J Inf Process Syst, Vol.16, No.4, pp.915~934, August 2020
https://doi.org/10.3745/JIPS.04.0184

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

A Systematic Literature Survey of Software Metrics, Code Smells and Refactoring Techniques

Mansi Agnihotri* and Anuradha Chug*

Abstract

Software refactoring is a process to restructure an existing software code while keeping its external behavior the same. Currently, various refactoring techniques are being used to develop more readable and less complex codes by improving the non-functional attributes of software. Refactoring can further improve code maintainability by applying various techniques to the source code, which in turn preserves the behavior of code. Refactoring facilitates bug removal and extends the capabilities of the program. In this paper, an exhaustive review is conducted regarding bad smells present in source code, applications of specific refactoring methods to remove that bad smell and its effect on software quality. A total of 68 studies belonging to 32 journals, 31 conferences, and 5 other sources that were published between the years 2001 and 2019 were shortlisted. The studies were analyzed based on of bad smells identified, refactoring techniques used, and their effects on software metrics. We found that "long method", "feature envy", and "data class" bad smells were identified or corrected in the majority of studies. "Feature envy" smell was detected in 36.66% of the total shortlisted studies. Extract class refactoring approach was used in 38.77% of the total studies, followed by the move method and extract method techniques that were used in 34.69% and 30.61% of the total studies, respectively. The effects of refactoring on complexity and coupling metrics of software were also analyzed in the majority of studies, i.e., 29 studies each. Interestingly, the majority of selected studies (41%) used large open source datasets written in Java language instead of proprietary software. At the end, this study provides future guidelines for conducting research in the field of code refactoring.

Keywords

Code Smells, Extract Class Refactoring, Feature Envy Bad Smell, Refactoring Techniques, Software Maintenance, Software Metrics

1. Introduction

One of the imperative properties of software is that it evolves with time. As the software adapts to the new environment, it moves away from its original design. Modification and enhancement of software make the code more complex; and, thus, lowers the quality of the software. All of these factors make the software maintenance phase costly and time-consuming. One of the most expensive activities performed during the d evelopment and evaluation phase of an application is the maintenance of the source code, since changes can certainly occur in software to remain updated, useful, and with high quality [1].

Bad smells develop in a software system when the design of the code component is incorrectly assumed

Manuscript received January 16, 2020; first revision April 23, 2020; accepted April 28, 2020.

Corresponding Author: Mansi Agnihotri (Mansi.15816490019@ipu.ac.in)

* University School of Information, Communication & Technology (USIC&T), Guru Gobind Singh Indraprastha University, New Delhi, India (Mansi.15816490019@ipu.ac.in, anuradha@ipu.ac.in)

or the solution by the developers is improperly designed [2]. A code smell is an exterior symptom indicating a deeper issue in the system [3]. They are generally not bugs and can be technically correct as they indicate some weakness in the design of the software that may lead to failure in the future. Bad smells in code can be detected and corrected using refactoring technique. According to Fowler [3], refactoring is a process that changes a software system to improve its internal structure without modifying the external behavior of the program code. It is a systematic method of minimizing the chances of introducing bugs by simply cleaning the code [3]. With the help of refactoring, bad designs may be converted into well-structured code by reworking on them. In addition, refactoring can significantly improve some of the software's external qualities such as reusability, maintainability, and readability [4]. In the past, several surveys have been conducted in the field of software refactoring [5], code smells [6] and software metrics [7]. To improve software quality, a detailed knowledge is required regarding the most frequently occurring code smells, most used refactoring technique and the software metrics that have a direct influence on them. These review studies conducted by Mens and Tourwe [5], Zhang et al. [6], and Xenos et al. [7] did not provide any holistic comprehensive view putting three perspectives together. In the current study, an attempt has been made to collectively review various studies from all the three perspectives (bad smell, refactoring, and software metrics) so as to provide a better insight to the developers. This study will help the users to gain better information on the three perspectives. Also, the tools used in the shortlisted studies are categorized based on their usage in software refactoring, code smells detection, and software metric calculation. This will further help the users to select the tools as per their requirement.

In the current study, 68 papers on the three perspectives and published between the years 2001 and 2019 were investigated. This current review aims to summarize, analyze and comprehend the studies based on the following aspects:

- Various bad smells identified, analyzed and/or corrected.
- Various refactoring techniques used to correct different kinds of code smell.
- Framework/platform used to apply refactoring techniques.
- Identification of various software metrics used to identify code smells.
- Comparison of various refactoring techniques in terms of their effects on software metrics.

To achieve the target, 8 digital libraries were considerably explored, and 68 studies were identified and incorporated in this review to answer the Research Questions (RQs) that are discussed later in this article. The remaining article is structured as follows: Section 2 presents the review methodology. Section 3 discusses the planning phase of the systematic literature review (SLR) process. Section 4 describes the conducting phase that includes the formulation of selection criteria, removing duplicate studies, etc. Section 5 discusses the results and answers the RQs that were raised while assessing the studies. Section 6 provides the conclusion of the paper followed by future directions, which is provided in Section 7.

2. Review Methodology

The process involved in conducting the review is presented in this section. The SLR here identifies, assesses, and interprets all available research applicable to refactoring techniques and code smells. The review methodology adopted is illustrated in Fig. 1; this methodology is majorly divided into three stages:

planning, conducting, and reporting. In the first stage, i.e., planning stage, the search databases were identified, the RQs were formed, and the relevant papers in the field of study were extracted. In the second stage, i.e., conducting stage, papers were shortlisted after removal of duplicate and irrelevant papers that did not conform to the inclusion criteria. The integration was carried out in the second stage on the basis of the information provided in each selected study. In the final stage, RQ's were answered.

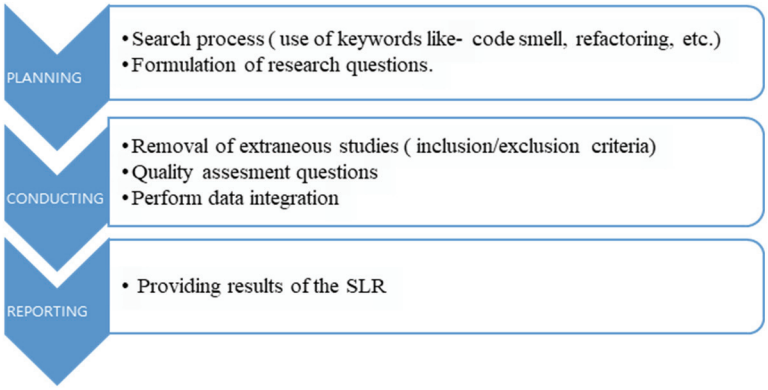

Fig. 1. Review process.

3. Planning Phase for Review

3.1 Search Process

The initial search began with exploring various online databases, electronic journals, and search engines. The search process was divided into two steps. In the first step, databases such as Scopus, Google Scholar, Science Direct, IEEE Xplore, and Springer were identified. Internet resources provided by Guru Gobind Singh Indraprastha University (GGSIPU) were also used. Appropriate search strings such as "code refactoring", "bad smell", "software metrices", "refactoring for bad smells", "refactoring techniques", and "code smell detection" were used to extract relevant studies. The search string that was formed using the search terms is given below:

> (SoftwareRefactoring OR CodeRefactoring OR RefactoringTechniques) AND (BadSmells or CodeSmells) AND (SoftwareMetrices OR MetricSuites OR Maintainability) OR (RefactoringTools OR BadSmell Tools).

Thousands of results were attained in the initial step using the above terms. In the second step, publications belonging to renowned journals, and conferences were selected. In addition, the search was restricted to the years between 2001 to 2019.

3.2 Formulation of Research Questions

A number of RQ's were raised while assessing the shortlisted studies. RQs provide an insight into the past and current trends in the field of code refactoring. RQs were precisely formulated in order to maintain

a proper flow of the study and to avoid deviation while going through the vast amount of information obtained from the selected papers. Table 1 provides the list of RQs for the current review.

Table 1. Research questions

S. No.	Research questions
RQ1	What are the various types of code smells that are detected in the studies?
RQ1.1	What are the technical aspects of most detected code smells?
RQ2	What are the different types of refactoring techniques used/identified in the studies?
RQ2.1	What are the technical aspects of most used refactoring techniques?
RQ3	What are the different types of software metrics used in the studies?
RQ4	What is the data collection procedure (open source, proprietary or sample dataset)?
RQ4.1	Which datasets are most frequently used?
RQ4.2	How are the datasets divided on the basis on size?
RQ4.3	Which is the most used language for the datasets?
RQ5	How are the studies analyzed on the basis of the usage of tools?
RQ5.1	Which are the most used tools in the shortlisted studies?

4. Conducting Phase for Review

The following section includes the activities performed during the conducting phase of SLR. It presents the selection criteria used to filter out the irrelevant studies. In addition, a questionnaire was formed to score the selected studies as per the RQs mentioned in Table 1. Pie charts and line graphs were used to show the bifurcation of the shortlisted studies as per the year and the source of publication.

4.1 Inclusion/Exclusion Criteria (Removal of Extraneous Studies)

Studies were organized in an orderly fashion, and duplicate and irrelevant studies were dropped from the review procedure. Based on the RQs listed in Table 1 the inclusion criteria were set in order to choose those studies that included some content on some aspects of software refactoring or code smells. The papers that focused on refactoring techniques to remove bad smell, improving software quality with the help of refactoring, detection of bad smells, effect of code refactoring on software maintenance, use of software metrics to identify code smells, effect of bad smells on quality attributes, and prediction models that identify opportunities for refactoring, qualified the inclusion criteria.

Inclusion criteria

- Empirical studies that identified different refactoring techniques.
- Empirical studies that identified/corrected code smells.
- Empirical studies that made use of software metrics to indicate the presence of bad smells.
- Empirical studies that proposed refactoring techniques to improve software quality.

Exclusion criteria

- Studies that did not focus on software refactoring or bad smells.
- Studies that could not answer the RQs as specified in Table 1.
- Studies that do not belong to the years between 2001 to 2019.
- Review papers.
- Studies that did not provide any experimental results.

4.2 Quality Assessment Criteria

A questionnaire was formulated to filter out irrelevant papers and provide a further refinement to assessment and selection of relevant studies. Initially, 106 studies were extracted on the basis of search strings using different search engines, out of which 76 studies were selected that qualified the inclusion criteria mentioned in Section 4.1. Table 2 presents the list of questions that were used as the checklist for quality assessment that scored the studies on a scale of 0 to 1.

Table 2. Quality assessment questions

S. No.	Quality assessment questions	Score (0/0.5/1)
Q1	Whether the study defines any goal or objective?	
Q2	Does the study define software refactoring/bad smell?	
Q3	Whether the study clearly defines the independent variables?	
Q4	Whether the dataset sources were indicated?	
Q5	Does the study make use of appropriate tools to perform the implementation?	
Q6	Whether the application of refactoring techniques was properly defined?	
Q7	Whether open-source datasets were used?	
Q8	Does the study specify various threats to validity?	
Q9	Does the study compare various techniques that were applied?	
Q10	Whether appropriate results were provided in the study?	
Q11	Whether the study mentions the limitations and advantages of the research?	
Q12	Whether a proper literature survey was conducted prior to the implementation of the proposed aim?	

4.3 Data Extraction

Data that answered all the RQs were extracted in the conducting phase. A database consisting of different features such as publication year, name of the study, author's name, publication source, objective of the study, results, tools used, techniques applied, etc., was prepared. Majorly, five attributes were selected that answered the RQs and, are listed in Table 3.

All the selected studies were analyzed from various perspectives, and an attempt was made to identify any association between the studies. The data were pictorially illustrated using pie charts, line graphs, bar chart, etc. The visualization technique helps the reader to absorb and interpret the data easily.

Table 3. Quality assessment attributes

Attribute	Research question
Bad smells identified/corrected	RQ1
Refactoring techniques	RQ2
Software metrics	RQ3
Datasets	RQ4, RQ4.1, RQ4.2, RQ4.3
Tools used	RQ5, RQ5.1

4.4 Distribution of Papers

The selected studies were divided into three categories depending upon their source of publication. The sources considered are journals, conferences, and others such as symposiums, book chapters, workshops,

etc. The distribution of studies is demonstrated using a pie chart in Fig. 2. The majority of studies, i.e., 32 studies, belonged to journal publications, followed by 31 studies from conferences, and 5 studies from various other sources.

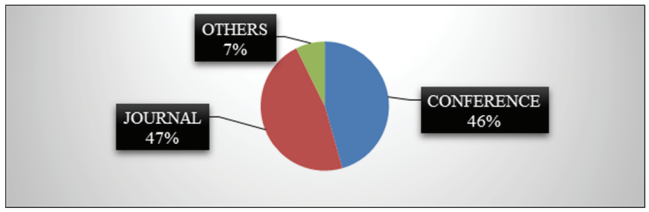

Fig. 2. Distribution of papers as per the source of publication.

4.5 Distribution of Papers as per Year of Publication

The PhD dissertation titled "Program restructuring as an aid to software maintenance" of William Griswold published in 1991 was considered to be the first significant work in the field of refactoring. Since then, multiple studies have covered the concept of refactoring and bad smells and proposed various techniques in a similar field. The shortlisted studies lie between the years 2001 and 2019. Fig. 3 shows the distribution of studies on the basis of years.

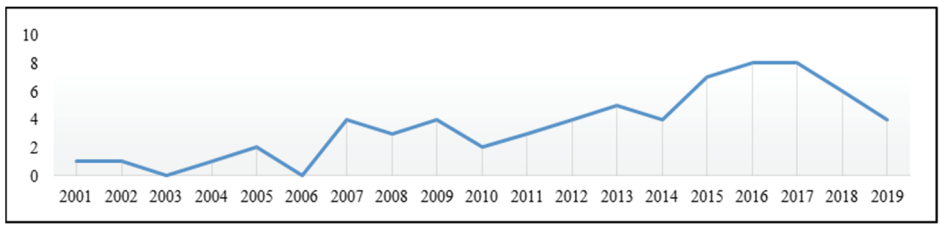

Fig. 3. Year-wise bifurcation of studies.

5. Reporting Phase

This section provides the results obtained from the SLR. Firstly, the quality assessment questions are presented along with the scores given to the selected studies. Then, the answers to the RQs are provided.

5.1 Quality Assessment Questions along with Their Scores

The quality assessment of selected studies was performed by team of 2 members that comprised 1 Assistant Professor and 1 Research Scholar from GGSIPU, Delhi, India. The selected studies were scored as follows:

- 1 mark for the complete qualification of the paper
- 0.5 mark for partial qualification.
- 0 mark for no qualification.

Each study was scored on a scale of 0 to 1, and the total score was calculated by marking them against all the quality assessment questions. The maximum and minimum marks obtained can be 12 and 0. After

Table 4. Scores for quality assessment questions

Ref.	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Total
[1]	1	1	0	1	1	0.5	1	1	1	1	1	0.5	10
[4]	1	1	1	1	0	0	0	0	1	1	1	1	8
[8]	1	1	1	1	0	0	1	1	1	0.5	0.5	1	9
[9]	1	1	1	1	1	0	0	0	1	1	1	1	9
[10]	1	0.5	1	1	1	0	1	0	1	1	0.5	0.5	8.5
[11]	1	1	1	0.5	0.5	1	0	0	1	1	1	0	8
[12]	1	1	1	1	1	0	1	0	0	1	0	0.5	7.5
[13]	1	1	1	1	1	0	1	0	1	1	0.5	0.5	9
[14]	1	1	1	1	1	0	1	1	1	1	0	1	10
[15]	1	1	0.5	1	0	1	1	0	0.5	0.5	1	1	8.5
[16]	1	1	0.5	1	1	0.5	0	1	1	1	1	1	10
[17]	1	1	0.5	1	1	0.5	1	0	1	1	1	0.5	9.5
[18]	1	1	1	1	1	0	1	1	1	1	1	1	11
[19]	1	1	0.5	0	1	1	0	0	0.5	1	0.5	0.5	7
[20]	1	1	1	1	0	0.5	0	0	0.5	0.5	1	1	7.5
[21]	1	1	0.5	1	0	0.5	0	0	1	1	0.5	0.5	7
[22]	1	1	0.5	1	1	1	0	0	0.5	1	0.5	0.5	8
[23]	1	1	0.5	0.5	0	0.5	0	0	1	1	1	1	7.5
[24]	1	1	0.5	0	0	0.5	0	0	1	1	1	1	7
[25]	1	1	0.5	1	0.5	1	1	1	1	1	1	1	11
[26]	1	1	0	1	1	0	0	0	1	1	0	1	7
[27]	1	1	0	1	0	0.5	1	0	0.5	1	1	1	8
[28]	1	1	0.5	1	1	0.5	1	0	1	1	1	1	10
[29]	1	1	0.5	1	1	0	1	0	0	0.5	0.5	0.5	7
[30]	1	1	0.5	1	1	1	1	0	1	1	1	1	10.5
[31]	1	1	0.5	1	0	0	1	0	0.5	0.5	0.5	1	7
[32]	1	1	0	1	1	1	1	0	1	1	1	1	10
[33]	1	1	0	1	1	0.5	1	1	1	1	1	1	10.5
[34]	1	1	0.5	1	1	1	1	0	1	1	0.5	1	10
[35]	1	1	1	1	0	0	1	0	0.5	0.5	0	1	7
[36]	1	1	0	1	0	0.5	1	0	0.5	0.5	0.5	1	7
[37]	1	1	1	1	1	1	1	0	1	1	1	1	11
[38]	1	1	0.5	1	0	0.5	0	1	0.5	0.5	1	0.5	7.5
[39]	1	1	0	1	0	0	1	0	0.5	1	1	1	7.5
[40]	1	1	0	1	0.5	1	1	1	1	1	0.5	1	10
[41]	1	0.5	0	1	0.5	0.5	1	1	1	0.5	1	0	8
[42]	1	1	0	1	0.5	0.5	1	1	1	1	0.5	1	9.5
[43]	1	0.5	0	1	0	0	1	1	1	1	0	1	7.5
[44]	1	1	1	1	0	0	1	0	1	1	0	1	8
[45]	1	1	1	1	0.5	0	0	05	1	0	1	1	8
[46]	1	1	0.5	1	0	0.5	0	0.5	1	1	0.5	1	8
[47]	1	1	0	1	0.5	0.5	1	1	0.5	0.5	0.5	0	7.5
[48]	1	1	0	1	0.5	1	1	0	1	1	0.5	1	9
[49]	1	1	0.5	1	0	0	1	1	1	1	1	1	9.5
[50]	1	1	1	1	1	1	1	1	1	1	1	1	12
[51]	1	1	0.5	1	1	1	1	1	1	1	0.5	0.5	10.5
[52]	1	1	0	1	1	1	1	1	1	1	1	1	11
[53]	1	1	0	1	1	0.5	1	1	1	1	0.5	1	10
[54]	1	1	1	1	1	1	1	1	1	1	1	1	12
[55]	1	1	0	1	0.5	0.5	1	0	0.5	1	0.5	1	8
[56]	1	0.5	1	1	0.5	0	1	1	1	1	1	0.5	9.5
[57]	1	1	0	1	1	0	1	1	1	1	1	1	10
[58]	1	1	0	1	0.5	0.5	1	1	1	0.5	0	1	8.5
[59]	1	1	0.5	1	0	0	1	1	1	1	0.5	1	9
[60]	1	1	0	1	1	0.5	1	1	0.5	1	0	1	9
[61]	1	1	1	1	1	0.5	1	0	1	1	0	1	9.5
[62]	1	1	0	1	0.5	1	1	0	1	1	0.5	1	9
[63]	1	1	0.5	1	0	1	1	1	1	1	1	1	10.5
[64]	1	1	1	1	0.5	0	1	1	1	1	1	1	10.5
[65]	1	1	1	1	0	0.5	1	1	1	1	1	1	10.5
[66]	1	1	0.5	1	1	0	0	1	1	1	1	1	9.5
[67]	1	1	1	1	1	0.5	1	1	1	1	1	1	11.5
[68]	1	1	1	1	1	0	1	1	1	1	0.5	1	10.5
[69]	1	1	1	1	1	0	1	1	1	1	1	1	11
[70]	1	1	0.5	1	1	0	1	0	1	1	0	1	8.5
[71]	1	1	0.5	1	1	1	1	1	1	1	1	1	11.5
[72]	1	1	0.5	1	0	0.5	1	1	1	1	1	1	10
[73]	1	1	0.5	1	1	1	1	1	1	1	1	1	11.5

calculating the final score, studies that scored below 8 were discarded; thus, 8 papers were not considered for the review process. Lastly, 68 studies qualified for the complete selection process and were chosen for the review. The complete list of scores obtained by the selected studies for the questions presented in Section 4.2, along with their paper references, is given in Table 4.

5.2 Reporting of Research Questions

The shortlisted studies were analyzed thoroughly on the basis of the raised RQs. Answers of the RQs helped us to view these studies from various perspectives. Solutions to the problem statement and implementation results were easily analyzed with the help of information collected while answering the RQs. Answers to the RQs and brief facts about them are presented in the following subsections.

RQ1. Various bad smells identified/corrected

Code smell is a feature of the program code that acts as a pointer to determine the opportunity for refactoring and its application on the code. Some of the code smells do not disturb the working of the software system and, thus, need not be removed. However, certain smells decrease the quality of software that further leads to an increase in cost of maintenance. To maintain the standards of the software and prevent failure of the system, harmful code smells must be detected and removed on a regular basis during software development activities. Refactoring is one way to remove the code smells. Smells can be identified manually or using several tools such as JDeodrant, JSpirit, etc. that are available in the market for smell detection and correction. Table 5 lists all the code smells detected along with their paper references and total count.

Feature envy, data class, and long method class are some of the most recognized bad smells that were detected in 22, 19, and 17 numbers of studies, respectively. “Data class” bad smell occurs if a class does not perform any function and contains only data or attributes. Such classes cannot operate by themselves on the data owned by them. “Feature envy” smell occurs in a method if it makes use of attributes (fields, methods) of some other class more than its own class. This indicates that the method is wrongly placed due to the low usage of its own data. “Long method” code smell occurs if the size of the method becomes too vast. Generally, LOC crossing the limit of 15 is considered in the category of long method bad smell. Apart from the list of bad smells mentioned in Table 5, several other code smells were identified in the shortlisted studies. Those smells have a count of 1 each and are not included in the table due to lack of space.

RQ1.1 Technical aspect of most detected code smells

1) Feature envy: It is a class-level code smell that is categorized under “couplers”. In general, it occurs when some data fields are moved to a particular class, and the operations performed on the data are left behind. Feature envy breaks encapsulation and makes unit testing difficult.

- *How can it be resolved*: If a method uses more attributes and functions of another class to perform any action, simply incorporate the logic of that particular action in the same class of that method.
- *Refactoring suggestions*: Move method, extract method, and extract class.

2) Long method: It is a method-level code smell that is categorized under “bloaters”. This bad smell complicates readability and understandability. Duplicate codes might get ignored if the size of the method is too long. Thus, it becomes necessary to resolve the long method smell.

- *How can it be resolved*: If a method needs description or comments then that section must be placed in another newly created method. Any part of the method that needs explanation must be split into another method in order to decrease the complexity.
- *Refactoring suggestions*: Replace temp with query, introduce parameter object, decompose conditionals, extract method, and replace method with method object.

3) Data class: It is a class-level code smell that is categorized under "dispensables". Generally, data classes are not much harmful, but when the size of the project increases, internal qualities such as coupling and cohesion are affected due to data class smell. Due to increased dependencies, coupling between classes is increased.

- *How can it be resolved*: Operations on the data or the required methods must be moved within the data class in order to decrease the dependencies.
- *Refactoring suggestions*: Move method, encapsulate field, hide method, remove setting method, extract method, and encapsulate collections.

Table 5. Bad smells detected

S. No.	Bad smell	References	Count
1	Lazy class	[9], [13], [16], [26], [39], [45], [50], [55], [60], [64], [65]	11
2	Temporary field	[9], [13], [16], [45], [64]	5
3	Duplicate code	[12], [55], [64]	3
4	Data clumps	[12], [1], [47], [51],[64]	5
5	Blob	[25], [37], [43],[65], [71], [73]	6
6	Spaghetti Code	[37], [50], [57], [60], [65], [71],[73]	7
7	Message chain	[12], [16], [18], [39], [47], [50], [54], [60], [64], [67]	10
8	Refused bequest	[14], [39], [47], [50], [59], [60], [64], [66]	8
9	Large class	[13], [18], [26], [35], [39], [44], [59], [64], [69]	9
10	Data class	[13], [14], [16], [18],[26], [39], [47], [53], [54], [59],[64] [65], [66], [67], [68], [69], [70], [71], [73]	19
11	Feature envy	[13], [14], [22],[25], [27], [1], [36], [39], [43], [47], [50], [51], [54], [59], [60], [64], [65], [66], [67], [69],[68], [73]	22
12	Inappropriate intimacy	[13], [16], [64]	3
13	God class	[14], [1], [36], [41], [47], [51], [52], [53], [60], [66], [68]	**11**
14	Long parameter list	[13], [16], [18], [39], [45], [50], [51], [53], [60], [64]	10
15	Brain method	[16],[1], [66]	3
16	Abstract class	[16], [18]	2
17	Complex	[18], [50], [57], [60]	4
18	Shotgun survey	[13], [64], [66], [73]	4
19	Schizophrenic class	[47], [73]	2
20	Divergent change	[25], [44], [60], [64]	4
21	Middle man	[12], [13],[16], [21], [39], [64]	6
22	Long method	[18], [24], [26], [1], [39], [43], [45], [50], [51], [52], [53], [58], [59], [60], [64], [68], [69]	17
23	Parallel inheritance hierarchy	[26], [64]	2
24	Intensive coupling	[1], [47], [66]	3
25	Brain class	[1], [41], [66]	3
26	Type checking	[19], [1], [52], [53]	4
27	Switch statements	[12], [26], [39], [45], [64]	5
28	Functional decomposition	[37], [57],[65], [71], [73]	5
29	Speculative generality	[12], [13], [16], [50], [64]	5
	Class data should be private	[50], [57], [60]	3
30	Class hierarchy problem	[51], [53]	2
31	Nested try statements	[51], [53]	2

RQ2. Refactoring techniques used or identified in the studies

Refactoring performs certain transformations on the source code that helps to preserve its behavior while restructuring it. Refactoring removes code smells, improving software quality of a project and reducing the maintenance cost. Users must not mix refactoring as rewriting since refactoring does not modify the functionality of a code. Developers must refactor their code on a regular basis to maintain the quality and standard of the software project. Refactoring can be either performed manually or by using several tools such as JDeodrant, IntelliJ, etc. that are available to perform automatic refactoring.

Fowler [3] introduced 72 different types of refactoring methods that can be used to improve software quality. Table 6 depicts various refactoring techniques, along with their assigned paper references. The table also includes the detail of the total count of papers in which the techniques were identified or applied.

After analyzing Table 6, we found that extract class, move method, and extract method are the top 3 most used refactoring techniques. They were used in a total of 19, 17, and 15 studies, respectively. Also, we found that extract method and move method are also some of the majorly identified refactoring techniques. Extract class refactoring is used to create a new class in which methods or data of the previous class are transferred. This refactoring technique is applied to a class when its scope becomes vague or it is overloaded with lots of responsibilities**.** Move method refactoring is used to relocate a method from its existing class to another class. The new class is the one where the method is used recurrently. Extract method refactoring is considered as one of the essential building blocks of the refactoring process. In this approach, a chunk of code is moved from the current method to a new method, whose name explains its functionality. Apart from the list of refactoring techniques mentioned in Table 5, several other techniques were used or identified in the shortlisted studies. However, since these refactoring methods have a count of 1 each, they are not included in the table due to lack of space.

RQ2.1 Technical aspects of most used refactoring techniques

1) Move method: It is used when a method is more frequently used by some other class rather than its own class. It is categorized under “moving features between objects”.

- *How is it performed*: The method is removed from the previous class and placed in the new class where it is utilized more frequently along with its dependent data.
- *Benefit*: It improves the cohesion within class and decreases the inter-class dependencies.
- *Bad smells eliminated*: Switch statements, parallel inheritance hierarchy, shotgun surgery, feature envy, inappropriate intimacy, data class, message chains, and message chains.

2) Extract method: It is used when a method contains many lines that can be fragmented into other methods. It is categorized under “composing methods”.

- *How to perform:* A new method is created and named as the function that it performs. Code from the previous method is copied and placed into the new method. The dependent fields are passed as parameters to this new method.
- *Benefit:* It improves code readability and decreases duplication of code.
- *Bad smells eliminated:* Duplicate code, message chains, long method, switch statements, feature envy, comments, and data class.

3) **Extract class:** It is used when a class performs more operations than required. It is categorized under “moving features between objects”.

- *How to perform:* A new class is created, and the relevant methods and fields are placed in it as per its functionality. A relationship, preferably unidirectional, between the old and new class is created, and the classes are renamed as per their jobs.
- *Benefit:* The code will become more understandable, and the Single Responsibility principle of a class will be maintained. This will further improve the reliability of the class.
- *Bad smells eliminated:* Large class, duplicate code, data clumps, divergent change, primitive obsession, inappropriate intimacy, temporary field.

Table 6. Refactoring techniques

S. No.	Refactoring technique	Refactoring applied	Count	Refactoring identified	Count
1	Extract method	[8], [11], [15], [17], [21], [23], [1], [37], [38], [4], [48], [51], [52], [53], [62],	15	[16], [33], [42], [43], [47], [50], [58], [60]	8
2	Remove setting method	-	-	[16]	1
3	Replace temp with query	[8], [24], [38]	3	[28]	1
4	Hide delegate	[67]	1	[16], [47]	2
5	Replace method with method object	[15], [23], [24]	3	[50]	1
6	Separate query from modifier	-	-	[28], [50]	2
7	Replace data value with object	[15]	1		
8	Extract superclass	[8], [46], [72], [73]	4	[50], [60]	2
9	Move field	[21], [23], [36], [37], [48], [54], [65], [67], [71], [73]	10	[33], [42], [50], [60]	4
10	Inline temp			[16], [33], [50]	3
11	Extract class	[8], [15], [20],[23], [35], [36], [37], [38], [4], [44], [48], [51], [52], [53], [62], [65], [71], [72], [73]	19	[16], [33], [47]	3
12	Move method	[15], [21], [22], [23], [24], [27], [1], [36], [37], [48], [51], [52], [54], [65], [67], [71], [73]	17	[16], [25], [31], [33], [42], [43], [47], [50], [60]	9
13	Encapsulate field	[8], [4], [54], [62], [67], [72]	6	[16], [47]	2
14	Consolidate conditional expression	[4], [62]	2	-	-
15	Add parameter	-	-	[33], [50]	2
16	Replace type code with state/strategy	[19], [40], [52]	3	-	-
17	Collapse hierarchy	[30], [32], [34], [72]	4	[16]	1
18	Introduce null object	-	-	[16], [50]	2
19	Introduce assertion	-	-	[33], [50]	2
20	Encapsulate collection	-	-	[16], [47]	2
21	Push down method	[23],[30], [32], [34], [37], [48], [65], [67], [71], [72]	10	[42], [50], [60]	3
22	Remove parameter	[53]	1	[28], [33], [50]	3
23	Pull up field	[23], [30], [32], [34], [37], [48], [65], [71], [72], [73]	10	[42], [50], [60]	3
24	Form template method	-	-	[47], [50]	2
25	Introduce parameter object	[24], [1]	2	[16], [47], [50]	3
26	Inline method	[23], [55]	2	[16], [33], [42], [50], [60]	5
27	Inline class	[8], [1], [37], [48], [53], [55], [65], [71], [72], [73]	10	[16],[25], [43], [47]	5
28	Pull up method	[17], [30], [32], [34], [37], [48], [52], [65],[71], [72], [73]	11	[42], [50]	2
29	Replace delegation with Inheritance	[23], [30], [32], [34], [72]	5	[16]	1
30	Push down field	[23], [30], [32], [34], [48], [65], [71], [72]	8	[42], [50], [60]	3
31	Extract subclass	[8], [20], [36], [49], [72], [73]	6	[25], [47]	2
32	Preserve whole object	[24]	1	[16], [47]	2
33	Extract interface	[36], [37], [63], [65], [73], [71]	6	[33], [42], [47], [60]	4
34	Rename method	[72]	1	[33], [43], [50]	3
35	Duplicate observed data	[36]	1	-	-
36	Replace inheritance with delegation	[23], [30], [32], [34], [72]	5	[16]	1

RQ3. Software metrics used

Measurement of software quality with the help of software metrics is of vital importance because metrics help to easily understand the properties of the source code. Fig. 4 shows the categorization of software metrics used in the studies on the basis of their internal quality. As Fig. 4 illustrates, complexity and coupling metrics are most frequently used, i.e., in 29 studies each, followed by cohesion and size metrics that are used in 28 numbers of studies each.

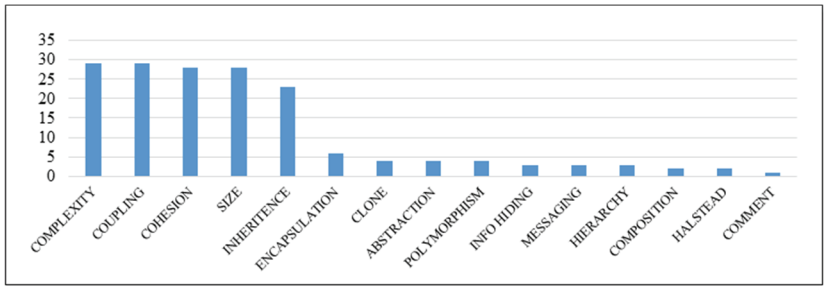

Fig. 4. Categories of software metrics used in the studies.

RQ4. Data collection procedure

Dataset is a compilation of logically related data that can be used for research. The datasets used in the studies are categorized as: open-source, sample, and proprietary datasets. Fig. 5 represents the count of shortlisted studies for each dataset type. The chart below shows that open-source datasets were used in the majority number of studies, i.e., 55, followed by sample datasets that were used in 9 studies. Very few studies used a proprietary dataset for their research.

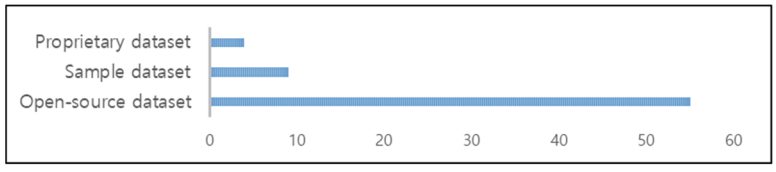

Fig. 5. Types of datasets

RQ4.1. Most frequently used datasets

After analyzing the studies, we found that five open-source datasets were the most frequently used datasets. Table 7 presents the names of the most used datasets along with paper references and count. All the five datasets are open source, large and written in the Java programming language.

Table 7. Most used datasets in the studies

Dataset name	Reference	Count
Apache Xerces	[41], [42], [47], [48], [50], [60], [65], [71], [72], [73]	10
JHotDraw	[30], [31], [37], [48], [49], [65], [64], [71], [73]	9
Gantt Project	[30], [48], [65], [67], [71], [72], [73]	7
Apache Ant	[17], [25], [43], [47], [50], [60], [65]	7
ArgoUML	[23], [27], [35], [47], [50], [56], [60]	7

RQ4.2. Size of datasets

The datasets are divided into three groups (small, medium, and large) on the basis of their size. Division of size is done as follows: (1) small, number of class<100 or number of methods<1000 or lines of code <5,000; (2) medium, number of classes lies between 100–500, number of methods lies between 1,000–5,000 or lines of code lies between 5,000–50,000; and (3) large, number of classes >500, number of methods >5,000 or lines of code >50,000. Fig. 6 illustrates the division of dataset size, and shows that large-sized dataset is mostly preferred for research purposes as evident in 41% of the studies, followed by medium-sized datasets in 31% studies.

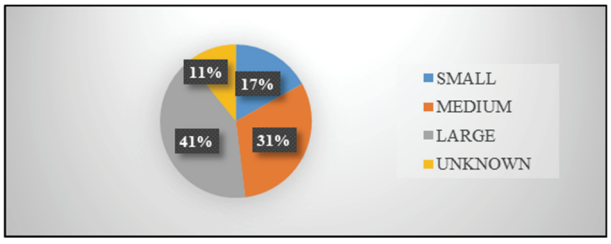

Fig. 6. Size-wise distribution of dataset.

RQ 4.3. Programming languages

The datasets obtained from the studies were written in C++, C, C#, or Java. Table 8 shows the count of datasets based on the programming language in which they were written. Java is the most used language; as 415 datasets were written in it. Other languages make less than 5% of the total usage in the studies.

Table 8. Count of datasets on the basis of programming language

S. No.	Language	Count
1	.Net	1
2	C	4
3	C++	7
4	C#	4
5	Java	415

Table 9. Comparison of work done

Type	Reference	Count
Automated	[9], [10], [11], [12], [13], [14], [16], [17], [18], [19], [22], [26], [29], [30],[32], [33], [34], [1], [36], [37], [38], [40], [41], [42], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [60], [61], [62], [63], [64], [66], [67], [68], [69], [70], [71], [73]	50
Manual	[8], [15], [20], [21], [23], [24], [25], [27], [28], [31], [35], [39], [4], [43], [44], [59], [65], [72]	18

RQ5. Usage of tools

Various tools were used in the shortlisted studies to perform refactoring, identify bad smells, and extract the values of software metrics. Table 9 shows the total count and paper references of the selected studies that performed the process as mentioned above, either manually or with the help of tools. The results of Table 9 show that the usage of tools is preferred over performing the task manually. A total of

50 studies used tools for their research; whereas manual work was done in 18 studies. The complete list of tools used in the shortlisted studies is provided in Table 10.

Table 10. List of the number of times specific tools were used for refactoring techniques, bad smell identification, and software metrics

S. No.	Tool name	Software metrics	Refactoring	Bad smell detection/correction	References
1	Metrics	3			[9], [17], [36]
2	SourceMeter	4			[10], [33], [38], [61]
3	RefactoringAssistant		1	1	[11]
4	PMD			4	[12], [64], [68], [69]
5	Columbus wrapper framework	1		1	[13]
6	Borland Together			1	[14]
7	JFly			1	[16]
8	JDeodrant		7	6	[17], [19], [22], [1], [36], [40], [51], [52], [58]
9	CCFinder			1	[17]
10	DÉCOR			2	[18], [57]
11	BSDT			1	[26]
12	CKJM	3			[29], [37], [52]
13	CCCC	1			[29]
14	Code-Imp		2	1	[30], [32], [34]
15	Robusta			1	[51]
16	RefFinder	2	3		[33], [1], [37], [71], [73]
17	Iplasma	1	1	2	[1], [41], [68], [69]
18	Eclipse		5	2	[1], [37], [48], [50], [51], [53]
19	IntelliJ	2	2	1	[1], [51], [54], [67]
20	Incode		1	1	[1], [36]
21	RefactorIT			1	[1]
22	Infusion			3	[47], [54], [67], [73]
23	Refactoring Miner		2		[42], [60]
24	Metric Parser	1			[46]
25	DART			1	[55]
26	Inspector Gadget		1	1	[56]
27	LET-C	1			[62]
28	Multiple language smells detector			1	[64]
29	Checkstyle			1	[64]
30	Spirit			1	[66]
31	Fluid tool			1	[68], [69]
32	Anti-pattern scanner			1	[69]
33	FRC detector			1	[70]
34	Design features and metrics for Jav	1			[69]

RQ5.1. Various tools used for refactoring, bad smell and software metrics

In the shortlisted studies, tools were majorly used for applying refactoring, detecting/correcting bad smells, and obtaining the values of software metrics. Table 10 shows the list of tools that were used along with their paper reference and count.

From Table 10, we can conclude that in 25 studies, refactoring is performed automatically. Also 37 studies, showed automatic detection/correction of bad smells. The values for software metrics are

calculated with the help of tools in 20 selected studies. The most used tools for software metrics are SourceMeter and Metrics. JDeodrant and Eclipse are most frequently used for refactoring whereas PMD and JDeodrant are most used tools for bad smell detection/correction.

6. Conclusions

In this study, a comprehensive literature review was performed to analyze refactoring techniques, code smells, and software metrics. After an exhaustive search was performed in eight digital libraries, 106 studies were selected between the years 2001 and 2019. On further filtration, 68 studies were shortlisted and analyzed to answer the RQs. The main findings of the SLR are:

- Extract method, extract class, and move method are the most used refactoring techniques in the selected studies.
- Feature envy, data class, and long method code smells are detected in most of the studies.
- Refactoring and bad smell detection are majorly performed automatically, i.e., with the help of tools.
- Complexity, coupling, cohesion, and size metrics are the most frequently used object-oriented software metrics.
- Datasets from open-source are majorly used in selected studies. ArgoUML, Apache Xerces, Apache Ant, Gantt Project, and JHotDraw are the top 5 most used datasets.
- Large datasets, mostly written in Java programming language, were preferred for research work.
- A vast number of studies made use of tools- SourceMeter, JDeodrant, and Metrics for their research purpose.

7. Future Directions

Code smells, refactoring techniques and software metrics share a relationship which when further analyzed can help the developers to improve software quality. The current work might help the readers in exploring the three fields from a broader perspective. After analyzing various studies and on drawing conclusions, some of the research work that can be done in this field is given as follows:

- It is observed that most of the case studies or projects selected in conducting research in the field of refactoring and bad smells are Java-based. Therefore, a detailed analysis can be conducted to generalize the results for all object-oriented language-based projects including C, C++, C#, etc.
- It is also observed that every class of the software contains numerous numbers of bad smells, which further increases the need for the application of different types of refactoring techniques. As a future direction, research can be conducted on prioritization of classes and identification of the optimum refactoring sequence in order to reduce the efforts of maintenance phase.
- There are only a few free tools available for code detection and application of refactoring techniques. Therefore, in future, a tool can be proposed that will benefit the task while simultaneously help in enhancing the software quality.
- A systematic review can be conducted to analyze the impact of refactoring on different software attributes such as internal and external quality features.
- An in-depth study can be conducted on the different types of code smells identified till date, along with their respective refactoring techniques.

- A detailed analysis can be conducted to explore the opportunities of search based refactoring which is an optimization problem in which the best sequence for refactoring is found using a searching algorithm.
- A further investigation can be done to analyze the relationship between refactoring and frequently used object-oriented software metrics, namely complexity, coupling, cohesion, and size metrics, in this research domain. Studies can also be analyzed on the basis of external quality attributes such as reliability, efficiency, usability, etc.
- A study can be conducted to investigate the impact of refactoring on development time and ease of locating errors in the source code.
- Effects of code refactoring techniques on development of agile software and improvement of database quality can also be explored in the coming future.
- Code refactoring has become as essential discipline of software development. A further investigation can be done to explore the opportunities of refactoring on user interface, detection and correction of anti-patterns, etc.
- A study can be conducted for prediction of code smells using various machine learning methods. The impact of software metrics on code smell prediction can also be analyzed.
- Further, prediction rules based on object oriented software metrics to detect code smells can also be generated using machine learning classifiers.
- In the near future, empirical studies can be conducted to explore the opportunities of various other fields such as web application refactoring, big data refactoring, cloud refactoring, spreadsheet refactoring, etc.

References

[1] F. A. Fontana, M. Mangiacavalli, D. Pochiero, and M. Zanoni, "On experimenting refactoring tools to remove code smells," in *Scientific Workshop Proceedings of the XP2015*, Helsinki, Finland, 2015, pp.1-8.

[2] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: an experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, article no. 5, 2012.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley; 2019.

[4] R. Malhotra and A. Chug, "An empirical study to assess the effects of refactoring on software maintainability," in *Proceedings of 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Jaipur, India, 2016, pp. 110-117.

[5] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126-139, 2004.

[6] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179-202, 2011.

[7] M. Xenos, D. Stavrinoudis, K. Zikouli, and D. Christodoulakis, "Object-oriented metrics-a survey," *in Proceedings of the Federation of European Software Measurement Associations (FESMA)*, Madrid Madrid, Spain, 2000, pp. 1-10.

[8] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and Software Technology*, vol. 51, no. 9, pp. 1319-1326, 2009.

[9] M. J. Munro, "Product metrics for automatic identification of "Bad smell" design problems in Java source-code," in *Proceedings of 11th IEEE International Software Metrics Symposium (METRICS'05)*, Como, Italy, 2005, pp. 15.

[10] I. Kadar, P. Hegedus, R. Ferenc, and T. Gyimothy, "A code refactoring dataset and its assessment regarding software maintainability," in *Proceedings of 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Suita, Japan, 2016, pp. 599-603.

[11] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *Proceedings of International Conference on Software Maintenance*, Montreal, Canada, 2002, pp. 576-585.

[12] M. Zhang, N. Baddoo, P. Wernick, and T. Hall, "Prioritising refactoring using code bad smells," in *Proceedings of 2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops*, Berlin, Germany, 2011, pp. 458-464.

[13] S. Singh and K. S. Kahlon, "Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5, pp. 1-10, 2011.

[14] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, vol. 80, no. 7, pp. 1120-1128, 2007.

[15] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring - improving coupling and cohesion of existing code," in *Proceedings of the 11th Working Conference on Reverse Engineering*, Delft, Netherlands, 2004, pp. 144-151.

[16] M. Hammad and A. Labadi, "Automatic detection of bad smells from code changes," *International Review on Computers and Software*, vol. 11, no. 11, pp. 1016-1027, 2016.

[17] E. H. Vashisht, S. Bharadwaj, and S. Sharma, "Impact of Clone Refactoring on External Quality Attributes of Open Source Softwares," *International Journal of Scientific Research in Computer Science, Engineering, and Information Technology*, vol. 3, no. 8, pp. 86-94, 2018.

[18] F. Khomh, M. Di Penta, and Y. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of 2009 16th Working Conference on Reverse Engineering*, Lille, France, 2009, pp. 75-84.

[19] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: identification and removal of type-checking bad smells," in *Proceedings of 2008 12th European Conference on Software Maintenance and Reengineering*, Athens, Greece, 2008, pp. 329-331.

[20] M. Alshayeb, "Refactoring effect on cohesion metrics," in *Proceedings of 2009 International Conference on Computing, Engineering and Information*, Fullerton, CA, 2009, pp. 3-7.

[21] JJ. Ratzinger, M. Fischer, and H. Gall, "Improving evolvability through refactoring," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, St. Louis, MO, 2005, pp. 1-5.

[22] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "JDeodorant: identification and removal of feature envy bad smells," in *Proceedings of 2007 IEEE International Conference on Software Maintenance*, Paris, France, 2007, pp. 519-520.

[23] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta, "On the impact of refactoring operations on code quality metrics," in *Proceedings of 2014 IEEE International Conference on Software Maintenance and Evolution*, Victoria, BC, Canada, 2014, pp. 456-460.

[24] P. Meananeatra, "Identifying refactoring sequences for improving software maintainability," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, Essen, Germany, 2012, pp. 406-409.

[25] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proceedings of 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Silicon Valley, CA, 2013, pp. 268-278.

[26] P. Danphitsanuphan and T. Suwantada, "Code smell detecting tool and code smell-structure bug relationship," in *Proceedings of 2012 Spring Congress on Engineering and Technology*, Xian, China, 2012, pp. 1-5.

[27] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, and A. De Lucia, "Identifying method friendships to remove the feature envy bad smell: NIER track," in *Proceedings of 2011 33rd International Conference on Software Engineering (ICSE)*, Honolulu, HI, 2011, pp. 820-823.

[28] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, "Automated support for program refactoring using invariants," in *Proceedings IEEE International Conference on Software Maintenance (ICSM)*, Florence, Italy, 2001, pp. 736-743.

[29] K. Stroggylos and D. Spinellis, "Refactoring: does it improve software quality?," in *Proceedings of 5th International Workshop on Software Quality (WoSQ*, Minneapolis, MN, 2007, p. 10.

[30] M. O Cinneide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam, "Experimental assessment of software metrics using automated refactoring," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, Lund, Sweden, 2012, pp. 49-58.

[31] C. Napoli, G. Pappalardo, and E. Tramontana, "Using modularity metrics to assist move method refactoring of large systems," in *Proceedings of 2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*, Taichung, Taiwan, 2013, pp. 529-534.

[32] M. O'Keeffe and M. O. Cinneide, "Search-based refactoring: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 5, pp. 345-364, 2008.

[33] P. Hegedus, I. Kadar, R. Ferenc, and T. Gyimothy, "Empirical evaluation of software maintainability based on a manually validated refactoring dataset," *Information and Software Technology*, vol. 95, pp. 313-327, 2018.

[34] M. O'Keeffe and M. O. Cinneide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502-516, 2008.

[35] J. Dexun, M. Peijun, S. Xiaohong, and W. Tiantian, "Detection and refactoring of bad smell caused by large scale," *International Journal of Software Engineering & Applications*, vol. 4, no. 5, pp. 1-13, 2013.

[36] A. Hamid, M. Ilyas, M. Hummayun, and A. Nawaz, "A Comparative study on code smell detection tools," *International Journal of Advanced Science and Technology*, vol. 60, pp. 25-32, 2013.

[37] A. Chug and M. Gupta, "A quality enhancement through defect reduction using refactoring operation," in *Proceedings of 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Udupi, India, 2017, pp. 1869-1875.

[38] Y. Khrishe and M. Alshayeb, "An empirical study on the effect of the order of applying software refactoring," in *Proceedings of 2016 7th International Conference on Computer Science and Information Technology (CSIT)*, Amman, Jordan, 2016, pp. 1-4.

[39] J. Park, B. Jeon, R. Y. C. Kim, and H. S. Son, "Improving source code against bad-smell code patterns," *Journal of Engineering Technology*, vol. 6, no. 2, pp. 503-516, 2018.

[40] N. Tsantalis and A. Chatzigeorgiou, "Identification of refactoring opportunities introducing polymorphism," *Journal of Systems and Software*, vol. 83, no. 3, pp. 391-404, 2010.

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

[42] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli, "A large-scale empirical exploration on refactoring activities in open source software projects," *Science of Computer Programming*, vol. 180, pp. 1-15, 2019.

[43] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for Smell Detection," in *Proceedings of 2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, Austin, TX, 2016, pp. 1-10.

[44] D. Jiang, P. Ma, X. Su, and T. Wang, "Distance metric based divergent change bad smell detection and refactoring scheme analysis," *International Journal of Innovative Computing, Information and Control*, vol. 10, no. 4, pp. 1519-1531, 2014.

[45] A. Rani and H. Kaur, "Detection of bad smells in source code according to their object oriented metrics," *International Journal for Technological Research in Engineering*, vol. 1, no. 10, pp. 1211-1214, 2014.

[46] Y. Kosker, B. Turhan, and A. Bener, "An expert system for determining candidate software classes for refactoring," *Expert Systems with Applications*, vol. 36, no. 6, pp. 10000-10003, 2009.

[47] N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue, "Revisiting the relationship between code smells and refactoring," in *Proceedings of 2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, Austin, TX, 2016, pp. 1-4.

[48] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-based refactoring using recorded code changes," in *Proceedings of 2013 17th European Conference on Software Maintenance and Reengineering*, Genova, Italy, 2013, pp. 221-230.

[49] J. Al Dallal, "Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics," *Information and Software Technology*, vol. 54, no. 10, pp. 1125-1141, 2012.

[50] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1-14, 2015.

[51] S. Tarwani and A. Chug, "Sequencing of refactoring techniques by Greedy algorithm for maximizing maintainability," in *Proceedings of 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Jaipur, India, 2016, pp. 1397-1403.

[52] R. Malhotra, A. Chug, and P. Khosla, "Prioritization of classes for refactoring: a step towards improvement in software quality," in *Proceedings of the Third International Symposium on Women in Computing and Informatics*, Kochi, India, 2015, pp. 228-234.

[53] A. Chug and S. Tarwani, "Determination of optimum refactoring sequence using A∗ algorithm after prioritization of classes," in *Proceedings of 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Udupi, India, 2017, pp. 1624-1630.

[54] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi, "An empirical study to improve software security through the application of code refactoring," *Information and Software Technology*, vol. 96, pp. 112-125, 2018.

[55] R. Ibrahim, M. Ahmed, R. Nayak, and S. Jamel, "Reducing redundancy of test cases generation using code smell detection and refactoring," *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 3, pp. 367-374, 2020.

[56] A. Blouin, V. Lelli, B. Baudry, and F. Coulon, "User interface design smell: automatic detection and refactoring of Blob listeners," *Information and Software Technology*, vol. 102, pp. 49-64, 2018.

[57] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, Italy, 2015, pp. 403-414.

[58] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou, "Size and cohesion metrics as indicators of the long method bad smell," in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, Beijing, China, 2015, pp. 1-10.

[59] M. A. S. Bigonha, K. Ferreira, P. Souza, B. Sousa, M. Januario, and D. Lima, "The usefulness of software metric thresholds for detection of bad smells and fault prediction," *Information and Software Technology*, vol. 115, pp. 79-92, 2019.

[60] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chavez, "Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, Paderborn, Germany, 2017, pp. 465-475.

[61] L. Kumar, S. M. Satapathy, and L. B. Murthy, "Method level refactoring prediction on five open source Java Projects using machine learning techniques," in *Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference)*, Pune, India, 2019, pp. 1-10.

[62] A. Chug and S. Gupta, "Enhancing the life of legacy software through refactoring based systematic transformation," in *Proceedings of the 10th International Multi-Conference on Complexity, Informatics and Cybernetics (IMCIC 2019)*, Orlando, FL, 2019, pp. 216-221.

[63] S. Kebir, I. Borne, and D. Meslati, "A genetic algorithm-based approach for automated refactoring of component-based software," *Information and Software Technology*, vol. 88, pp. 17-36, 2017.

[64] G. Rasool and Z. Arshad, "A Lightweight approach for detection of code smells," *Arabian Journal for Science and Engineering*, vol. 42, no. 2, pp. 483-506, 2016.

[65] M. W. Mkaouer, M. Kessentini, M. O. Cinneide, S. Hayashi, and K. Deb, "A robust multi-objective approach to balance severity and importance of refactoring opportunities," *Empirical Software Engineering*, vol. 22, no. 2, pp. 894-927, 2016.

[66] S. A. Vidal, C. Marcos, and J. A. Diaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, vol. 23, no. 3, pp. 501-532, 2014.

[67] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "JMove: a novel heuristic and tool to detect move method refactoring opportunities," *Journal of Systems and Software*, vol. 138, pp. 19-36, 2018.

[68] F. Arcelli Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43-58, 2017.
[69] F. Arcelli Fontana, M. V. Mantyla, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143-1191, 2015.
[70] J. Dexun, M. Peijun, S. Xiaohong, and W. Tiantian, "Functional over-related classes bad smell detection and refactoring suggestions," *International Journal of Software Engineering & Applications*, vol. 5, no. 2, pp. 29-47, 2014.
[71] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving multi-objective code-smells correction using development history," *Journal of Systems and Software*, vol. 105, pp. 18-39, 2015.
[72] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb, "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm," *Software Quality Journal*, vol. 25, no. 2, pp. 473-501, 2015.
[73] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," *Software Quality Journal*, vol. 23, no. 2, pp. 323-361, 2014.

Mansi Agnihotri **https://orcid.org/0000-0002-0865-1428**

She received Bachelor's degree in computer applications from Bharati Vidyapeeth Deemed University, Pune and Master's degree in computer applications from Guru Gobind Singh Indraprastha University, Delhi. She is currently pursuing PhD from University School of Information, Communication & Technology, Guru Gobind Singh Indraprastha University, Delhi, since August 2019. Her areas of interest are software engineering, data mining and machine learning.

Anuradha Chug **https://orcid.org/0000-0002-2763-2839**

She has long teaching experience of almost 20 years to her credit as faculty and in administration at various educational institutions in India. She has worked as guest faculty in Netaji Subhash Institute of Information and Technology, Dwarka, New Delhi and Regular Faculty at Government Engineering College, Bikaner. Before picking the current assignment as Assistant Professor at USICT, GGSIP University, she has also worked as Academic Head, Aptech, Meerut and Program Coordinator at Regional Centre, Indira Gandhi National Open University (IGNOU), Meerut. In academics, she has earned her doctorate degree in Software Engineering from the Delhi Technological University, Delhi, India. Before pursuing PhD, she has achieved top rank in her M.Tech. (IT) degree and conferred the University Gold Medal in 2006 from Guru Gobind Singh Indraprastha University. Previously she has acquired her Master's degree in Computer Science from Banasthali Vidyapith, Rajasthan in the year 1993. Her H-index as reported by Google Scholar is 6. She has published more than 30 research papers in international and national journals and conferences. She has also served as reviewer of several national and international journals and conferences in the area of software engineering (ACM transaction, IJKESE, Informatica, Inderscience, JOT, SEED, WCI). Currently, she is also serving as a Co-PI for a DST funded project, in the area of Internet of Things (IoT).