International Journal of Recent Technology and Engineering (IJRTE)
ISSN: 2277-3878, Volume-8 Issue-6, March 2020

Long Method and Long Parameter List Code Smells Detection using Functional and Semantic Characteristics

Randeep Singh, Amit Bindal, Ashok Kumar

Abstract*: Long-term evolution and maintenance of the software system result in the introduction of different kinds of code smell in the underlying source code of the software system. These code smells are the direct indication of degraded quality and increased understandability and maintainability efforts at the developer's end. Identification of these symptoms (code smells) that affects quality is an important aspect of software maintenance. Therefore, this paper targets identifying two key code smells, namely Long Method and Long Parameter List. The presence of these code smells directly affects the understandability and reusability of the underlying code. The proposed Long Method code smell detection technique depends on four main criteria, the size aspect of the method, Cyclomatic complexity of the method, functional relatedness of the method, and the semantic relatedness among different statements of the method. The proposed functional relatedness aspect at the method level is based on the idea of usage patterns present in the method. These usage patterns help in predicting functionality of the method and are a direct indicator of the fact whether the method is uni- functional (performs a single task) or multifunctional (performs more than one task). The proposed semantic relatedness is based on the tokens extracted at the method level and represents the importance of semantic (underlying concept) aspects at the method level. The proposed approach for Long Parameter List smell detection is also based on two important aspects, namely the size of the parameter list and the complexity of the data types used in the parameter list. The proposed approaches of this paper are experimentally validated and tested against state of the art existing approaches/ tools. The obtained experimental results point to the accuracy and relevance of the proposed approaches.

Index Terms*: Code Smell, Long Method, Long Parameter List, Semantic, Functional, Cyclomatic Complexity.

I. INTRODUCTION

In IT industry, long-term software engineering activities on a software system make its underlying architecture starts deviating from its original structure. In other words, the quality of a software system starts degrading. The symptoms in source code that points to the problems that resulted in degraded quality are termed as code smells. The presence of various kinds of code smells directly affects the maintainability and reusability of the software system [6; 7]. In order to avoid such maintainability and reusability problems, the software system is regularly analyzed, code smells are identified, and the underlying quality (measured in terms of cohesion and coupling) of the system is enhanced by performing refactoring operation [8]. Refactoring is a software restructuring approach that helps in improving the quality without affecting its functionality and preserving its observable behavior [5].

God Class, Feature Envy, Long Method, Data Class are one of the important relevant code smells that gathers the attention of the maintainer team [24]. Therefore, this paper targets two interrelated and relevant (that should be handled with higher priority) code smells namely Long Method (LM) and Long Parameter List (LPL) that directly affect the understandability and maintainability of the software system. Existing direct key approaches and techniques for detecting these code smells include [11; 29; 21; 30; 13; 32]. The approach by [29; 30; 13; 32] is based on the combination of trivial size and complexity metrics only. Moreover, these approaches impose a constant threshold to different metric values. However, to the best of author's knowledge, these constant threshold values are not always feasible and adequate. [11] Proposes a regression-based mathematical model called BLR (Binary Logistic Regression) for detecting LM code smell. However, the downside of this approach is that application of this approach requires an initial calibration phase, which is expert opinion based. This imposes a restriction on the effectiveness of this proposed approach. As far as the approaches that detect the LPL are static in nature and they consider total parameter count as the only aspect for identifying LPL code smell [12]. However, it is our belief that the complexity of the data types assigned to these different parameters also directly affects the understandability at the developer's end.

The restrictions of various existing approaches such as the use of very trivial software metrics and dependence of the expert opinion are removed in the approach proposed in this paper. The proposed approach of this paper is free from these restrictions and considers the semantics of the underlying source- code as important aspects that must be considered while detecting LM and LPL code smells. This paper proposes three new criteria/ metrics, namely Functional Relatedness within a method (cohesion of the method), Semantic Relatedness among different statements of a method, and the complexity of different data types used in the parameter list of a method. These metrics help in capturing more dynamic aspects of the software system and

Revised Manuscript Received on March 15, 2020.

Randeep Singh, Research Scholar, Department of Computer Science & Engineering, M. M. Engineering College, M. M. (Deemed to be University) Mullana, Ambala, Haryana, India

Amit Bindal, Associate Professor, Department of Computer Science & Engineering, M. M. Engineering College, M. M. (Deemed to be University) Mullana, Ambala, Haryana, India

Ashok Kumar, Ex-Professor, Department of Computer Science & Engineering, M. M. Engineering College, M. M. (Deemed to be University) Mullana, Ambala, Haryana, India.

thus helps in improving the code smell detection results.
The rest of this paper is structured as follows: Section-2 presents the literature survey; section-3 explains the proposed approach of this paper. Section-4 presents the experimental setup details. Section-5 provides the obtained results as part of experimentation and discusses those obtained results. Section-6 gives conclusion of the proposed approach of this paper and presents future possible directions for researchers.

II. LITERATURE SURVEY

Existing approaches, techniques, and tools for the Long Method and Long Parameter List code smell are helpful in providing future work directions to any researcher. Therefore, this section summarizes the key researches that are already carried out in this direction. The Long Method and Long Parameter List code smells are actively studied in literature. Code smells like long method and long parameter list being a dynamic concept for researchers throughout the globe and thus different verifiable studies have been done to get perception into the matter.
[17] proposed a technique which is textual-based which is used for detecting long method code smells, named as TACO (Textual Analysis for Code smell detection), and the result has been assessed on three Java projects which indicate that this approach is able to detect between 50% and 77% of the smell with an accuracy ranging between 63% and 67%. [21] Used size and cohesion metrics to find the occurrences of long method code smell, through a method based on case study on java open-source. The results predict that one size and four cohesion metrics resolved the long method bad smell, with a higher precision as compared to the previous studies. [30] proposed an approach (coincide by a tool) that aims at identifying source code smells that work jointly to provide specific functionality. The results of the proposed technique have been validated both in an industrial and an open-source. The approach works better compared to previous ones. [33] Performs an empirical validation of the ability of four cohesion, two coupling, and two size metrics to predict the existence of a long method smell. Eduardo Fernandes et al. proposed a detailed study of four detection tools to find Long Method smell [25]. The author uses three metrics for comparison named agreement, recall, and precision in addition to two software systems based on qualitative and quantitative data provides redundant results for the same long method smell. Kanita et al. proposed a machine learning approach and obtained a result with 99.76 % a high accuracy result for the long method which help the developers to develop software with high quality [35]. Amandeep Kaur in [27] proposed a new technique SVMCSD for long method detection based on the support vector machine learning technique validated on two Open source projects detects more occurrences of bad smell with higher accuracy. Malhotra et al. in [19] combined Chidamber and Kemerer metric and proposed a new metric named a QDIR (Quality Depreciation Index Rule) to identify the affected classes and results shows 80% of the quality of the code improved by using the refactoring techniques and rest of the 20% by using the design metrics. Panita Meananeatra [12] proposed a software metrics, which is defined by data flow and the control flow graphs to select an appropriate refactoring technique to remove long method code smell by computing metrics, candidate refactoring, and maintainability with identification of refactoring. Songsakdi Rongviriyapanish [20] proposes an algorithm to find long method bad smell by using "Extract Method" refactoring by checking the workability and correctness by using an experiment on the algorithm. The authors in [34] proposed a cohesion improvement approach using frequent usage pattern concept. Ahmad Tahmid [22] analyzing the code smells clusters by detection of code smell by using tools such as JUNIT, source code architecture, graph generation and finally find the code smell based on size, number and connection behavior. Sandhya Tarwani [23] proposed a technique with the help of algorithm known as greedy algorithm that evaluates the sequence of refactoring along with the best-refactoring technique to remove code smell that further increase the quality and maintainability of the software. Limei Yang [10] proposed an approach known as AutoMeD that identifies and extracts bad smells in long methods without changing the behavior. The selection of code which code is to be extracted is totally based on finding code blocks that are breakup by blank lines. Fabio Palomba [26] proposed that co-occurrences of the different code smell represent by an approach known as association rule mining that finds the frequent relationship between the code dataset. Fabio Palomba [5] proposed a mixed-method based on qualitative and quantitative study of 117 releases that are taken from the source code from 9 open-source software systems to find developers view on relationship between code smells between projects which are helpful in finding the social and technical issues in between the software development company.

III. PROPOSED METHODOLOGY

The section of this paper gives details about the approach, which is proposed in order to detect two code smells, namely Long Method and Long parameter list.

A. Long Method Code Smell Detection

In the proposed approach, the Long Method (LM) code smell in an object-oriented Java software system is detected based on multiple characteristics of the underlying method of a class. These important characteristics include 1) size of the method, 2) functional relatedness (aka cohesion) factor of the method, 3) Cyclomatic complexity of the method, and 4) semantic relatedness factor of the method. The importance and the details of these

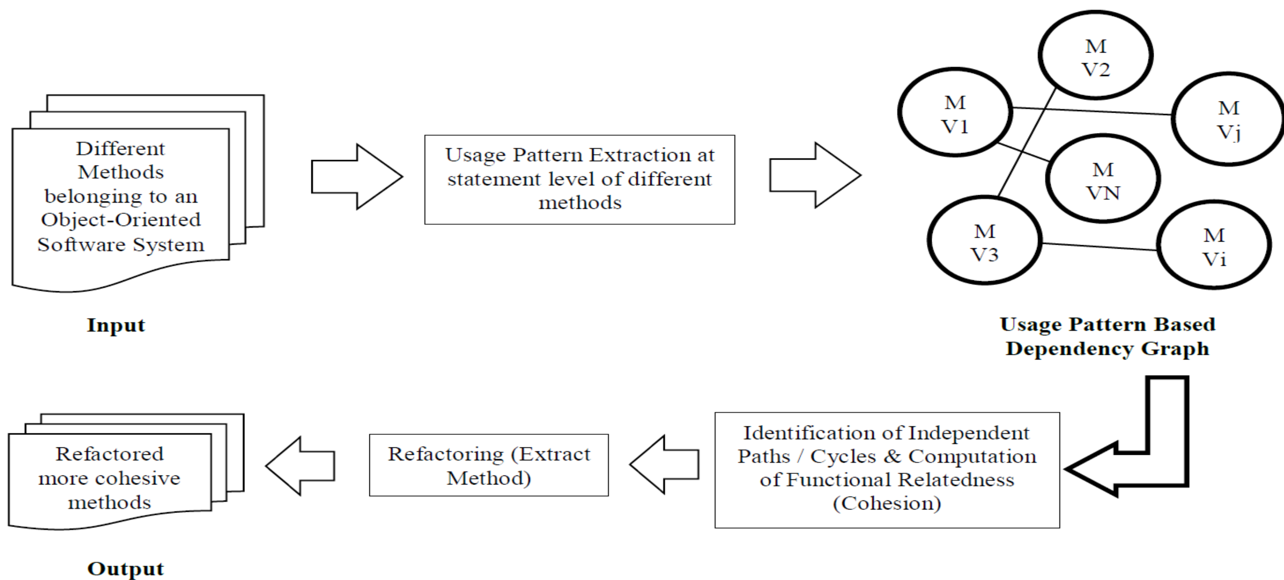

Figure-1: Proposed Usage Pattern-based approach for identifying Long Method code smell.

different characteristics for detecting LM code smell are further explained in following sub-sections:

1. Size as a factor for detecting LM Code Smell

The size is an important parameter that is useful for estimating the length characteristic of a long method in a software system. Size is a direct indicator of the understandability of the underlying method present in the underlying software. In software engineering, the Lines of Code (LOC) is a well-known software metric that is widely used in the literature [9]. This paper considers the same metric to estimate the underlying length of a given method belonging to a class in a software system. This paper considers utilizing the existing LOC metric because of the fact that this metric is well tested by the researchers.

2. Functional Relatedness as a factor for detecting LM Code Smell

In a well-designed object-oriented software system's architecture, a method is supposed to carry out only a single function in order to be easily manageable and understandable. Any method that is implementing multiple functionalities is a direct indicator of the presence of a long method (LM) code smell in the underlying software system. Therefore, this paper considers it worth presenting a new technique for measuring the functional relatedness at the method level. This measurement acts as a key criterion for detecting LM code smell when used with other characteristics used for detecting the code smell.

The proposed functional relatedness measurement approach is based on the extension of the idea of *Usage Patterns* as previously proposed by us in [36; 37]. The concept of usage pattern in software engineering is used to represent the usage behavior present at different levels such as module/ class/ method/ statement. Here, it is important to note that the usage behavior at a different level in software is meant for different member variables associated with the software. In the LM code smell detection approach of this paper, the usage pattern is identified/ represented at statement level that belongs to a method in a class. The idea here is to utilize the concept of usage pattern in determining the functional relatedness/ cohesion among different statements of a method. Based on this functional relatedness score, the possibility of LM code smell is identified in the underlying source code of the method. The basic proposed idea used to determine the functional relatedness among different statements of a method and hence identify the possibility of LM code smell is depicted in figure-1.

The proposed approach in figure-1 takes the source code of the software as input, and the elementary unit of processing is the method belonging to different classes/ interfaces. First, the proposed approach identifies the usage pattern for different member variables belonging to the

software system present at the statement level of a method. Here, it is important to note that the identified usage patterns include both direct as well as indirect usage behaviors present at statement level. The direct usage pattern includes different member variables that are directly used/ accessed by the corresponding statement. Similarly, the indirect usage pattern includes all those member variables of the system that are used indirectly (in object-oriented paradigm through other method calls). The output of this step is the set of member variables that are directly or indirectly accessed by the corresponding statement belonging to the method. The usage patterns thus identified are further used to differentiate between the functional related and unrelated statements present inside a method.

In the second step of the proposed approach, the identified usage patterns at the statement level are represented in the form of undirected graph $G = (V, E)$. Here, V is the set of vertices

and $|V| = |(MV_1, MV_2, \ldots, MV_N)|$. Similarly, E is the set of edges that represents the usage pattern based dependencies at the statement level. The graph representation is used to visualize the usage pattern based dependencies among different statements of a method. In the undirected graph, the nodes represent different member variables belonging to the system (in figure-1 different member variables of the system are shown

as $MV_1, MV_2, \dots, MV_N$. Similarly, different edges in the undirected graph represent usage pattern based dependencies among different member variables at the statement level. Here, for the ith statement of the method whose usage pattern set is of size n, the total number of edges in the usage pattern based dependency graph is given by total number of pairwise combinations possible for the given usage pattern set and is mathematically expressed as $|E| = (n\ 2)$. The third step of the proposed approach in figure-1 determines the total number of independent paths/ cycles T_{IPC} available in the constructed dependency graph and computes the functional relatedness (cohesion) among different statements of a method. Based on this determination, the functional relatedness at method level (overall cohesion) is computed using the following formula:

$$FunctionalRelatedness = Cohesion_{FR} = \left(1 - \frac{T_{IPC}}{N}\right)$$

Here, N is the total number of unique member variables belonging to the software system and the value of the $Cohesion_{FR}$ metric varies in the range[0,1]. A lower value of this metric denotes the presence of more non-functional statements in a given method and vice-versa. Lower values denote the possibility of presence of LM code smell symptoms in the source code of system.

The final step of the approach depicted in Figure-1 performs the refactoring of the source code of the underlying method of a class by performing extract method refactoring. In the proposed approach, the computed functional relatedness score along with the identified independent path/ circle information is utilized for performing extract method opportunities. The aim of the extract method refactoring is to convert any multi-functional non-cohesive method to various multiple uni-functional methods. The various algorithmic steps that are followed in order to perform the extract method refactoring for a method M_i are shown in figure-2 below:

```
Extract_Method_Refactoring (M_i)
Input:
  1. T_IPC - Total Independent Paths/ Cycles list that are identified.
  2. Usage Pattern sets of different statements belonging to a method.
  3. MV_1, MV_2, ..., MV_N - List of different member variables belonging to the system.

Output:
  1. Set of refactored methods.

Algorithmic Steps:
  1. Method_Number = 1
  2. For I = 1 to T_IPC
  3. {
  4.   UPS = Determine the set of member variables present in the I^th independent path.
  5.   PARAMETER_LIST = Parameters that are used in statements whose usage pattern is UPS.
  6.   Move all statements of the method whose usage pattern set is UPS in another method whose method name is given by string "Method+Method_Number".
  7.   Parameter List of the extracted method "Method+Method_Number" is given by PARAMETER_LIST.
  8.   Method_Number++.
  9. }
```

Figure-2: Pseudocode for performing extract method refactoring.

3. Cyclomatic Complexity as a factor for detecting LM Code Smell

Another key characteristic of the LM code smell is the underlying complexity of the underlying source code. The complexity of a method is having a direct relationship with the maintainability and modifiability of the underlying method. Therefore, the complexity is one of the important key characteristics for detecting LM code smell. Hence, this paper considers measuring underlying complexity at the method level during the proposed LM code smell detection approach. McCabe Cyclomatic Complexity (CC) is one of the most widely used software quality metrics that is utilized to measure the underlying code complexity and hence its understandability [1]. The LM code smell detection approach of this paper uses the same quality metric to estimate the complexity of the underlying method. The CC quality metric represents the underlying source code of a method as a control flow graph using the conditional or decision points present in the source code and ultimately measures the total number of linearly independent paths in the control graph. Mathematically, the CC metric is given by the following equations:

$$CC = E - N + 2$$

Here, E is the total number of edges in the control graph and N is the total number of nodes in the same control graph of the method.

Further, it is important to note here that the proposed functional relatedness/ cohesion measurement scheme of this paper is very different from the concept measured using the CC metric. This is because the CC metric is based on various decision points available in the method and the proposed functional relatedness is based on the usage pattern for different member variables in the software system.

4. Semantic Relatedness as a factor for detecting LM Code Smell

For a multi-functional method in a software system (a method that is affected by LM code smell), the semantic similarity among the underlying contents is another criterion that can play a key role in the correct identification of the LM code smell. In a multi-functional method, the semantic similarity among different statements of the method varies a lot as compared to uni-functional methods. Therefore, the proposed approach of this paper also considers semantic similarity as another important parameter for detecting LM code smell. The proposed semantic relatedness measure is based on the tokens extracted from different statements of the method and utilizes conceptual similarity among different tokens using open multilingual corpus called WordNet[1] [4]. In WordNet, different extracted tokens (which can be nouns, verbs, adjectives, and adverbs) are represented by a synset, which denotes a concept or a sense of a group of terms and is ultimately organized into different taxonomic hierarchies. The semantic relatedness among different extracted tokens/words is measured in literature

[1] http://www.nltk.org/howto/wordnet.html

using various similarity metrics [28; 16]. This paper considers measuring pairwise semantic relatedness among different tokens using a path-based semantic similarity metric. Such metrics measure semantic relatedness based on two factors, namely the path length between underlying concepts represented by two tokens and the depth of each of these concepts in the taxonomic hierarchies. Wu et al. proposed one of path-based metrics and it is mathematically expressed as follows [3]:

$$Semantic_Similarity\ (C_1, C_2) = \frac{2 * depth(C_1, C_2)}{len(C_1, C_2) + 2 * depth(C_1, C_2)}$$

Here, the expression $len(C_1, C_2)$ denotes the shortest path distance between any two concepts C_1 and C_2. Similarly, the expression $depth(C_1, C_2)$ represent the depth of two concepts measured in terms of Lowest Common Subsumer (LCS) of underlying concepts C_1 and C_2. The LCS is defined as the common parent of C_1 and C_2 in the taxonomic hierarchies. In the above expression, the depth of LCS represents the distance between the root and the LCS node in the graph. The value of this metric varies between 0 and 1 with 1 as the highest semantic relatedness measure between any two tokens.

Finally, in the proposed approach, the semantic relatedness at the kth method level say M_k is measured using the following equation:

$$Semantic\ Relatedness\ (M_k) = SR(M_k) = Average(\sum_{i=1}^{t} \sum_{j=1}^{t} Semantic_Similarity(C_i, C_j))$$

This proposed equation computes the semantic relatedness by averaging the pairwise individual similarity score between different tokens identified from the statements of the method M_k. The individual tokens are identified by extracting elementary words present in the method source code. These high quality tokens (elementary concepts) are extracted after performing operations such as stopwords removal and porter stemming as done in [31].

Detection of LM Code Smell

On the basis of the four key characteristics of LM code smell (as discussed above), the proposed approach detects the LM code smell using the following formula:

$$Long\ Method = (LOC > \alpha) \wedge (Cohesion_{FR} > \beta) \wedge (CC > \gamma) \wedge (SR > \delta)$$

Here, the variables $\alpha, \beta, \gamma, and\ \delta$ are used as different thresholds above which a given method has the probability of being a Long Method and thus affected with the LM code smell. Moreover, these parameters help us denote the user knowledge during LM code smell detection if the user decides their values.

B. Long Parameter List Code Smell Detection

A large parameter list is generally a symbol of a method that is doing too much of functionality and also is difficult to understand and change at the programmer end. Most of the existing approaches for identifying Long Parameter List (LPL) in literature are confined to only counting the total number of parameters used with a method. However, the authors of this paper are of the opinion that the complexity of individual parameters also plays an important role in the understandability at the user end and thus LPL code smell. Therefore, it should also be considered as a key characteristics for the detection of LPL code smell. Based on this idea, the LPL code smell detection technique as proposed in this paper divides different parameter data types into various categories that denote the complexity in terms of understandability at the programmer end. Table- 1 denotes this categorization and assigns different weights assigned to them that ultimately denote the understandability complexity at the developer's end. These weights are determined in adhoc manner and simply denotes the understandability efforts required at developer's end based on the data type used for the parameter. Lower the value, minimum will be the understandability issues and vice versa. These weights increases gradually as the complexity of the data type increases.

Table-1: Data Types categorization and assigned weights.

	Data Type Categories			
	Simple	**Medium**	**Complex**	**Highly Complex**
Assigned Weight	0.10	0.40	.70	1.00

The proposed approach first computes the understandability complexity for a given method say M_i. The formula used for this purpose is as shown below:

$$Complexity(M_i) = \frac{\sum_{k=1}^{N} DataTypeComplexity(k)}{N}$$

The function $DataTypeComplexity(k)$ returns the complexity of the kth data type and N is the total number of parameters that belongs to the method M_i.

Finally, the LPL code smell is detected using the following criteria as represented in the below equation:

$$LPL\ (M_i) = Complexity(M_i) > \vartheta\ OR\ ParameterCount > \psi$$

Here, ϑ, ψ are the lower bound above which the method M_i possess LPL code smell and is generally developer-specific in order to denote the expert knowledge. $ParameterCount$ is the function that returns the total number of parameters in the method M_i.

IV. EXPERIMENTAL SETUP AND PLANNING

The aim of this paper is to detect the LM and LPL code smells in an object-oriented software system. The proposed approach for detecting these code smells needs to be evaluated and validated experimentally. Therefore, this section gives details about the design of the experimental study performed for this purpose. This section is further divided into following subsections:

Experimental Planning

In order to systematically validate and evaluate the proposed approach, this paper follows the GQM (Goal-Question-Metric) approach [2]. In this approach, first of all, different goals are formulated and then different questions are clarified. Finally, in order to answer the formulated research questions, different data set are considered. By mapping the experimental outcomes and the set goals, we

are able to clarify and validate the proposed approach. The considered goal in this paper is the ability of the proposed approach in detecting LM and LPL code smells.

Research Questions

As the proposed approach of this paper detects LM and LPL code smells available in the source code of the system. Therefore, the following research questions are formulated and answered in order to validate the proposed approach in this paper:

RQ1: Is the proposed efficient enough to improve the quality of the underlying software system? The aim of this considered research question is to test the effectiveness and hence the quality of the proposed approach of this paper. This paper considers evaluating the proposed approach using the well-known *TurboMQ* metric that is widely being used for evaluating modularization of the software system [38; 18]. This metric is specially selected because of the fact that the LM code smell is generally an indication of the fact that the underlying method contains multiple functionalities and hence lower underlying cohesion of the software system. Removing LM code smell must result in the improving of the underlying cohesion of the software system.

Table-2: Considered Data Sets for experiment verification.

S.No.	Software Name	Version	Size (KLOC)	# Classes	Description
1.	JGraphT	0.9.0	14.18	218	A Java library for graph-based data structures and algorithms
2.	CheckStyle	6.4.1	60.00	399	A code analysis tool that helps in adhering to Java coding standards while development
3.	Apache Jena	2.12.1	84.10	697	A Java framework that provides support for building semantic web applications
4.	MobileMedia	Version 7	32.16	55	A Java software for manipulating an image, video, and audio
5.	JUnit	4.0	264.56	984	A Java testing framework
6.	Quartz	2.1.7	26.80	176	An Apache-licensed Java-based job scheduling library

RQ2: How does the accuracy of the proposed approach of this paper vary as compared with the state of the art approaches? This research question is formulated in order to evaluate the accuracy of the proposed approach of this paper against state of the art approach present in the literature for the purpose of detecting LM code smell. To answer this formulated research question, this paper considers evaluating the approaches being compared using the well-known Information Retrieval (IR) metric called F-Measure [15].

Data Set Selection and Unit of Analysis

For any approach to be widely applicable, it is necessary that it must be experimentally validated against a wide array of projects belonging to different domains and are of different sizes. Therefore, this paper considers Open Source Java Softwares that are publically available, are of varied sizes (measured as KLOC), and finally belongs to separate domains. Table-2 in this paper depicts various software systems considered for the experimental evaluation of the approach shown in Figure-1. These software are of different sizes, domains, and are taken from GitHub repository or are publically available on the internet. The second to fifth columns give various key details such as name of the software, its version, size in KLOC, total number of classes, and a short description that gives detail about the domain of the considered software system.

Data Collection and Analysis

The dataset used for the experimental purpose is shown in Table-1. The basic information such as size, the total number of classes, etc. is determined in this paper using a well-known standard software engineering tool called *LocMetrics*[2]. The computation of functional relatedness based on the usage patterns is done using a tool designed by authors. The information about the CC metric value is gathered using another state of the art tool called *JArchitect*[3]. The computation of semantic relatedness is performed using an open-source Java API called *WNetSS API*[4] in combination with self designed tool that parses the source code of the software and extracts different tokens at method level.

Besides this, the proposed approach is evaluated (answer to RQ1) using TurboMQ metric that measures and determines the balance between the intra-connectivity and inter-connectivity. Finally, in order to compare the proposed approach (to answer RQ2) with another approach in literature, the F-Measure metric is used that is based on comparison between the actual and observed LM and LPL code smell list. The actual code smell list is expert-based and is generally determined using the developer's knowledge (expert opinion). In this paper, in order to minimize any kind of user biasness, the actual code smell list is obtained based on three key automatic tools, namely JDeodorant[5], CheckStyle[6], and PMD[7]. Each of the software

[2] http://www.locmetrics.com/
[3] https://www.jarchitect.com/Metrics
[4] http://wnetss-api.smr-team.org/
[5] https://github.com/tsantalis/JDeodorant
[6] https://github.com/checkstyle/checkstyle
[7] https://pmd.github.io/

considered under the study is first analyzed using each of the considered tools and finally the obtained code smell list of each of these tools are merged to have common list of LM and LPL code smells. This list is ultimately used with the observed code smell list obtained after applying the proposed approach of this paper.

V. RESULTS AND DISCUSSION

In this section of the paper, we present the results obtained after evaluating different software systems considered in Table-2 above using the approach depicted in Figure-1 and ultimately presents an interpretation of the obtained results. The result and interpretation (conclusion) are divided into two sub-sections that correspond to two formulated research questions of this paper.

RQ1: Is the proposed efficient enough to improve the quality of the underlying software system? Based on the experimentation performed on the considered dataset, the total number of LM and LPL code smells observed are shown in Table-3. After identifying traces of long method code smell, it is mitigated by applying extract method refactoring. After performing refactoring, it is important to determine the quality of the software system. This is done by measuring the quality of the software using the TurboMQ metric in two scenarios. In first considered scenario, we measure the software system's quality before applying extract method refactoring (i.e. quality of the original software system). In second scenario, we measure the quality of same software system after identifying LM code smell and applying extract method refactoring. For an efficient approach, the refactored system after applying proposed approach must result in improved underlying quality. Table- 3 shows the value of TurboMQ metric above considered two scenarios. From Table- 3, it is clear that the overall quality of different software systems varies from a minimum of 21% to a maximum of 38%. This improvement denotes a significant enhancement in the underlying quality of the software system. Moreover, from Figure-3 that plots different code smells per software, it can be clearly observed that different software is affected more with LM code smell as compared to LPL code smell.

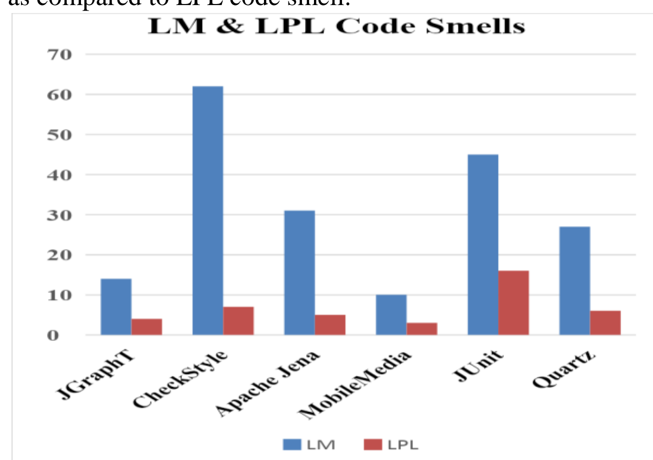

Figure-3: LM & LPL Code Smells Detected in different Software Systems.

Table-3: Obtained Results for LM and LPL code smell.

S.No.	Software Name	# LM	# LPL
1.	JGraphT	14	4
2.	CheckStyle	62	7
3.	Apache Jena	31	5
4.	MobileMedia	10	3
5.	JUnit	45	16
6.	Quartz	27	6

Table-4: Observed TurboMQ metric values.

S.No.	Software Name	TurboMQ Metric Value		Percentage Improvement
		Before Applying the Proposed Approach	After Applying the Proposed Approach	
1.	JGraphT	1.24	1.64	24%
2.	CheckStyle	0.97	1.23	21%
3.	Apache Jena	3.42	4.46	23%
4.	MobileMedia	0.65	0.95	32%
5.	JUnit	2.12	2.85	26%
6.	Quartz	0.45	0.72	38%

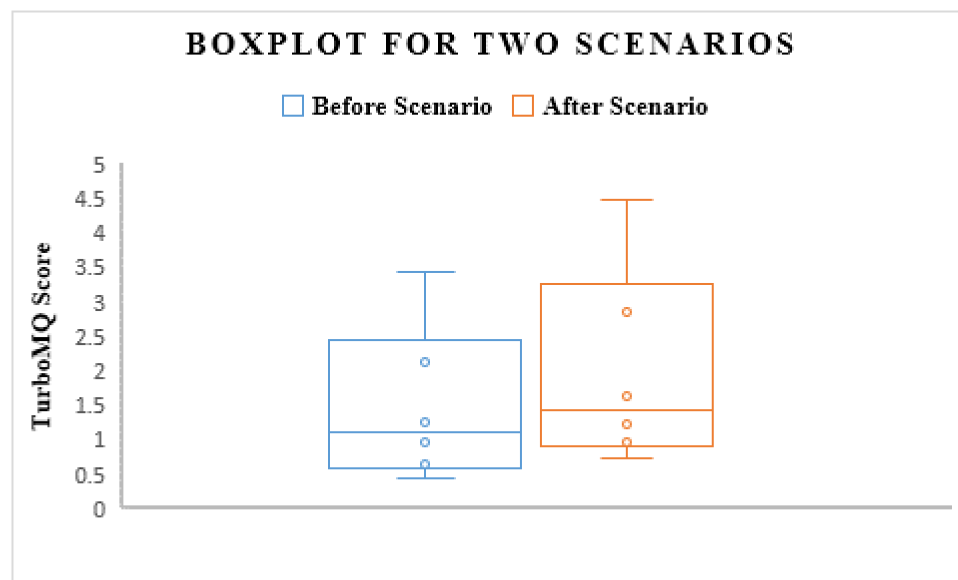

Figure-4: Boxplots of TurboMQ in two scenarios, namely before and after.

Figure-4 shows a boxplot of the quality of the software systems measured in terms of the TurboMQ metric score. It plots two scenarios- Before and After applying the proposed approach of this paper. From the plot, it is clear that the overall quality of software systems after applying our proposed approach significantly improves the quality (least coupling and higher cohesion). This improved quality is an indicator of the fact that the underlying different methods are almost uni- functional (nil or least traces of LM code smell) and are generally doing only a single function.

RQ2: How does the accuracy of the proposed approach of this paper vary as compared with the state of the art approaches?

This research question considers performing comparison between our proposed approach and the state of the art technique proposed in the literature by [29] that identifies LM code smell. Their proposed approach is based on two metrics namely size (LOC) and complexity (CC). This approach is chosen, as it is recent and directly related to LM code smell detection. As part of comparison, first, both the approaches are applied to the studied software systems and the corresponding LM code smells are recorded as observed list say L_1. Now, in order to make fair comparison, we obtained an authentic (actual) list of LM code smells in different software. Here, we are having two choices, namely 1) to obtained LM code smell list based on expert (developer's knowledge), and 2) to obtain the same list based on well-tested and widely used tools, namely JDeodorant[8], PMD[9], and CheckStyle[10]. In order to get actual LM code smell list, each of the studied systems is analyzed by these tools and their individual list is ultimately combined to get final actual LM code smell list say L_2. As stated above, the final comparison between two obtained lists L_1 and L_2 is done using F-Measure metric. Table-4 shows the comparison results between two approaches.

[8] http://http://www.jdeodorant.com/
[9] http://pmd.sourceforge.net/
[10] http://checkstyle.sourceforge.net/

Table-5: Comparison Results using the F-Measure metric.

S.No.	Software Name	F-Measure Metric Value			
		Long Method (LM) Code Smell		Long Parameter List (LPL) Code Smell	
		Proposed Approach by [29]	Our Proposed Approach	Rival Approach	Our Proposed Approach
1.	JGraphT	0.67	0.83	0.73	0.84
2.	CheckStyle	0.72	0.92	0.84	0.93
3.	Apache Jena	0.64	0.88	0.71	0.79
4.	MobileMedia	0.78	0.92	0.85	0.91
5.	JUnit	0.69	0.91	0.74	0.85
6.	Quartz	0.74	0.89	0.81	0.90

From Table-4, it is clear that the F-Measure metric for the approach proposed in this paper is having a higher value score for each of the studied software systems. This higher value indicates that the proposed approach provides higher accuracy (precision) and sensitivity (recall) measured using F-Measure metric. Lower values in case of approach given by [29] indicate the fact that the size and complexity metrics alone cannot identify all the LM code smells in a software system.

To the best of the author's knowledge, the LPL approaches in literature are based only on the counting of total number of parameters in the method [14]. This paper considers comparing LPL code smell detection approach of this paper against PMD and CheckStyle tool which considers the threshold value as 10 and 7 respectively. From the results shown in Table-4, it is clear that our proposed LPL code smell detection approach outperforms slightly. This is due to the fact that parameter count directly affects the understandability of the code, however, considering the semantics of the data types of those parameters is also necessary. The improved results that are obtained after rigorous experimentation confirm the validity of the proposed approach in this paper.

VI. CONCLUSIONS AND FUTURE WORKS

This paper targets detecting two important and interrelated code smells, namely Long Method and Long Parameter List. The LM code smell is detected based on four criteria, namely Size (LOC Metric), Complexity (CC Metric), and two newly proposed metrics that measure functional relatedness based Cohesion ($Cohesion_{FR}$ Metric), and semantic relatedness $SR\ metric$ among different statements of a method. The proposed approach also gives direct importance to user feedbacks (developer's knowledge) in terms of threshold limits to these used four metrics value. Similarly, the LPL code smell is detected based on two criteria, namely the size (count of total number of parameters present) of parameter list, and the semantics of different datatypes assigned to these different parameters. Here, the semantics of parameter types is given due importance and the proposed approach also considers user feedback in terms of threshold to these metric values. Both the proposed approaches are experimentally validated against standard open-source object-oriented Java software systems of different sizes and domains. The obtained results show a significant improvement in the quality of the underlying software systems.

The future works related to this paper are many. First, we are planning to provide an automated tool for the proposed approach for the research community. Secondly, the proposed concept and metrics can be tested for their feasibility in detecting other types of code smells in a software system. Finally but not least, the proposed approach can be used in reusable component identification and the cohesion improvement (software restructuring) of the underlying software system.

REFERENCES

1. T. J. McCabe, "A Complexity Measure," in IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308-320, Dec. 1976.
2. V. Basili, G. Caldiera & D. Rombach, "The Goal Question Metric Approach", Encyclopedia of Software Engineering, John Wiley & Sons, pp. 528-532. 1994.
3. Z. Wu & M. Palmer, "Verb semantics and lexical selection", ACL Proceedings of Annual Meeting on Association for Computational Linguistics, pp. 133—138, 1995.
4. C. Fellbaum, "WordNet: An electronic lexical database", MIT Press, 1998.
5. M. Fowler, "Refactoring: Improving the Design of Existing Code",
6. Boston, MA, USA: Addison-Wesley Longman Publishing Co.,Inc.,1999.
7. M. Fowler, K. Beck, J. Brant, W. Opdyke, & D. Roberts, "Refactoring: Improving the Design of Existing Code", AddisonWesley, 2000.
8. S. McConnell, "Code Complete - A Practical Handbook of Software Construction", 2nd ed., Microsoft Press, 2004.
9. K. Stroggylos & Diomidis Spinellis, "Refactoring–Does It Improve Software Quality", In Proceedings of the 5th International Workshop on Software Quality (WoSQ'07).IEEEComputerSociety,Washington,DC,USA,10–. https://doi.org/10.1109/WOSQ.2007.11,2007.
10. A.L. Timoteo, A. Alvaro, E.S. De Almeida, & S.R. De Lemos Meira, ,"Software metrics: a survey",2008.
11. L. Yang, H. Liu, & Z. Niu, "Identifying fragments to be extracted from long methods", 16th Asia-Pacific Software Engineering Conference, IEEE, 43–49, 2009.
12. S. Bryton, F.B. E. Abreu, & M. Monteiro, "Reducing subjectivity in code smells detection: Experimenting with the long method", In Seventh International Conference on the Quality of Information and Communications Technology, IEEE, pp. 337-342, 2010.
13. P. Meananeatra, S. Rongviriyapanish & T. Apiwattanapong, "Using software metrics to select refactoring for long method bad smell", The 8th Electrical Engineering/ Electronics, Computer, Telecommunications and Information Technology (ECTI) Association of Thailand – Conference, 2011.

14. P. Danphitsanuphan, & T. Suwantada, "Code smell detecting tool and code smell-structure bug relationship", In Spring Congress on Engineering and Technology, IEEE,pp. 1-5, 2012.
15. F.A. Fontana, P. Braione, & M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment" Journal of Object Technology, vol. 11, no. 2, pp. 5-1, 2012.
16. A. Field, "Discovering Statistics using IBM SPSS Statistics", SAGE Publications Ltd, 2013.
17. L. Meng, R. Huang, & J. Gu, "Measuring semantic similarity of word pairs using path and information content. Int. J. Future Gener. Commun. Netw, vol. 7, no. (3), pp. 183-194, 2014.
18. F. Palomba, "Textual Analysis for Code Smell Detection", International Conference on Software Engineering, Florence, 2015.
19. T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, & R. Kroeger,"Comparing software architecture recovery techniques using accurate dependencies", In IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE, Vol. 2, pp. 69-78, May, 2015.
20. R. Malhotra, A. Chug & P. Khosla, "Prioritization of Classes for Refactoring: A Step towards Improvement in Software Quality", Proceedings of the Third International Symposium on Women in Computing and Informatics, pp. 228-234, 2015.
21. S. Rongviriyapanish & N. Karunlanchakorn, "Automatic Code Locations Identification for replacing temporary variable with query method", 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2015.
22. S. Charalampidou, A. Ampatzoglou, & P. Avgeriou, "Size and cohesion metrics as indicators of the long method bad smell: An empirical study", 11th International Conference on Predictive Models and Data Analytics in Software Engineering, New York, 2015.
23. A. Tah mid, N. Nahar & K. Sakib, "Understanding the Evolution of Code Smells by Observing Code Smell Clusters", IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016.
24. S. Tarwani & A. Chug, "Sequencing of Refactoring Techniques by Greedy Algorithm for maximizing Maintainability", International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2016.
25. S.A. Vidal, C. Marcos, & J.A. Diaz-Pace, "An approach to prioritize code smells for refactoring", Automated Software Engineering, vol. 23, no.3,pp. 501-532,2016.
26. E. Fernandes, J. Oliveira, G. Vale, T. Paiva, & E. Figueiredo, "A Review-based Comparative Study of Bad Smell Detection Tools," 20th International Conference on Evaluation and Assessment in Software Engineering, 2016.
27. F. Palomba, R. Oliveto, A. De Lucia, "Investigating Code Smell Co-occurrences using Association Rule Learning: A Replicated Study", IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), 2017.
28. A. Kaur, S. Jain & S. Goel, "A Support Vector Machine Based Approach for Code Smell Detection", International Conference on Machine Learning and Data Science (MLDS), 2017.
29. X.G. Zhang, S.Q. Sun, & K.J. Zhang, "A novel comprehensive approach for estimating concept semantic similarity in wordnet. arXiv preprint arXiv:1703.01726",2017.
30. D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, & A. Chavez, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects", In Proceedings of the 11th Joint Meeting on Foundations of Software Engineering ,ACM, pp. 465-475, August, 2017.
31. S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, & P. Avgeriou, "Identifying Extract Method Refactoring Opportunities Based on Functional Relevance", IEEE Transactions on Software Engineering, vol. 43, no. 10, pp. 954–974, 2017.
32. A. Rathee, & J.K.Chhabra, "Software remodularization by estimating structural and conceptual relations among classes and using hierarchical clustering", In International Conference on Advanced Informatics for Computing Research, Springer, Singapore ,pp. 94-106, ,March,2017.
33. S. Vidal, S. Zulliani, C. Marcos, & J. Pace, "Assessing the Refactoring of Brain Methods", ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 27, no.1, 2018.
34. S. Charalampidou, E.M. Arvanitou, A. Ampatzoglou, P. Avgeriou, Chatzigeorgiou & I. Stamelos, "Structural Quality Metrics as Indicators of the Long Method Bad Smell: An Empirical Study", 44th Euromicro Conference on Software Engineering and Advanced Applications, 2018.
35. A. Rathee, & J.K. Chhabra, "Improving cohesion of a software system by performing usage pattern based clustering", Procedia Computer Science, vol. 125, pp. 740-746, 2018.
36. K. K. Hadziabdic, R. Spahic, "Comparison of Machine Learning Methods for Code Smell Detection Using Reduced Features", 3rd International Conference on Computer Science and Engineering (UBMK), 2018.
37. R. Singh & A. Kumar, "To Improve Code Structure by Identifying Move Method Opportunities Using Frequent Usage Patterns in Source-Code", In International Conference on Advanced Informatics for Computing Research, Springer, Singapore, pp. 320-330, July, 2018.
38. R. Singh, A. Bindal, & A. Kumar, "A User Feedback Centric Approach for Detecting and Mitigating God Class Code Smell Using Frequent Usage Patterns" ,Journal of Communications Software and Systems, vol. 15, no. 3, 2019.
39. A. Rathee & J.K. Chhabra, "A multi-objective search based approach to identify reusable software components", Journal of Computer Languages, vol. 52, pp. 26-43, 2019.

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 PhD 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.