These software systems are selected because they are widely being used among researchers in code smell evaluation and other software maintenance-related studies [19, 3, 31]. The first system, MobileMedia[1] (MM), is an open-source software product line project that manipulates photos, videos, and music on mobile devices. Similarly, the Health Watcher[2] (HW) is a web-based software system designed to help the user to register, manage the complaints in a public health care system.

TABLE I
METRICS FOR TARGET SYSTEMS

S.No.	System Name	Version	No. of Class	Size (KLOC)	No. of Methods
1	MobileMedia (MM)	9	55	3.216	290
2	Health Watcher (HW)	10	118	8.702	671

B. GC Code Smell Reference List

Before performing the experimental evaluation, the underlying source code of the considered systems is systematically analyzed. The aim is to discover the actual traces of GC code smell present within these systems. For this purpose, we sent an invitation to 15 developers having at least 5 years of expertise in software development and project

[1] http://ptolemy.cs.iastate.edu/design-study/#mobilemedia

[2] http://ptolemy.cs.iastate.edu/design-study/#healthwatcher

management. This invitation is sent to multiple IT companies and they were asked to nominate different experts. Finally, the team members are randomly selected and they are unaware of each other and our motive. After selecting the team, each member is presented with the source code of the software under study. Each team member is asked to identify the god classes present within the software system based on the source-code presented to them and their domain knowledge. Each of the team members finally returns a list of god classes present within the software.

After receiving the lists, a common consensus is made among the team by arranging a meeting and promoting discussion among them. Finally, a single list containing the names of god classes regarding the considered software is prepared. This final list is ultimately used as a reference list to compare our results. Table-II shows the final reference list of GCs in the two considered software systems.

TABLE II
GOD CLASS REFERENCE LIST FOR THE TARGET SYSTEMS

Software Name	Number of God Class	Classes belonging to the Reference List
Health Watcher(HW)	2	HealthWatcherFacade, HealthWatcherFacadeInit
Mobile Media(MM)	7	AlbumController, CaptureVideoScreen, MediaAccessor, MediaController, MediaUtils, PhotoViewCntroller, SmsMessaging

C. Evaluation Criteria Used

In order to evaluate the proposed approach, we rely on three key information retrieval metrics namely Precision, Recall, and F-Measures [21].

These metrics compare our obtained results against the obtained reference list. The precision metric is used to measure the accuracy and is defined as the number of identified relevant GC code smells by the total number of identified code smells. Similarly, the recall metric is used to measure the completeness of the obtained results. It is defined as the ratio of the total number of identified relevant GC code smells and the total number of god classes in the reference list. Here, a high value of precision indicates that the proposed approach is capable of correctly identifying the god classes. Similarly, a score for the recall metric indicates that nearly all the god classes in the reference list are in the obtained results.

V. RESULTS AND DISCUSSION

This section of the paper discusses the experimental results and finally provides a discussion on the analysis of the obtained results. In order to analyze and discuss the obtained results, the following research questions are formulated in this paper:-

RQ1. How the proposed approach performs in detecting the GC code smell?

RQ2. How the proposed approach does differ in comparison to traditional metric-based approach?

A. Summary of the Detected GC Code Smells

Table-III shows the details about the total number of the detected GC code smells identified using our proposed approach. Table-II specifies the count of the GC code smell detected in different systems for two scenarios viz when the said systems are analyzed by the proposed approach and the reference list as obtained from a team of experts.

TABLE III
TOTAL NUMBER OF IDENTIFIED GC CODE SMELL

S.No.	System Name	# of GC Code Smell Detected	# of GC Code Smells in the Reference List
1	MobileMedia (MM)	9	7
2	Health Watcher (HW)	3	2

RQ1. How the proposed approach performs in detecting the GC code smell?

The capability of the proposed approach in detecting the GC code smells is evaluated using the standard information retrieval metrics as specified in the section above. The table-4 shows the results of the evaluation. In the presented results the obtained average precision value is 95% and it indicates that the proposed approach is capable of accurately predicting the GC code smells. Similarly, the obtained recall value is 100% and it indicates that the proposed approach detects all the GC code smells as listed in the reference list. From the results depicted in the table-IV, it can be concluded that the proposed approach in this paper stands firm in detecting GC code smell. Moreover, the original definition of GC as proposed by [12] is also tested on the considered software systems using the *iPlasma* tool[3]. This tool is mentioned in the [19] and is based on the detection strategies as defined by Lanza and Marinescu [12]. This tool is unable to detect any presence of GC code-smell in the two considered software systems.

TABLE IV
EVALUATION OF THE PROPOSED APPROACH

S.No.	System Name	Precision	Recall	F-Measure
1	MobileMedia (MM)	92.6%	100%	96.2%
2	Health Watcher (HW)	97.3%	100%	98.6%

RQ2. How the proposed approach does differ in comparison to traditional metric-based approach?

This research question aims at comparing our proposed approach with the closest rival approach present in the

3 http://loose.upt.ro/iplasma/

literature. The authors in [18] conclude that the KLOC (Line Of Count per Thousand) and WMC (Weighted Method Count) are the keys in detecting the GC code smells. Therefore, we have considered an approach proposed by [17] for comparison purposes. The authors consider evaluating GC code-smell in three open-source software systems. They are focused on determining the trend of GC code-smell along with the evolution of the considered software. Here, the authors have used WMC metric along with other traditional metrics namely Tight Class Coupling (TCC) and Access to Foreign Data (ATFD) for detecting the GC code smell. The authors used these three metrics in combination for detecting the GC code smell with different thresholds for the metrics. Here, it is necessary to note that the experimental study conducted by the authors in [17] is based on the same approach as proposed by the authors in [12].

TABLE V

S.No.	System Name	# Classes	# Detected God Classes (GC)		% Classes with GC Smell		% Change in #GC
			The approach by [17]	Our Proposed Approach	The approach by [17]	Our Proposed Approach	
1	Lucene	651	26	30	3.99%	4.61%	0.62%
2	Xerces	712	57	69	8.00%	9.69%	1.69%
3	Log4j	337	10	11	2.96%	3.26%	0.30%

COMPARISON OF THE GC DETECTION

Table-V shows the comparison results for the two approaches. Column 4 and 5 shows the total number of GC detected by different approaches. The column number 6 and 7 show the percentage classes that are affected by GC code smell in the considered software system. The last column shows the percentage change in the god classes present in the system. From the values, it is clear that the GC's present in the Xerces system is higher and it correlates with the pattern observed by the authors in [17] based on the study of the different timelines of the considered system. They observed that GC's count keep on increasing with the timeline of the development of the Xerces system. Similar are the results for the rest of the considered systems. The obtained results in table-4 indicate that our proposed approach is more capable of identifying the GC opportunities which can be easily mitigated using the extract class refactoring. This is due to the fact that the authors in [17] missed out the important KLOC metric as per the author's investigation in [18]. Further, the table-6 gives details about the class names that are affected by GC code-smell. In this table, the list is shown only for Log4j software system because the authors in [17] do not provide an exhaustive list of class names that are affected with the GC code-smell for all the software systems under study. Here, we rely on the iPlasma tool for getting class names that are affected with the GC code-smell as per the rival approach. From the list mentioned in the table-6, it is noted that the *CronExpression.java* class is not detected by rival approach and further all the classes detected by the approach proposed by [12] are also detected by our proposed approach. By manual inspection of the *CronExpression.java* class, we came to know that the rival approach is unable to detect it as GC because the value of the ATFD metric value is <5. However, this class is highly coupled and together is hard to understand due to its large size.

VI. CONCLUSIONS AND FUTURE WORK

The maintainability of a software system broadly affected by the choice of the suitable metrics that are capable of accurately detecting the code smells present in the system. The source code metrics are being widely used in software maintenance and code smell detection. The present study in this paper aims at detecting the GC code smell in a software system. The detection process is based on the use of a traditional metric (KLOC) and two newly proposed source-code metrics namely $COUP_{GC}$ and AUR_{GC} metric. The proposed approach is based on the simultaneous occurrence of three conditions namely 1) presence of a highly coupled class, 2) the presence of a complex class, and 3) high usage for the foreign members within a class.

TABLE VI
SUMMARY OF THE GC NAMES

Approach	God Class Names
Rival Approach [17]	CommandLine, FastDatePrinter, FastDateParser, DatePatternConverterTest, Interpreter, PluginBuilder, RollingFileManager, Server, SLF4JLogger, FixedDateFormat
Proposed Approach	CommandLine, FastDatePrinter, FastDateParser, Interpreter, PluginBuilder, RollingFileManager, Server, SLF4JLogger, DatePatternConverterTest, FixedDateFormat, CronExpression

These conditions are detected with the help of source-code metrics. Further, as part of the investigation, the proposed approach is also compared with the approach proposed by [17]. The comparison results indicate that our proposed approach is

capable of identifying more GC code smells. Moreover, as the proposed approach is tested on the Java software systems, however, it can be easily applied to the software systems designed in another language. This is because the metric used in the proposed approach can easily be computed on other software systems (written in another language) and they are not confined only to the systems designed in Java language.

The future work related to our present work is many. First of all, an experimental evaluation can be carried out in order to determine whether the proposed approach can outperform the various machine learning technique. Secondly, the suitability of other metrics can be evaluated for the purpose of detecting the GC code smell. Thirdly, the proposed approach can be tested or modified for its ability to detect another kind of code smells. Finally, we are preparing a prototype of a tool that can fully automate the proposed approach of this paper.

REFERENCES

[1] L. Amorim, E. Costa, N. Antunes, B. Fonseca and M. Ribeiron," Experience Report: Evaluating the Effectiveness of Decision Trees for Detecting Code Smells", *in Proc. of the IEEE 26th International Symposium of Software Reliability Engineering(ISSRE)*, IEEE Computer Society, Washington, DC, USA, pp. 261-269, 2015.

[2] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis", 1st ed., John Wiley and Sons, March 1998.

[3] J.M. Conejero et al." On the Relationship of Concern Metrics and Requirements Maintainability", *Inf. and Sof. Technology (IST*), 2011.

[4] M. Eaddy et al." Do Crosscutting Concerns Cause Defects", *IEEE Trans. on Software Engineering*, pp. 497-515, 2008.

[5] F. Ferrari, et al. "An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs", in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pp. 65-74,2010.

[6] F. A. Fontana, et al. "Automatic Detection of Bad Smells in Code: An Experimental Assessment.", *Journal of Object Technology*, vol. 11(2), no. 5, pp. .1-38,2012.

[7] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka, "Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains,", *IEEE International Conference on Software Maintenance*, Eindhoven, pp. 260-269,2013.

[8] F. A. Fontana, M. V. Mantyla, M. Zanoni, and A. Marino," Comparing and experimenting machine learning techniques for code smell detection", *Empirical Softw. Engg.* , vol. 21, no. 3, pp. 1143-1191, June 2016.

[9] M. Fowler, "Refactoring – Improving the Design of Existing Code", 1st ed., Addison-Wesley, June 1999.

[10] G. Jay, J. Hale, R. Smith, D. Hale, N. Kraft, and C. Ward, "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship", *Journal of Software Engineering and Applications*, vol. 2, no. 3, pp. 137-143,2009.

[11] K. Wael, et al. "A cooperative parallel search-based software engineering approach for code-smells detection", *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841-861,2014.

[12] M. Lanza., R. Marinescu., "Object-Oriented Metrics in Practice", Springer-Verlag, New York, Inc., 2006.

[13] A. Maiga., N. Ali, N. Bhattacharya, A. Sabane, Y-G. Gueheneuc and E. Aimeur, "SMURF: SVM-based Incremental Anti-pattern Detection Approach." *19th Working Conference on Reverse Engineering*, pp. 466–475, 2012.

[14] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws", *in Proc. of Int'l Conf. on Software Maintenance (ICSM)*, pp. 350-359,2004.

[15] N. Moha, Y. Gueheneuc, L. Duchien, and A. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20-36, Jan.-Feb, 2010.

[16] E. Murphy-Hill, A. Black, "An interactive ambient visualization for code smells", *In Proceedings of the 5th international symposium on software visualization*, ACM, pp 5–14,2010.

[17] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems", *IEEE International Conference on Software Maintenance*, Timisoara, pp. 1-10, 2010.

[18] J. Padilha, J. Pereira , E. Figueiredo , J. Almeida , A. Garcia, C. Sant'Anna," On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study", *In Jarke M. et al. (eds) Advanced Information Systems Engineering. CAISE. Lecture Notes in Computer Science*, vol. 8484, Springer, Cham, 2014.

[19] T. Paiva, A. Damasceno., E. Figueiredo et al. "On the evaluation of code smells and detection tools", *J Softw Eng Res Dev*, vol. 5, no. 7,2017.

[20] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia," Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells", *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pp.101-110, September 29-October 03, 2014.

[21] W. J. Perry, K. Allen, B. M. Madeline, "Machine literature searching X. Machine language; factors underlying its design and development", *American Documentation*., vol. 6, no. 4,pp. 242,1955.

[22] R. Ghulam, and Z. Arshad, "A review of code smell mining techniques", *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867-895, 2015.

[23] J.A.M. Santos, M.G.D. Mendonça, and C.V.A. Silva, "An exploratory study to investigate the impact of conceptualization in god class detection", *In Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering, EASE '13*, New York, USA. ACM, pp. 48–59, 2013.

[24] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, "JDeodorant: identification and removal of type-checking bad smells", *in Proceedings of the 12th European conference on software maintenance and reengineering. IEEE*, pp 329–331, 2008.

[25] A. Yamashita, L. Moonen," To what extent can maintenance problems be predicted by code smell detection? An empirical study", *Inf. Softw. Technol.*, vol. 55, no. 12, pp. 2223–2242, 2013.

[26] C. Marinescu, R. Marinescu, PF. Mihancea, D. Ratiu, R. Wettel, "iPlasma: an integrated platform for quality assessment of object-oriented design", *In Proceedings of the 21st IEEE international conference on software maintenance. IEEE*, pp. 25–30, 2005.

[27] N. Zazworka, C. Ackermann, "CodeVizard: a tool to aid the analysis of software evolution", *in Proceedings of the 4th international symposium on empirical software engineering and measurement*, ACM, article 63, 2010.

[28] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo," A review-based comparative study of bad smell detection tools", *In: Proceedings of the 20th international conference on evaluation and assessment in software engineering (EASE '16*, ACM, article 18, 2016.

[29] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia and D. Poshyvanyk, "When and why your code starts to smell bad", *In Proceedings of the 37th international conference on software engineering. IEEE Press*, pp. 403–414, 2015.

[30] S.A. Vidal, C. Marcos, & Díaz-Pace, "An approach to prioritize code smells for refactoring", *J.A. Autom Softw Eng* ,vol. 23, no. 3,pp. 501-532,2016. https://doi.org/10.1007/s10515-014-0175-x

[31] A. Shatnawi, A. Seriai, H. Sahraoui, "Recovering Architectural Variability of a Family of Product Variants", *In: Schaefer I., Stamelos I. (eds) Software Reuse for Dynamic Systems in the Cloud and Beyond. ICSR 2015*, Lecture Notes in Computer Science, vol 8919. Springer, Cham, 2014.

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.