Received 30 August 2023, accepted 9 November 2023, date of publication 17 November 2023,
date of current version 22 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3334258

SURVEY

Research Trends, Detection Methods, Practices, and Challenges in Code Smell: SLR

MUHAMMAD ANIS AL HILMI[1], ALIFIA PUSPANINGRUM[1], DARSIH[1], DANIEL ORANOVA SIAHAAN[2], (Member, IEEE), HERNAWATI SUSANTI SAMOSIR[3], (Member, IEEE), AND AMELIA SAHIRA RAHMA[2]
[1]Department of Informatics, Politeknik Negeri Indramayu, Indramayu 45252, Indonesia
[2]Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya 60117, Indonesia
[3]Faculty of Vocational Studies, Institut Teknologi Del, Toba 22381, Indonesia

Corresponding authors: Muhammad Anis Al Hilmi (alhilmi@polindra.ac.id) and Daniel Oranova Siahaan (daniel@if.its.ac.id)

This work was supported by the Politeknik Negeri Indramayu on the 2022–2023 Inter-Higher Education Cooperation Research (PKPT) Scheme in Collaboration with Institut Teknologi Sepuluh Nopember (ITS) Surabaya.

ABSTRACT **Context:** A code smell indicates a flaw in the design, implementation, or maintenance process that could degrade the software's quality and potentially cause future disruptions. Since being introduced by Beck and Fowler, the term code smell has attracted several studies from researchers and practitioners. However, over time, studies are needed to discuss whether this issue is still interesting and relevant. **Objective:** Conduct a thorough systematic literature review to learn the most recent state of the art for studying code smells, including detection methods, practices, and challenges. Also, an overview of trends and future relevance of the topic of code smell, whether it is still developing, or if there has been a shift in the discussion. **Method:** The search methodology was employed to identify pertinent scholarly articles from reputable databases such as ScienceDirect, IEEE Xplore, ACM Digital Library, SpringerLink, ProQuest, and CiteSeerX. The application of inclusion and exclusion criteria serves to filter the search results. In addition, forward and backward snowballing techniques are employed to enhance the comprehensiveness of the results. **Results:** The inquiry yielded 354 scholarly articles published over the timeframe spanning from January 2013 to July 2022. After inclusion, exclusion, and snowballing techniques were applied, 69 main studies regarding code smells were identified. Many researchers focus on detecting code smells, primarily via machine learning techniques and, to a lesser extent, deep learning methods. Additional subjects encompass the ramifications of code smells; code smells within specific contexts, the correlation between code smells and software metrics, and facets about security, refactoring, and development habits. Contexts and types of code smells vary in the focus of the study. Some tools used are Jspirit, aDoctor, CAME, and SonarQube. The study also explores the concept of design smells and anti-pattern detection. While a singular dominating technique to code smell detection has yet to be thoroughly investigated, other aspects of code smell detection remain that still need to be examined. **Conclusion:** The findings underscore scholarly attention's evolution towards code smells over the years. This study identified significant journals and conferences and influential researchers in this field. The detection methods used include empirical, machine learning, and deep learning. However, challenges include subjective interpretation and limited contextual applicability.

INDEX TERMS Code smell, detection, systematic review, software quality, bad smell.

I. INTRODUCTION AND BACKGROUND

The concept of "code smell," initially introduced by Kent Beck, pertains to observable manifestations that may indicate underlying issues inside a software system [1]. Nevertheless, this assertion only holds sometimes in some instances, and Martin Fowler explains: "Smells are not inherently bad on their own - they are often an indicator of a problem rather than the problem themselves" [2]. Numerous studies have highlighted code smells as a significant signal of potential

The associate editor coordinating the review of this manuscript and approving it for publication was Giuseppe Destefanis.

errors directly associated with a system's overall quality [3]. Code smells are often associated with deficiencies in system design, suboptimal implementation techniques, and the presence of micro and nanopatterns, all of which can contribute to a decrease in software quality [4], [5], [6]. Identifying and managing olfactory perceptions are crucial in software quality assurance, as they contribute significantly to developing high-quality software.

Regarding software quality, apart from its relation to maintainability issues, several studies have looked for links to code smells with other problems, one of which is the vulnerability of program code [7], [8]. Regarding the code smell detection method, several studies have used machine learning [9], [10]. As a research field that has been around for a long time, since Beck and Fowler introduced it in 1997, the term code smell has attracted several studies from researchers and practitioners. However, over time, studies are needed to discuss whether this issue is still interesting and relevant in the future.

The main objective of this study is to provide valuable insights to researchers and professionals regarding the current status of research on the identification of code smells. The primary objective of this study was to identify potential avenues for future research in the field of code smell detection, enhance software development processes, and ensure the production of high-quality software. A comprehensive analysis of the available literature was undertaken to accomplish this objective. The primary objective of this study is to conduct a complete analysis of the research subjects about identifying code smells. This analysis will encompass the examination of prior research, the methodology and instruments employed in said research, and the practical obstacles and occurrences observed in the field of code smell detection.

The current manuscript is structured subsequently: Section II provides a comprehensive examination of the current survey about code smells. This paper's third section analyzes the study's aims, research questions, and review methodologies. Subsequently, it presents a thorough summary of the fundamental discoveries obtained from the investigation. Following this, delves into a comprehensive analysis of the data and acknowledges the inherent limitations included in the research. Finally, Section IV functions as the concluding section of the study.

A. CODE SMELL

This is a short section explaining code smells taken from those proposed by Fowler and Beck [1]. This originally long list of smells could be better for understanding the details, only made for a quick overview. Table 1 below presents a summary of Fowler and Beck's list along with the taxonomy proposed by Mantyla [11], summarized by Haque [12].

1) BLOATERS

Bloaters are a category of smells that pose difficulties in efficient control, primarily due to their substantial size or intensity. Mantyla provides a comprehensive compilation of software design concerns, encompassing the subsequent components: The five code smells referred to are the Long Method, Large Class, Primitive Obsession, Long Parameter List, and Data Clumps. Bloaters commonly arise because of the incorporation of additional functionality into existing systems.

TABLE 1. Taxonomies of code smells.

Taxonomies	Code Smells
Bloaters	Long/God Method, Large Class, Primitive Obsession, Long Parameter List, and Data Clumps
Object-oriented Abusers	Switch Statements, Temporary Fields, Refused Bequest, Alternative Classes with Different Interfaces, and Parallel Inheritance Hierarchies
Change Preventers	Divergent Change and Shotgun Surgery
Dispensables	Lazy Class, Data Class, Duplicate Code, Speculative Generality, and Dead Code
Encapsulators	Message Chain and Middleman
Couplers	Feature Envy and Inappropriate Intimacy
Others	Incomplete Library Class and Comments

a: LONG METHOD

The term "Long Method," referred to as the "God Method," describes a method or procedure characterized by extensive code, typically exceeding ten lines, and incorporating numerous functions.

b: LARGE CLASS

The Large Class encompasses a multitude of fields and techniques. The size of classes frequently expands due to developers' inclusion of different responsibilities within them [13].

c: PRIMITIVE OBSESSION

Many programming languages include support for two distinct data types: record types, which are used to store data in a structured manner, and primitive types, which are used to enhance the organization and structure of code. The phenomenon of Primitive Obsession occurs when a software program is afflicted by the tendency of its developer to favor primitive objects instead of employing more sophisticated classes. The concept of Primitive Obsession is not directly associated with the bloater's feature, although it frequently plays a role in the development of large classes and Long Methods.

2) OBJECT-ORIENTED ABUSERS

The olfactory classification diverges from the fundamental concepts of Object-Oriented Programming (OOP).

a: SWITCH STATEMENT

Object-oriented programming (OOP) is a preventive measure against the utilization of switches, which can result in code duplication within programs. In the given instance, altering an introductory statement and introducing a new one will invariably impact the pre-existing program logic.

b: TEMPORARY FIELDS

Temporary fields are the specific instance or temporary variables utilized within sections of a program. This type of code frequently necessitates clarification because of the absence of a variable's reference beyond its scope and intended usage, rendering it indescribable. Temporary variables are commonly utilized in algorithm implementations as a means for developers to circumvent long parameter lists.

c: REFUSED BEQUEST

A Refuse Bequest is a situation in which a subclass selectively inherits some methods and attributes from its parent class in the code. The absence of enforcement of inheritance consistently signifies a significant design issue due to the need for more precise definitions within hierarchies, resulting in the emergence of sibling classes alongside parent classes.

3) CHANGE PREVENTERS

Change Preventers refer to a group of smells that pose challenges to maintenance procedures due to their presence in current implementations. The olfactory sensation in question may be discerned when a class or its method assumes responsibility for multiple features or functionalities.

a: DIVERGENT CHANGE

Classes that possess many extraneous methods are recognized as classes exhibiting divergent changes. Divergent change arises when developers resort to duplicating and inserting code instead of consolidating the functionality within a single class.

b: SHOTGUN SURGERY

Shotgun Surgery exhibits notable similarities to divergent changes. Nevertheless, numerous classes are structured with layered relationships in this situation. Consequently, when a software developer modifies a minor component of a system, a significant amount of time is required to update the corresponding class.

4) DISPENSABLES

This smell category refers to programs containing unnecessary things that should be removed.

a: LAZY CLASS

Maintaining useless classes results in good use of cost and time. This kind of thing is unnecessary. A class that is not needed but still exists is called a Lazy class.

b: DATA CLASS

Data classes are categorized exclusively as fields, getter, and setter methods. In most cases, other classes tend to modify the data fields employed. These classes lack essential attributes and cannot function inside their respective domains.

c: DUPLICATE CODE

When repetitive code is present across numerous lines, it is possible to substitute or combine the instances. The occurrence of duplicate code is frequently observed in two closely related classes, although it is not uncommon for a single class to include duplicate code.

II. RELATED WORKS

In the code smell research literature, 15 papers are summarized in Table 2. The following is a summary of related research. The related work in code smell detection and analysis can be categorized into several groups based on their focus and methodologies.

A. CODE SMELL DETECTION APPROACHES AND SURVEYS

Several studies have explored detecting code smells. Gupta et al. [13] conducted a comprehensive systematic survey encompassing various aspects of bad smells, including detection techniques and correlations between them. Their study aimed to provide a holistic overview of research in this area and identify code smells that necessitate more attention in the detection process. Agnihotri and Chug [14] performed a systematic survey from the perspectives of bad smells, refactoring, and software metrics. This study aimed to offer valuable insights to software developers and categorized tools based on their application in code smell detection, software refactoring, and metric calculations. Similarly, Dos Reis et al. [15] conducted a systematic literature survey to identify prominent techniques and tools discussed in the literature for detecting and visualizing code smells.

B. TOOL-CENTRIC INVESTIGATIONS

Other studies have specifically focused on tools used in code smell detection. Fernandes et al. [16] conducted a study to compile and document various tools reported and utilized in the literature for detecting bad smells. They aimed to provide a concise summary of these tools, highlighting their main features and effectiveness in detecting bad smells. Liu and Zhang [17] conducted a mapping study to scrutinize the purpose of code smell research and the scope of detection tools. They delved into the empirical software engineering context to determine if the available detection tools could identify all types of code smells.

C. MACHINE LEARNING-BASED CODE SMELL DETECTION

With the advancement of machine learning techniques, several studies have explored their application in code smell detection. Rasool and Arshad [18] reviewed mining techniques for code smells, analyzing different approaches and their results. Azeem et al. [19] conducted a systematic literature review and meta-analysis using machine-learning techniques to detect code smells. They investigated the types of code smells considered, classifications used, and training strategies employed in existing studies. Al-Shaaby et al. [20] undertook a systematic review of detecting code smells

TABLE 2. Summary of selected literature on code smell reviews.

Studies	Goal	RQs
[13], [14], [15]	These academic works aspire to comprehensively review existing research on bad smells, analyzing their detection techniques and correlations and identifying overlooked code smells. The studies also aim to categorize tools used in refactoring, code smell detection, and software metrics studies. The studies also seek to identify the main code smell detection techniques and tools in literature while assessing the integration of visual techniques for enhanced support.	The studies address various research questions, including the influence of bad smells on software, the detectability of known bad smells, correlations between detection techniques for bad smells, and the diversity of techniques available for bad smell detection. They also investigate types of detected code smells, technical attributes of common code smells, diverse refactoring techniques, prevalent software metrics, data collection procedures, dataset characteristics, tool usage analysis, reported techniques for detecting code smells, their effectiveness, and approaches to visualizing code smells for practitioners.
[16], [17]	The studies aim to compile a comprehensive list of tools for detecting bad smells in software by identifying and documenting these tools from existing literature. This goal is motivated by the need to summarize the extensive range of proposed tools in this area. Additionally, the research seeks to understand the primary objectives of code smell studies and the spectrum of code smells that detection tools can effectively identify.	These studies address key inquiries on bad smell detection tools, aiming to identify prevalent tools proposed or utilized in the literature, analyze their main features, and determine the primary categories of bad smells they target. Concurrently, the study investigates the objectives of code smell studies in empirical software engineering research, exploring their comprehensiveness, and assessing the extent of comprehensive experimental analysis conducted in these studies.
[16], [19], [20], [21]	These studies seek to comprehensively analyze code smell detection research by investigating the types of code smells considered, variables for identification, classification methods, and machine learning strategies. It also examines code smell detection datasets, tools, software metric extraction, and techniques, aiming to identify predictors, machine learning methods, analyzed code smells, datasets, projects, and performance measures, focusing on proposing improvements for code smell detection through machine learning.	These studies aim to investigate the landscape of code smell detection research comprehensively. They focus on key aspects, including the types of code smells detected using machine learning, applied machine learning techniques, accuracy measures, utilized datasets, and prominent tools. The research questions encompass machine learning techniques for detection, commonly detected code smells, accuracy measures, utilized datasets, frequently used machine learning tools, predictors, AI methods, analyzed code smells, datasets and projects, performance metrics, and innovative ideas in the field.
[22], [12], [23]	The primary objective of these 3 researches are to provide empirical insights into the effects of code smells, addressing the current gaps in understanding. They seek to analyze empirical studies that delve into these effects comprehensively. Additionally, the study aims to explore the discrepancies in the performance of machine learning algorithms versus developers' views, specifically focusing on the reproducibility of existing studies.	The research questions address various aspects of code smells. They inquire about the themes in studies on code smell effects, the consistency of experimental setups, and the convergence of findings. Additionally, the study examines the reasons behind code smell occurrence and their impact on software development. Furthermore, they assess reproducibility in recent AI/ML-based code smell studies, focusing on machine learning models and dataset acquisition. These questions aim to enhance our understanding of code smells and their implications.
[24], [25], [26]	The research aims to explore the potential of code smells as indicators of system-level maintainability, assess the effectiveness of concern metrics in detecting specific code smells, and provide an in-depth understanding of well-known security smells in microservices, thereby guiding the development of new techniques and solutions for enhanced software quality.	The research questions encompass using code smells to evaluate software product maintainability at a system level and compare code smell approaches with expert and metrics-based methods. They further delve into the accuracy of concern metrics compared to traditional metrics for code smell detection. They examine the influence of subjects' backgrounds on the efficiency of detected code smells. Moreover, these studies address the identification of recognized security smells in microservice-based applications and explores strategies for refactoring these applications to alleviate the impact of security smells.

using machine learning. Their study comprehensively analyzed datasets, tools, and techniques used in this domain. Lewowski and Madeyski [21] conducted a systematic literature review to identify and investigate state-of-the-art detection of code smells using artificial intelligence techniques. This study aims to identify various predictors used in code smell detection prediction models, artificial intelligence methods used, code smells analyzed in scientific literature, datasets and projects used in research to predict code smells, and performance metrics most commonly used in literature.

D. IMPACT OF CODE SMELLS AND THEIR ASSOCIATION WITH DEVELOPERS

Certain studies, such as Santos et al. [22], delved into the effects of code smells and their implications for developers. They conducted a systematic literature review to provide empirical support for understanding the effects of code smells. Another research by Haque et al. [12] aimed to represent code smells more comprehensibly for developers and understand their impact on software development. Lewowski and Madeyski [23] focused on the consistency between machine learning algorithms' performance detecting code smells and developers' views. Their research emphasized the reproducibility of existing studies and the alignment between machine learning models and developer perceptions.

E. CONTEXT-SPECIFIC STUDIES

Yamashita and Counsell [24] conducted empirical research to explore the potential of code smells as indicators of software product maintainability. Their study aimed to evaluate code smell detection approaches and compare them with expert and metrics-based methods. Padilha et al. [25] reported empirical results regarding using the anxiety metric in detecting specific code smells. Their investigation aimed to

TABLE 3. Research questions on literature review.

ID	Research Question	Motivation
RQ1	Which journal/conference is the most significant code smell journal?	Identify the most significant journal/conference in the code smell topic
RQ2	Who are the most active and influential researchers in the code smell topic?	Identify the most active and influential researchers who contributed so much on a research area of code smell
RQ3	What kind of research topics are selected by researchers in the code smell field?	Identify research topics and trends in code smell
RQ4	What are practices of code smell detection according to published empirical studies?	Identify the practices of code smell detection according to published empirical studies
RQ5.1	What kind of methods are used for code smell detection?	Identify opportunities and trends for code smell detection method
RQ5.2	Which method performs best when used for code smell detection?	Identify the best method in code smell detection
RQ5.3	What kind of method improvements are proposed for code smell detection?	Identify the proposed method improvements for detect code smell
RQ6	What are the challenges of code smell detection?	Identify the challenges in code smell detection

assess the effectiveness of this metric compared to traditional ones. Additionally, Ponce et al. [26] conducted a multivocal literature review on security smells in microservices-based applications. Their research provided insights into recognized security smells and their impact on microservices applications.

These various categories and summaries of studies collectively contribute to the comprehensive understanding and advancement of code smell detection, analysis, and its impact on software development practices.

Although existing reviews highlight that code smells are quite research intense, little is known about research trends, methods, practices, and challenges to code smell detection. This article aims to provide a substantial update exploring the latest (mostly in the last decade) trends in code smell research. By integrating the findings from a summary of related research, this article will provide an overview of the latest developments in code smell detection, their impact, and related practices and challenges.

III. RESEARCH METHOD

In this research, the systematic literature review is used as a research methodology to define, identify, assess, and analyze to answer the code smell phenomenon. Several researchers have investigated several topics in code smell. In this research, we followed [27] the guidelines to conduct a systematic review. There are several steps in a systematic review: planning, implementation, and reporting results.

A. PLANNING THE REVIEW

Several research questions are proposed according to the research objectives. In addition, the details are described below for searching terms and inclusion/exclusion criteria as the strategy.

TABLE 4. Search sources.

Electronic databases	ACM Digital Library, IEEE Xplore, Proquest, SpringerLink, ScienceDirect, and CiteSeerX
Searched items	Journal, workshop, and conference papers
Search applied on	Full text-to avoid missing any of the papers that do not include our search keywords in titles or abstracts, but are relevant to the review object
Language	English
Publication period	From January 2013 to July 2022

1) REVIEW OBJECTIVES AND RESEARCH QUESTIONS

This systematic literature review investigates how code smell has been empirically addressed. These several research questions are proposed in our research:

The selection paper is a prior stage to find a significant role in our topic. In this research, we used several pieces of literature, as shown in Table 4.

2) SEARCH STRATEGY

Once the study goals and questions were established, a formal search strategy was devised to systematically assess all relevant empirical materials about the objective of this review.

In the collection process, we search as many papers as possible to get more representative papers for our systematic literature review. Electronic databases and proceedings are included to define search space. Firstly, electronic databases are identified to find meaningful references (snowballing). This additional approach was implemented to incorporate any possible literature that may have been necessary to be incorporated [28]. Subsequently, the collected studies underwent the application of inclusion and exclusion criteria.

3) SEARCH CRITERIA

We used several terms from the main topic. Electronic databases are identified using well-known keywords related to code smells, such as code smell detection, anti-pattern, refactoring, and software smell. We considered specific terms by using quotes and connectives AND and OR. The search string is presented as follows.

("code smell" OR "software smell") AND ("detection" OR "anti-pattern" OR "refactoring") AND ("methodology" OR "technique" OR "tool" OR "practice" OR "challenge")

4) INCLUSION AND EXCLUSION CRITERIA

In order to ascertain the suitability of a study for inclusion, the researchers employed a set of predetermined criteria for inclusion and exclusion. The criteria for inclusion in this study are as follows:

(I1) The document is composed in the English language;

(I2) The search terms specified in the search criteria section are pertinent to the topic;

(I3) The study presented in this paper is based on empirical research and has been published in a reputable academic journal and presented at a conference;

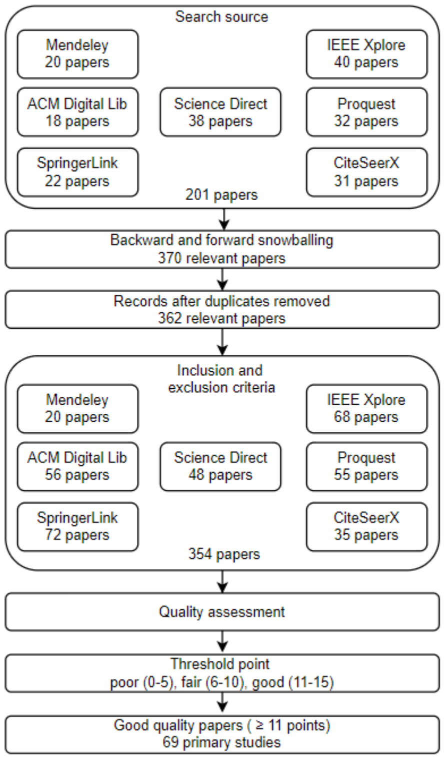

FIGURE 1. Stages of search to produce a primary study.

(I4) The study was published from January 2013 to July 2022.

The refinement of exclusion criteria was undertaken throughout the research. Initially, a thorough examination was conducted on a subset of the obtained articles to establish a consensus among all researchers participating in the Systematic Literature Review (SLR). Subsequently, the exclusion criteria were repeatedly refined:

(E1) Research that does not primarily focus on code scent but mentions code smell incidentally, such as studies that use code smell as a descriptive term;

(E2) Research that does not address code smells in the field of software engineering;

(E3) Research projects that fail to meet the specified inclusion criteria; and

(E4) In academic discourse, various forms of communication are employed to convey ideas and perspectives. These include opinions, viewpoints, keynotes, discussions, editorials, comments, tutorials, prefaces, anecdote papers, and presentations in slide formats, which corresponding written papers may not necessarily accompany.

B. CONDUCTING THE REVIEW

This section shows the results of our systematic search and data extraction from pertinent sources and databases.

1) STUDY SEARCH AND SELECTION

During the period from January 2013 to July 2022, a comprehensive search was conducted across various academic databases to collect relevant studies on the topic of code smells, with the duration of the search aligned with related work, ranging from 10 to 20 years. Consequently, the time-frame for this paper follows a 10-year duration, aiming to capture not only historical perspectives but also to discern recent trends from the last decade.

TABLE 5. Quality criteria for study selection.

Item	Assessment criteria	Score	Description
QA1	Was there a clear statement of the objectives of the research?	-1	No, the objectives were not described
		0	The objectives were partially but unclearly described
		1	Yes, the objectives were well described and clear
QA2	Does the research introduce a detailed description of the proposed solution or approach?	-1	No, details were missing
		0	Partially, if you wish to use the approach or solution, you must read the references
		1	Yes, the approach can be used with the presented details
QA3	Is the proposed solution or approach validated?	-1	No, it was not validated
		0	It was validated in a lab, or only portions of the proposal were validated
		1	Yes, by a case study
QA4	Does the research present an opinion or viewpoint?	-1	Yes, it does
		0	Partially because the corresponding work was explained, and the paper was set into a specific context
		1	No, the paper was based on research
QA5	Has the study been cited in other scientific publications?	-1	No, no one cited the study
		0	Partially. Between one and five scientific papers cited the study
		1	Yes, more than five scientific papers cited the study

We searched online libraries using predetermined search terms: The ACM Digital Library, IEEE Xplore, Proquest, SpringerLink, and ScienceDirect are prominent academic databases, and CiteSeerX.

We run searches on electronic databases in parallel. Initially, a comprehensive data search was conducted, and the obtained findings were meticulously documented in a spreadsheet and Mendeley, a reference management software. We screened by checking the title, abstract, and keywords. The process involves the identification of pertinent scholarly articles, which are then categorized and recorded in a spreadsheet. Subsequently, these articles are downloaded and imported into the Mendeley program for further management and organization. We also ensure that no studies are redundant when using this approach. From there, we get 201 relevant studies.

The backward and forward snowballing techniques were then used to increase the number of relevant studies to make them more comprehensive. From this stage, we obtained an additional 169 studies. So, the total number

TABLE 6. Data extraction form.

#	Study data	Description	Relevant RQ
1	Title	-	RQ3
2	Authors	-	RQ2
3	Year	-	RQ1
4	Publisher	-	RQ1
5	Type of article	Journal, conference	RQ1
6	Author country	-	RQ2
7	Application context	specific context	RQ4
8	Research goal	What is the contribution of the study?	RQ3
9	Research goal category	-	RQ3
10	Research methods	What research methods did the study employ?	RQ5
11	Data	What data did the study use?	RQ5
12	Validation	What validation technique did the study apply?	RQ5
13	Code smell detection technique	What code smell detection technique did the study use?	RQ5
14	Code smell detection tools	What code smell detection tool did the study use?	RQ5
15	Challenge and limitation	-	RQ6
16	Future work	-	RQ6

of studies obtained after snowballing was 370 studies. Next, an examination of duplicate papers was carried out; inclusion criteria were applied, where the title, abstract and keywords were assessed, thereby reducing the number of initial studies to 354. The search yielded a total of 354 studies, distributed across different databases as follows: Mendeley with 20 studies, CiteSeerX with 35 studies, Science Direct with 48 studies, IEEE Xplore with 68 studies, Proquest with 55 studies, ACM with 56 studies, and Springer with 72 studies.

After that, an examination was carried out with five questions, the details of which are in Table 5 based on [30]. The maximum value for each question is 1; the smallest is -1. The assessment is carried out by three researchers for each paper, so the maximum score for a paper if it is assessed as perfect by three examiners is 15 points. After that, from all papers, to get papers with good quality, sort them based on points. With three quality categories: poor (0-5), fair (6-10), and good (11-15). Then, papers were taken that were in a good category, namely 11 points and above, resulting in 69 primary studies. Figure 1 displays the study search and selection process for 69 primary studies.

This extensive collection of studies from diverse sources forms the foundation for this systematic literature review, enabling a comprehensive analysis of the research trends, detection methods, practices, and challenges related to code smells.

2) QUALITY ASSESSMENT

Quality evaluation was employed to evaluate the methodological quality of the primary research. In our study, we utilized the quality assessment methodology employed by [29] and [30]. The checklist employed for assessing the quality of the studies included in the analysis is presented in Table 5.

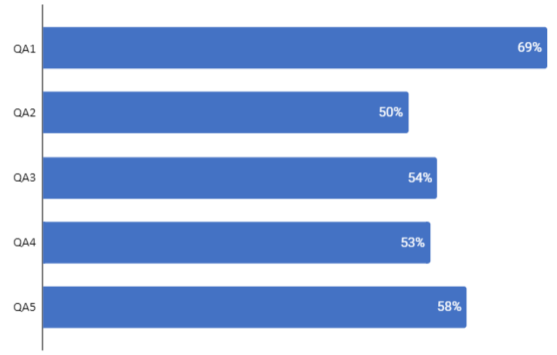

FIGURE 2. Percentage scores for the quality assessments of the studies.

Sixty-nine primary studies were evaluated using quality evaluations, as indicated in Table 5. The initial item (QA1) evaluates the objective of each investigation. The question had a favorable response in 69% of the conducted studies. The second criterion (QA2) evaluated whether the study comprehensively described the approach employed. The question elicited a positive response in 50% of the conducted studies. The third item, referred to as QA3, inquires about the validation method implemented to assess the accuracy and reliability of the obtained results. Notably, 54% of the studies included in the analysis utilized proper validation procedures in their research protocols. The fourth item, QA4, evaluates the extent to which studies are grounded in research instead of being influenced by personal opinion or subjective perspective. It was found that 53% of the studies provided a positive response in this regard. The final component, QA5, is ascertaining the number of citations from research studies. As a result, most studies, precisely 58%, received citations from other studies exceeding five times. Figure 2 illustrates the scores obtained from the quality assessment of the primary studies.

3) DATA EXTRACTION AND SYNTHESIS

The data extraction process was conducted to acquire pertinent information that applies to the study inquiry. The data were extracted by a predetermined extraction form (Table 6). This form facilitated the comprehensive documentation of primary studies to meet our research inquiry.

C. FINDINGS AND DISCUSSION OF OUR REVIEW

Within this particular portion, we will explicate the discoveries and deliberation of our comprehensive analysis of our designated research inquiries.

1) SIGNIFICANT JOURNAL AND CONFERENCE PUBLICATIONS

In this SLR, 69 primary studies were analyzed around code smell. The distribution over the years shows how interest

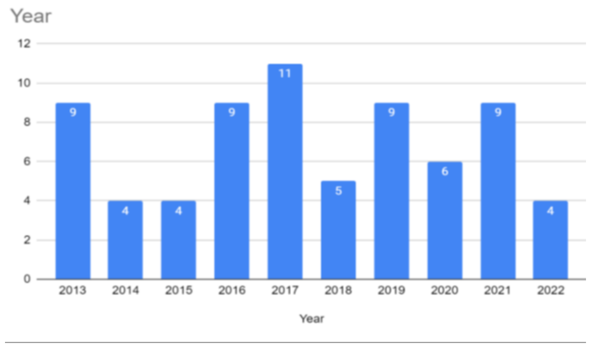

FIGURE 3. Distribution of selected studies over the years.

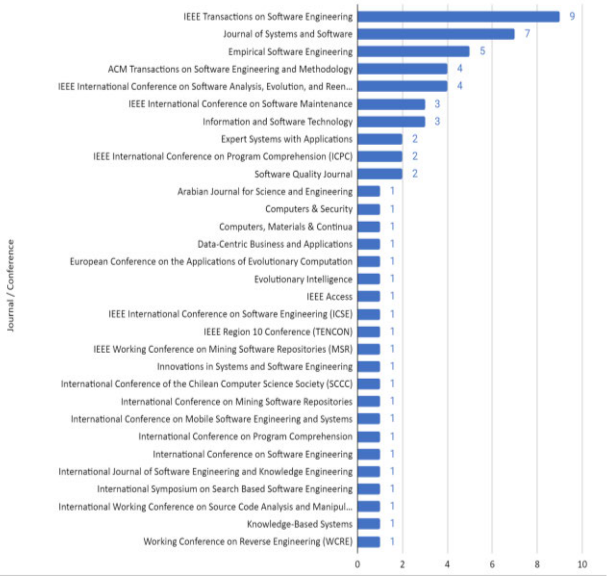

FIGURE 4. Journal and conference publications and distribution of selected studies.

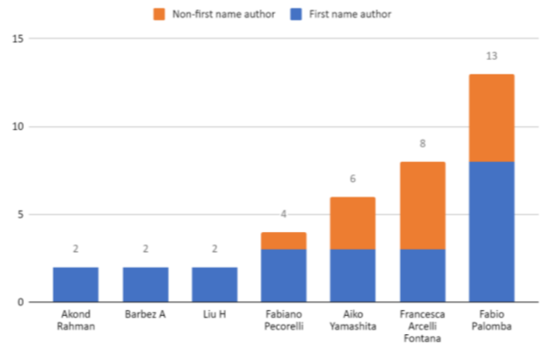

FIGURE 5. Influential research and number of studies.

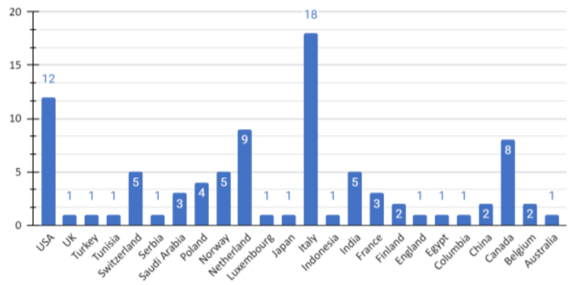

FIGURE 6. Country of author distribution.

in code smells has changed over time. Figure 3 concisely summarizes the distribution studies conducted over the years. Figure 3 also shows that there are fluctuations in the trend, sometimes it goes up, but in several years, it also shows a decline. This can happen because there is a shift in the research paradigm, such as several studies that introduce new terms such as design smell [10], security smell [7], [31], and behavioral smell [8], which are more specific than code smells.

According to a careful examination of relevant primary studies, it has been determined that the IEEE Transaction on Software Engineering holds significant prominence as the foremost publication in code smell detection. This phenomenon is visually represented in Figure 4.

Based on the data presented in Figure 4, it can be observed that IEEE Transactions on Software Engineering has a notable concentration of high-quality papers about code smells. The list of journals and conferences can be a reference for researchers in the future where to publish their articles. Complementing the data for the list of journal publishers, Table 7 presents the Scimago Journal Rank (SJR) values and Q categories (Q1-Q4) of the journals encompassed in the significant investigation of code smell. The table does not encompass conferences.

2) MOST ACTIVE AND INFLUENTIAL RESEARCHERS

The researchers who contributed significantly and demonstrated high activity levels in code smell were examined and recognized from the chosen primary studies. Figure 5 illustrates the researchers who have demonstrated the highest level of activity and influence within the domain of code smell. The researchers were ranked based on the number of primary studies they contributed to. It is worth mentioning that Fabio Palomba, Francesca Arcelli Fontana, Aiko Yamashita, Fabiano Pecorelli, Liu Hui, Antoine Barbez, and Akond Rahman are active researchers on code smell topics.

Figure 6 shows the distribution of authors who have conducted many studies on code smells by country. We list all authors, meaning not just the first author. Europe dominates the number of studies related to code smell, followed by the USA and Canada. Writers from Asia, Africa, and Australia also exist, but the quantity is small. Even so, this shows that the discussion of code smell on all continents attracts researchers and practitioners to conduct studies about it.

The research methods utilized in the 69 selected papers are depicted in Figure 7. Out of the 69 research within this particular category, most are empirical, Most of the examined methods or tools (65 out of 69) were subjected to empirical evaluation. However, it should be noted that such evaluations were conducted without the inclusion of real-world

TABLE 7. Scimago Journal Rank (SJR) of selected journals.

#	Journal Publications	SJR	Q Category
1	IEEE Transactions on Software Engineering	1.71	Q1 in Software
2	Journal of Systems and Software	1.1	Q1 in Software Q1 in Information Systems Q1 in Hardware and Architecture
3	Empirical Software Engineering	1.29	Q1 in Software
4	ACM Transactions on Software Engineering and Methodology	1.19	Q1 in Software
5	Information and Software Technology	1.18	Q1 in Software Q1 in Information Systems Q1 in Computer Science Applications
6	Expert Systems with Applications	1.87	Q1 in Artificial Intelligence Q1 in Computer Science Applications Q1 in Engineering (miscellaneous)
7	Software Quality Journal	0.54	Q1 in Media Technology Q2 in Safety, Risk, Reliability and Quality Q3 in Software
8	Arabian Journal for Science and Engineering	0.48	Q1 in Multidisciplinary
9	Computers and Security	1.61	Q1 in Computer Science (miscellaneous)
10	Computers, Materials & Continua	0.53	Q2 in Computer Science Applications
11	Evolutionary Intelligence	0.57	Q3 in Artificial Intelligence
12	IEEE Access	0.93	Q1 in Computer Science (miscellaneous)
13	Innovations in Systems and Software Engineering	0.29	Q4 in Software
14	International Journal of Software Engineering and Knowledge Engineering	0.28	Q4 in Artificial Intelligence Q3 in Computer Graphics and Computer-Aided Design Q3 in Computer Networks and Communications Q4 in Software
15	Knowledge-Based Systems	2.07	Q1 in Artificial Intelligence Q1 in Information Systems and Management Q1 in Management Information Systems Q1 in Software

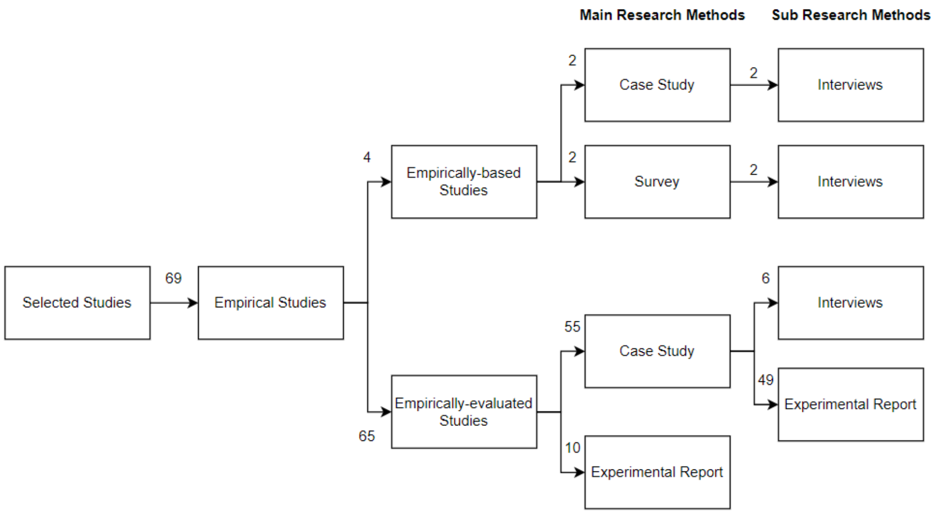

FIGURE 7. Distribution of research methods.

experiments or investigations. In the interim, four studies (surveys and interviews) are included in empirically-based (qualitative) studies. Of the 65 included in empirically-evaluated (quantitative) studies. 10 of them directly presented the experimental results, while the remaining 55 were in the form of case studies, which were also conducted with interviews as a sub-research method (6 studies) and case studies then presented the experimental results of 49 studies. For qualitative studies, two studies raise case studies, and two others use surveys. That two terms, empirically-based study and empirically-evaluated study, following this [27].

D. RESEARCH TOPICS AND PRACTICES IN CODE SMELL

The examination of the chosen original studies indicates that the existing body of code smell research mainly concentrates on five specific topics:

1) CODE SMELL DETECTION

Numerous scholarly publications examine various strategies and methodologies employed in identifying code smells within software systems. This includes machine learning-based, deep learning, heuristic, and metric approaches.

TABLE 8. Code smell research paper topics.

Topics	Studies
Code Smell Detection	[40][41][47][55] [61][62][54][45][42][59]
Code Smells and Machine Learning	[63][64][58][65][8][48] [9] [52][57][10][66][67][51][56][53][68][69][65][70][71] [50][43]
Code Smell Impact	[72][24][73] [3][74][75][76] [5] [77][39][78]
Code Smells and Metrics	[79][80][81][82][83][84][46][85][86]
Code Smell Refactoring	[87][88]
Code Smells in Specific Contexts	[89][35][36][38][73][8][55][90][91][92]
Security Smells	[31][8][7]
Code smell and developer	[32][33][34] [93] [94] [95]

Sub-topic: code smells detection and machine/deep learning: numerous scholarly works delve into the identification of distinct code smells through the utilization of machine learning methodologies and the examination of machine learning strategies, including deep learning, for forecasting code smells.

2) CODE SMELL IMPACT

Several articles discuss the effects of code smells on software, This study aims to investigate the influence of software testing practices on the overall quality of software, maintainability, and performance.

Sub-topic: code smells effect and software engineering metrics: numerous scholarly studies investigate the correlation between code smells and distinct software metrics.

3) CODE SMELL REFACTORING

Several articles discuss refactoring techniques to overcome code smells in software.

4) CODE SMELLS IN SPECIFIC CONTEXTS

Several articles discuss code smells specific to a programming language (such as JavaScript), a platform (such as Android), or a particular software environment.

Sub-topic: security smells: several articles discuss detecting and analyzing security smells in software code.

5) CODE SMELL AND DEVELOPER

Several articles discuss code smell and its relationship with developers, for instance, the level of comprehension, conduct, and interpretation of the code smell by the developer.

Moreover, Table 8 displays a summary of the topics of each study.

From the topics in Table 8, most of the discussion is on detecting code smells, especially those related to machine learning, and developing deep learning. The influence of code smells, and their correlation with other metrics in software engineering is a considerable discussion among numerous individuals. For those who conduct studies related to real life, namely related to developers and software development, several studies are interesting to note, such as in studies [32], [33], [34], where it is sought to find out how developers respond or interact with code. Smell in terms of understanding, different experience levels, and related to the tools used.

Some studies discuss code smells in specific contexts; for example, in certain programming languages, the majority discuss Java, but some discuss other programming languages such as JavaScript [35], [36], [37]. another context, some studies discuss code smells in mobile apps [38], even its impact on energy consumption on mobile devices [39]. Also, code smells on infrastructure configurations such as Ansible and Chef [7] and [31]. those discussing refactoring, there are only two studies, and three studies discuss the connection between code smell and security. This is a future research opportunity because this field can explore many things.

E. METHODS USED IN CODE SMELL DETECTION

To detect the existence of a code smell in a program code, from the 69 selected studies, several methods can be summarized, including [40] and [41] using a competitive coevolutionary search; it is an approach used in the paper to detect code smells, where two populations undergo simultaneous evolution, where the objective of one population is contingent upon the existing population of the other. The individual asserted that their precision and recall scores surpassed 80% on a pre-established criterion. Similar to the method, [42] the term ''multi-objective'' is used in this paper to describe the approach of simultaneously considering multiple objectives or goals. Specifically, it involves maximizing the detection of code-smell examples while minimizing the detection of well-designed code examples. This approach generates detection rules using multi-objective genetic programming (MOGP). The study assessed the suggested multi-objective approach by evaluating seven extensive open-source systems. The results indicated that, on average, the approach successfully detected most of the five distinct types of code smells with a precision rate of 87% and a recall rate of 92%. Furthermore, statistical analysis of the experiments, which were repeated 51 times, demonstrated that the multi-objective approach outperformed existing code smell detectors statistically. Some use detection methods based on structural [43], textual [44], and a combination of the two, as in [45] and [46]. Although the detected code smells are specific to only a few types, for example, God Class, Data Class, Feature Envy, and Long Method. Some detect code smells through SQL queries and regex [47], and there is also a detection visualization. Of the 69 selected articles, since 2015 [48] many code smell detection methods have started to use machine learning techniques, [48] utilizing data changes over time, not just directly from the program code, to then carry out mining, the algorithms used are a priori and FP growth, some also use association rules [49].

According to [9], based on a comparison of several techniques, the best algorithm is J48 (decision tree) and random forest, with high accuracy, above 96%. However, that study was done in 2016, so it is clear that changes are possible. In this study [50], the authors have employed six commonly utilized machine learning methods, namely the Naive Bayes Classifier, Multilayer Perceptron, LogitBoost, Bagging, Random Forest, and Decision Tree, for the purpose of identifying code smells. The findings of this study demonstrate that code smell serves as a robust indicator of the likelihood of class changes. Additionally, the multilayer perceptron emerges as the most efficient method for forecasting change proneness based on code smell. In [51], the whale algorithm was used in the search for code smells; the findings of the study indicate that the suggested framework demonstrates a notable level of accuracy in identifying code smells. Specifically, it achieves an average precision of 94.24% and a recall of 93.4% across five open-source software projects. These results were obtained through the utilization of an 80% training and 20% testing data split. The suggested framework enhances the identification of code smells, hence enhancing the overall quality of software while simultaneously reducing the need for maintenance, time, and cost.

Apart from machine learning, recently several studies have also used deep learning in detecting code smells, as in [52], which specifically discusses Brain Class and Brain Method; he claims that the CNN algorithm, capable of producing 95-97% accuracy, as well as other studies discussing deep learning such as [8], [43], [52], and [53]. Of the existing studies, no studies stand out in detecting code smells because each study usually focuses on a certain context, for example, implementation in a certain programming language, a certain paradigm, or even a certain type of code smell. For tools in detecting code smells, some that have been mentioned are Jspirit [54], aDoctor [55], CAME [43], SLIC [7], ADIODE [56], JDeodorant, HIST, InCode [57], DÉCOR [58], SMAD [10], SLAC [44], and SonarQube [59].

Study [10] introduces the concept of design smells and anti-pattern detection to identify code components that require refactoring. Moreover, the study [46] presents the results of implementing 32 machine-learning algorithms after the execution of feature selection via six distinct iterations of the filter approach to detect code smells. The findings indicate that the machine learning models have demonstrated an improvement in accuracy by 26.5%, an increase in f-measure by 70.9%, a surge in the area under the ROC curve by 26.74%, and a reduction in average training time by 62 seconds when compared to the performance measures of machine learning models executed without feature selection. The utilization of the mutual information feature selection approach in conjunction with the random forest correlation methodology exhibits the most significant influence on performance measurements compared to other filter methods. After doing dimensionality reduction, it was observed that the boosted decision trees (J48) and Naive Bayes algorithms had superior performance among the 32 classifiers.

So that it can be abstracted, on the topic of code smell detection, based on published empirical studies, including various techniques, tools, and methodologies used by researchers and practitioners to identify and analyze code smells in software systems. These practices are derived from the findings of studies that have explored code smells in real-world software projects. Some of the common practices of code smell detection are as follows:

1) MANUAL CODE INSPECTION

One of the early and fundamental practices of code smell detection involves manual code inspection by experienced developers. They review the source code to identify patterns and structures indicative of code smells. Manual inspection of developers' expertise and knowledge of design principles and best practices.

2) STATIC CODE ANALYSIS

The use of static code analysis techniques is prevalent in automatically identifying code smells within software projects. These tools analyze the source code without executing it and flag potential instances of code smells based on predefined rules and patterns. Some popular static analysis tools for code smell detection include SonarQube.

3) METRICS-BASED DETECTION

Researchers often use various software metrics to detect code smells. Metrics like cyclomatic complexity, lines of code, and coupling between objects can indicate potential code smells. By setting threshold values for these metrics, researchers can identify code fragments that exceed the specified limits and may contain code smells.

4) MACHINE LEARNING

Numerous empirical investigations have been conducted to investigate the utilization of machine-learning approaches in detecting code smells. By training machine learning models on labeled datasets of code smells, researchers can build classifiers capable of automatically identifying code smells in new codebases.

5) CODE SMELL PATTERNS

Researchers have identified and documented specific code smell patterns that can be matched against the source code. These patterns serve as templates for identifying code smells based on their structural and behavioral characteristics.

6) RULE-BASED SYSTEMS

Rule-based systems define rules that identify code smells based on specific coding patterns or anti-patterns. When code fragments match these rules, they are flagged as potential instances of code smells.

7) VISUALIZATION TECHNIQUES

Some studies explore using visualization techniques to represent code smells more intuitively and easily. Visual representations help developers identify and understand complex code smell occurrences.

8) COMPARATIVE STUDIES

Empirical studies often compare different code smell detection techniques, tools, or thresholds to determine their effectiveness and efficiency in identifying code smells.

TABLE 9. The challenges of code smell detection.

Study	Challenge	Impact
[40]	Lack of documentation by developers, as code smells are not usually documented like bugs.	The challenge of the lack of documentation by developers makes it difficult to find code that smells examples except in a few open-source systems that are evaluated manually.
[42]	Several studies have specifically targeted a few code smells, but not all of them.	Code smell detection still needs attention to certain code smells for implementation in the real world. No tools or methods can be used in general to recognize well, at least the type of code smell based on Mantyla [11].
[54][49] [71] [57]	Subjective interpretations, low agreement between detectors, and threshold dependability.	It may affect the accuracy and reliability of the results.
[55] [7]	Application in limited contexts.	For implementation in the real world, code smell detection still needs to pay attention to certain contexts, such as programming languages.
[9]	Imbalanced dataset for ML/DL-based detection method.	It can lead to poor accuracy for machine learning-based approaches, making it difficult to detect code smells effectively.
[52]	Requires a large dataset size for the learning process, especially deep learning-based detection.	It can lead to low precision and recall scores in detecting code-smells.
[32] [33] [34]	Limited developer knowledge of code smells, especially for junior devs.	Technical debt.
[54]	The priority of the type of code smell is not yet clear.	It can be difficult for developers to identify and prioritize code smells, leading to technical debt and decreased software quality.
[62]	Unclear threshold.	It can lead to differing results and degraded performance in code smell detection.

F. THE CHALLENGES OF CODE SMELL DETECTION

The primary investigations documented several problems. Several factors are associated with the code smell idea, field conditions, tool restrictions, and particular contexts. These factors are summarized in Table 9, which provides an overview of the issues identified in the primary investigations.

Major studies in code smell detection identify several challenges faced in this endeavor. First, the need for more documentation by developers is a frequent problem. The documentation of code smells is sometimes less comprehensive than that of defects, posing challenges in locating instances of code smells, particularly in open-source systems where their assessment is typically conducted manually. Furthermore, subjective understanding and low agreement among code smell detectors are important constraints. These challenges can negatively impact the accuracy and reliability of detection results due to varying interpretations between detectors. Applying code smell detection in a limited context is also a concern. This detection must consider contextual factors such as the programming language or development environment to make the results more valid and relevant in everyday use.

Conversely, the presence of an imbalanced dataset in the context of machine learning or deep learning-based detection algorithms can result in suboptimal performance when identifying code smells. This can impact the accuracy and effectiveness of detection. Developers' limited knowledge, especially those who are new, about code smells is also a barrier. This limitation can lead to technical accumulation (technical debt). Moreover, the need for more apparent prioritization in identifying the various code smell types poses additional difficulties. This may result in software degradation and further technical accumulation. The detection results of code smells are also influenced by the requirement for enhanced precision in setting the threshold. The problems above underscore the code smell detection process's intricacy and the necessity for a meticulous strategy and efficient solutions to tackle these challenges.

G. LIMITATION OF THE REVIEW

The inclusion of certain studies may have an impact on the overall completeness of our findings. A specified strategy, thorough search methodology, and numerous databases were employed to mitigate this potential danger. In the research search and selection process, we employed forward and backward snowballing techniques to ensure the inclusion of a comprehensive set of primary studies. The study results were managed, and the utilization of spreadsheet tools and Mendeley software prevented the potential duplication of primary studies. A phased search method was also devised to handle instances of text duplication effectively. During the process of inclusion and exclusion, our scanning was limited to the title, abstract, and keywords of each database. This approach may have included irrelevant, relevant, or unrelated papers in the final list. To ensure the exclusion of irrelevant publications, we incorporated a series of phases for evaluating the eligibility of full-text articles. In the case of untracked pertinent papers, we assumed the associated risk by asserting that the fundamental content of the paper ought to be discernible through the title, abstract, and keywords.

IV. CONCLUSION

The present study included a systematic literature review (SLR) about code smells. Out of the 354 publications identified throughout the search process, 69 primary studies were chosen based on their adherence to the predetermined inclusion and exclusion criteria. These selected studies were published within the time frame spanning from 2013 to July 2022. The main focus of this study was the detection of code smells, which were identified mainly using a machine learning approach and some studies using deep learning. Studies also cover the impact of code smells, refactoring, and interactions with software metrics and security aspects. Active and influential researchers in this field are identified. Various detection methods have been identified, such as competitive coevolutionary search and multi-objective genetic programming. Studies also compare different detection tools. Challenges in this study include subjective understanding and low agreement between detectors, applicability to limited contexts and unbalanced datasets, limited developer knowledge, unclear priorities, and unclear thresholds. This demonstrates the complexity of detecting code smells and the need for more sophisticated approaches and tools.

REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, 1st ed. Boston, MA, USA: Addison-Wesley Professional PTG, 2012.

[2] K. Z. Sultana, Z. Codabux, and B. Williams, "Examining the relationship of code and architectural smells with software vulnerabilities," Oct. 2020, *arXiv:2010.15978*.

[3] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems: An empirical study combining quantitative and qualitative data," *Empirical Softw. Eng.*, vol. 19, no. 4, pp. 1111–1143, Aug. 2014, doi: 10.1007/s10664-013-9250-3.

[4] 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 *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 403–414, doi: 10.1109/ICSE.2015.59.

[5] Z. Codabux, K. Z. Sultana, and B. J. Williams, "The relationship between code smells and traceable patterns—Are they measuring the same thing?" *Int. J. Softw. Eng. Knowl. Eng.*, vol. 27, no. 9–10, pp. 1529–1547, Nov. 2017, doi: 10.1142/s0218194017400095.

[6] Z. Codabux, K. Z. Sultana, and B. Williams, "The relationship between traceable code patterns and code smells," in *Proc. SEKE*, Jul. 2017, pp. 444–449, doi: 10.18293/SEKE2017-121.

[7] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 164–175, doi: 10.1109/ICSE.2019.00033.

[8] E. Amer and S. El-Sappagh, "Robust deep learning early alarm prediction model based on the behavioural smell for Android malware," *Comput. Secur.*, vol. 116, May 2022, Art. no. 102670, doi: 10.1016/j.cose.2022.102670.

[9] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, Jun. 2016, doi: 10.1007/s10664-015-9378-4.

[10] A. Barbez, F. Khomh, and Y.-G. Guéhéneuc, "A machine-learning based ensemble method for anti-patterns detection," *J. Syst. Softw.*, vol. 161, Mar. 2020, Art. no. 110486, doi: 10.1016/j.jss.2019.110486.

[11] M. Mantyla, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Proc. Int. Conf. Softw. Maintenance (ICSM).*, 2003, pp. 381–384, doi: 10.1109/icsm.2003.1235447.

[12] M. S. Haque, J. Carver, and T. Atkison, "Causes, impacts, and detection approaches of code smell: A survey," in *Proc. ACMSE Conf.*, Mar. 2018, pp. 1–8, doi: 10.1145/3190645.3190697.

[13] A. Gupta, B. Suri, and S. Misra, "A systematic literature review: Code bad smells in Java source code," in *Proc. Int. Conf. Comput. Sci. Appl.*, 2017, pp. 665–682, doi: 10.1007/978-3-319-62404-4_49.

[14] M. Agnihotri and A. Chug, "A systematic literature survey of software metrics, code smells and refactoring techniques," *J. Inf. Process. Syst.*, vol. 16, no. 4, pp. 915–934, Aug. 2020, doi: 10.3745/JIPS.04.0184.

[15] J. P. dos Reis, F. B. E. Abreu, G. de Figueiredo Carneiro, and C. Anslow, "Code smells detection and visualization: A systematic literature review," *Arch. Comput. Methods Eng.*, vol. 29, no. 1, pp. 47–94, Jan. 2022, doi: 10.1007/s11831-021-09566-x.

[16] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proc. 20th Int. Conf. Eval. Assessment Softw. Eng.*, Jun. 2016, pp. 1–12, doi: 10.1145/2915970.2915984.

[17] X. Liu and C. Zhang. (2017). *The Detection of Code Smell on Software Development: A Mapping Study*. [Online]. Available: http://checkstyle.sourceforge.net

[18] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *J. Softw., Evol. Process*, vol. 27, no. 11, pp. 867–895, Nov. 01, 2015, doi: 10.1002/smr.1737.

[19] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Inf. Softw. Technol.*, vol. 108, pp. 115–138, Apr. 2019, doi: 10.1016/j.infsof.2018.12.009.

[20] A. Al-Shaaby, H. Aljamaan, and M. Alshayeb, "Bad smell detection using machine learning techniques: A systematic literature review," *Arabian J. Sci. Eng.*, vol. 45, no. 4, pp. 2341–2369, Apr. 1, 2020, doi: 10.1007/s13369-019-04311-w.

[21] T. Lewowski and L. Madeyski, "Code smells detection using artificial intelligence techniques: A business-driven systematic review," in *Developments in Information & Knowledge Management for Business Applications* (Studies in Systems, Decision and Control). Cham, Switzerland: Springer, 2022, pp. 285–319, doi: 10.1007/978-3-030-77916-0_12.

[22] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. D. Nascimento, M. F. Freitas, and M. G. D. Mendonça, "A systematic review on the code smell effect," *J. Syst. Softw.*, vol. 144, pp. 450–477, Oct. 2018, doi: 10.1016/j.jss.2018.07.035.

[23] T. Lewowski and L. Madeyski, "How far are we from reproducible research on code smell detection? A systematic literature review," *Inf. Softw. Technol.*, vol. 144, Apr. 2022, Art. no. 106783, doi: 10.1016/j.infsof.2021.106783.

[24] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *J. Syst. Softw.*, vol. 86, no. 10, pp. 2639–2653, Oct. 2013, doi: 10.1016/j.jss.2013.05.007.

[25] J. Padilha, J. Pereira, E. Figueiredo, J. Almeida, A. Garcia, and C. Sant'Anna, "On the effectiveness of concern metrics to detect code smells: An empirical study," in *Proc. Int. Conf. Adv. Inf. Syst. Eng.*, 2014, pp. 656–671.

[26] F. Ponce, J. Soldani, H. Astudillo, and A. Brogi, "Smells and refactorings for microservices security: A multivocal literature review," 2021, *arXiv:2104.13303*.

[27] I. Inayat, S. S. Salim, S. Marczak, M. Daneva, and S. Shamshirband, "A systematic literature review on agile requirements engineering practices and challenges," *Comput. Hum. Behav.*, vol. 51, pp. 915–929, Oct. 2015, doi: 10.1016/j.chb.2014.10.046.

[28] D. Badampudi, C. Wohlin, and K. Petersen, "Experiences from using snowballing and database searches in systematic literature studies," in *Proc. 19th Int. Conf. Eval. Assessment Softw. Eng.*, New York, NY, USA, Apr. 2015, pp. 1–10, doi: 10.1145/2745802.2745818.

[29] E.-M. Schön, J. Thomaschewski, and M. J. Escalona, "Agile requirements engineering: A systematic literature review," *Comput. Standards Interfaces*, vol. 49, pp. 79–91, Jan. 2017, doi: 10.1016/j.csi.2016.08.011.

[30] I. K. Raharjana, D. Siahaan, and C. Fatichah, "User stories and natural language processing: A systematic literature review," *IEEE Access*, vol. 9, pp. 53811–53826, 2021, doi: 10.1109/access.2021.3070606.

[31] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, pp. 1–31, Jan. 2021, doi: 10.1145/3408897.

[32] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, ''Why don't software developers use static analysis tools to find bugs?'' in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 672–681, doi: 10.1109/ICSE.2013.6606613.

[33] A. Yamashita and L. Moonen, ''Do developers care about code smells? An exploratory survey,'' in *Proc. 20th Work. Conf. Reverse Eng. (WCRE)*, Oct. 2013, pp. 242–251, doi: 10.1109/WCRE.2013.6671299.

[34] D. Taibi, A. Janes, and V. Lenarduzzi, ''How developers perceive smells in source code: A replicated study,'' *Inf. Softw. Technol.*, vol. 92, pp. 223–235, Dec. 2017, doi: 10.1016/j.infsof.2017.08.008.

[35] A. M. Fard and A. Mesbah, ''JSNOSE: Detecting Javascript code smells,'' in *Proc. IEEE 13th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2013, pp. 116–125, doi: 10.1109/SCAM.2013.6648192.

[36] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol. *An Empirical Study of Code Smells in JavaScript Projects*. Accessed: Aug. 15, 2023. [Online]. Available: http://eslint.org/docs/rules/no-cond-assign

[37] I. Shoenberger, M. W. Mkaouer, and M. Kessentini, ''On the use of smelly examples to detect code smells in Javascript,'' in *Proc. Eur. Conf. Appl. Evol. Comput.* Lecture Notes in Computer Science: Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics. Cham, Switzerland: Springer, 2017, pp. 20–34, doi: 10.1007/978-3-319-55792-2_2.

[38] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, ''Understanding code smells in Android applications,'' in *Proc. IEEE/ACM Int. Conf. Mobile Softw. Eng. Syst. (MOBILESoft)*, May 2016, pp. 225–236, doi: 10.1145/2897073.2897094.

[39] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, ''On the impact of code smells on the energy consumption of mobile applications,'' *Inf. Softw. Technol.*, vol. 105, pp. 43–55, Jan. 2019, doi: 10.1016/j.infsof.2018.08.004.

[40] M. Boussaa, W. Kessentini, and S. Bechikh. (2013). *Competitive Coevolutionary Code-Smells Detection Article View Project Bi-Level Optimization View Project*. [Online]. Available: https://www.researchgate.net/publication/305215765

[41] D. Sahin. *Code-Smells Detection as a Bi-Level Problem*. Accessed: Aug. 15, 2023. [Online]. Available: http://www.egr.msu.edu/~kdeb/COIN.shtml

[42] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, ''Multi-objective code-smells detection using good and bad design examples,'' *Softw. Qual. J.*, vol. 25, no. 2, pp. 529–552, Jun. 2017, doi: 10.1007/s11219-016-9309-7.

[43] A. Barbez, F. Khomh, and Y.-G. Guéhéneuc, ''Deep learning anti-patterns from code metrics history,'' in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2019, pp. 114–124, doi: 10.1109/ICSME.2019.00021.

[44] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, ''Security smells in ansible and chef scripts,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, pp. 1–31, Jan. 2021, doi: 10.1145/3408897.

[45] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, ''A textual-based technique for smell detection,'' in *Proc. IEEE 24th Int. Conf. Program Comprehension (ICPC)*, May 2016, pp. 1–10, doi: 10.1109/ICPC.2016.7503704.

[46] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, ''The scent of a smell: An extensive comparison between textual and structural smells,'' *IEEE Trans. Softw. Eng.*, vol. 44, no. 10, pp. 977–1000, Oct. 2018, doi: 10.1109/TSE.2017.2752171.

[47] G. Rasool and Z. Arshad, ''A lightweight approach for detection of code smells,'' *Arabian J. Sci. Eng.*, vol. 42, no. 2, pp. 483–506, Feb. 2017, doi: 10.1007/s13369-016-2238-8.

[48] S. Fu and B. Shen, ''Code bad smell detection through evolutionary data mining,'' in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Oct. 2015, pp. 1–9, doi: 10.1109/ESEM.2015.7321194.

[49] F. Palomba, R. Oliveto, and A. De Lucia, ''Investigating code smell co-occurrences using association rule learning: A replicated study,'' in *Proc. IEEE Workshop Mach. Learn. Techn. Softw. Quality Eval. (MaLTeSQuE)*, Feb. 2017, pp. 8–13, doi: 10.1109/MALTESQUE.2017.7882010.

[50] N. Pritam, M. Khari, L. H. Son, R. Kumar, S. Jha, I. Priyadarshini, M. Abdel-Basset, and H. V. Long, ''Assessment of code smell for predicting class change proneness using machine learning,'' *IEEE Access*, vol. 7, pp. 37414–37425, 2019, doi: 10.1109/ACCESS.2019.2905133.

[51] M. M. Draz, M. S. Farhan, S. N. Abdulkader, and M. G. Gafar, ''Code smell detection using whale optimization algorithm,'' *Comput., Mater. Continua*, vol. 68, no. 2, pp. 1919–1935, Apr. 2021, doi: 10.32604/cmc.2021.015586.

[52] A. K. Das, S. Yadav, and S. Dhal, ''Detecting code smells using deep learning,'' in *Proc. IEEE Region Conf. (TENCON)*, Oct. 2019, pp. 2081–2086, doi: 10.1109/TENCON.2019.8929628.

[53] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, ''Deep learning based code smell detection,'' *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1811–1837, Sep. 2021, doi: 10.1109/TSE.2019.2936376.

[54] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi, ''JSpIRIT: A flexible tool for the analysis of code smells,'' in *Proc. 34th Int. Conf. Chilean Comput. Sci. Soc. (SCCC)*, Nov. 2015, pp. 1–6, doi: 10.1109/SCCC.2015.7416572.

[55] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, ''Lightweight detection of android-specific code smells: The aDoctor project,'' in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2017, pp. 487–491, doi: 10.1109/SANER.2017.7884659.

[56] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, M. Makhlouf, and L. B. Said, ''Code smell detection and identification in imbalanced environments,'' *Expert Syst. Appl.*, vol. 166, Mar. 2021, Art. no. 114076, doi: 10.1016/j.eswa.2020.114076.

[57] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, ''A large empirical assessment of the role of data balancing in machine-learning-based code smell detection,'' *J. Syst. Softw.*, vol. 169, Nov. 2020, Art. no. 110693.

[58] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, ''Comparing heuristic and machine learning approaches for metric-based code smell detection,'' in *Proc. IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC)*, May 2019, pp. 93–104, doi: 10.1109/ICPC.2019.00023.

[59] V. Lenarduzzi, N. Saarimäki, and D. Taibi, ''Some SonarQube issues have a significant but small effect on faults and changes. A large-scale empirical study,'' *J. Syst. Softw.*, vol. 170, Dec. 2020, Art. no. 110750, doi: 10.1016/j.jss.2020.110750.

[60] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. B. Chikha, ''Competitive coevolutionary code-smells detection,'' in *Proc. Int. Symp. Search Based Softw. Eng.* Lecture Notes in Computer Science: Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics, vol. 8084, 2013, pp. 50–65, doi: 10.1007/978-3-642-39742-4_6.

[61] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, ''A cooperative parallel search-based software engineering approach for code-smells detection,'' *IEEE Trans. Softw. Eng.*, vol. 40, no. 9, pp. 841–861, Sep. 2014, doi: 10.1109/TSE.2014.2331057.

[62] H. Liu, Q. Liu, Z. Niu, and Y. Liu, ''Dynamic and automatic feedback-based threshold adaptation for code smell detection,'' *IEEE Trans. Softw. Eng.*, vol. 42, no. 6, pp. 544–558, Jun. 2016, doi: 10.1109/TSE.2015.2503740.

[63] A. Kaur, S. Jain, and S. Goel, ''SP-J48: A novel optimization and machine-learning-based approach for solving complex problems: Special application in software engineering for detecting code smells,'' *Neural Comput. Appl.*, vol. 32, no. 11, pp. 7009–7027, Jun. 2020, doi: 10.1007/s00521-019-04175-z.

[64] M. De Stefano, F. Pecorelli, F. Palomba, and A. De Lucia, ''Comparing within- and cross-project machine learning algorithms for code smell detection,'' in *Proc. 5th Int. Workshop Mach. Learn. Techn. Softw. Quality Evol.*, Aug. 2021, pp. 1–6, doi: 10.1145/3472674.3473978.

[65] S. Jain and A. Saha, ''Rank-based univariate feature selection methods on machine learning classifiers for code smell detection,'' *Evol. Intell.*, vol. 15, no. 1, pp. 609–638, Mar. 2022, doi: 10.1007/s12065-020-00536-z.

[66] N. Almarimi, A. Ouni, and M. W. Mkaouer, ''Learning to detect community smells in open source software projects,'' *Knowl.-Based Syst.*, vol. 204, Sep. 2020, Art. no. 106201, doi: 10.1016/j.knosys.2020.106201.

[67] T. Guggulothu and S. A. Moiz, ''Code smell detection using multi-label classification approach,'' *Softw. Quality J.*, vol. 28, no. 3, pp. 1063–1086, Sep. 2020, doi: 10.1007/s11219-020-09498-y.

[68] A. Alazba and H. Aljamaan, ''Code smell detection using feature selection and stacking ensemble: An empirical investigation,'' *Inf. Softw. Technol.*, vol. 138, Oct. 2021, Art. no. 106648, doi: 10.1016/j.infsof.2021.106648.

[69] M. De Stefano, F. Pecorelli, F. Palomba, and A. De Lucia, ''Comparing within-and cross-project machine learning algorithms for code smell detection,'' in *Proc. 5th Int. Workshop Mach. Learn. Techn. Softw. Qual. Evol., Co-Located With ESEC/FSE*, New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 1–6, doi: 10.1145/3472674.3473978.

[70] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, and G. Sladić, ''Automatic detection of long method and god class code smells through neural source code embeddings,'' *Expert Syst. Appl.*, vol. 204, Oct. 2022, Art. no. 117607, doi: 10.1016/j.eswa.2022.117607.

[71] H. Grodzicka, A. Ziobrowski, Z. Łakomiak, M. Kawa, and L. Madeyski, ''Code smell prediction employing machine learning meets emerging Java language constructs,'' in *Data-Centric Business and Applications* (Lecture Notes on Data Engineering and Communications Technologies). Cham, Switzerland: Springer, 2020, pp. 137–167, doi: 10.1007/978-3-030-34706-2_8.

[72] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, ''Quantifying the effect of code smells on maintenance effort,'' *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

[73] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, ''Investigating the energy impact of Android smells,'' in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2017, pp. 115–126, doi: 10.1109/SANER.2017.7884614.

[74] F. Hermans and E. Aivaloglou, ''Do code smells hamper novice programming? A controlled experiment on scratch programs,'' in *Proc. IEEE 24th Int. Conf. Program Comprehension (ICPC)*, May 2016, pp. 1–10, doi: 10.1109/ICPC.2016.7503706.

[75] Z. Soh, A. Yamashita, F. Khomh, and Y.-G. Guéhéneuc, ''Do code smells impact the effort of different maintenance programming activities?'' in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, vol. 1, Mar. 2016, pp. 393–402, doi: 10.1109/SANER.2016.103.

[76] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, ''Smells like teen spirit: Improving bug prediction performance using the intensity of code smells,'' in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2016, pp. 244–255, doi: 10.1109/ICSME.2016.27.

[77] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, ''On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation,'' *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1188–1221, Jun. 2018, doi: 10.1007/s10664-017-9535-z.

[78] F. Palomba, D. A. Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, ''Beyond technical aspects: How do community smells influence the intensity of code smells?'' *IEEE Trans. Softw. Eng.*, vol. 47, no. 1, pp. 108–129, Jan. 2021, doi: 10.1109/TSE.2018.2883603.

[79] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka, ''Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains,'' in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2013, pp. 260–269, doi: 10.1109/ICSM.2013.37.

[80] T. Hall, M. Zhang, D. Bowes, and Y. Sun, ''Some code smells have a significant but small effect on faults,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 1–39, Sep. 2014, doi: 10.1145/2629648.

[81] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, ''Automatic metric thresholds derivation for code smell detection,'' in *Proc. IEEE/ACM 6th Int. Workshop Emerg. Trends Softw. Metrics*, May 2015, pp. 44–53, doi: 10.1109/WETSoM.2015.14.

[82] S. Kaur and R. Maini. (2016). *Analysis of Various Software Metrics Used To Detect Bad Smells*. [Online]. Available: https://www.theijes.com

[83] I. Pigazzini, F. A. Fontana, and B. Walter, ''A study on correlations between architectural smells and design patterns,'' *J. Syst. Softw.*, vol. 178, Aug. 2021, Art. no. 110984, doi: 10.1016/j.jss.2021.110984.

[84] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, and D. Poshyvanyk, ''When and why your code starts to smell bad (and whether the smells go away),'' *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1063–1088, Nov. 2017, doi: 10.1109/TSE.2017.2653105.

[85] B. Walter, F. A. Fontana, and V. Ferme, ''Code smells and their collocations: A large-scale experiment on open-source systems,'' *J. Syst. Softw.*, vol. 144, pp. 1–21, Oct. 2018, doi: 10.1016/j.jss.2018.05.057.

[86] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, ''Toward a smell-aware bug prediction model,'' *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 194–218, Feb. 2019, doi: 10.1109/TSE.2017.2770122.

[87] H. Liu, X. Guo, and W. Shao, ''Monitor-based instant software refactoring,'' *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1112–1126, Aug. 2013, doi: 10.1109/TSE.2013.4.

[88] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, ''Multi-criteria code refactoring using search-based software engineering: An industrial case study,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, pp. 1–53, Aug. 2016.

[89] A. Chatzigeorgiou and A. Manakos, ''Investigating the evolution of code smells in object-oriented systems,'' *Innov. Syst. Softw. Eng.*, vol. 10, no. 1, pp. 3–18, Mar. 2014, doi: 10.1007/s11334-013-0205-z.

[90] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen, ''Code smells for model-view-controller architectures,'' *Empirical Softw. Eng.*, vol. 23, no. 4, pp. 2121–2157, Aug. 2018, doi: 10.1007/s10664-017-9540-2.

[91] M. K. Sarker, A. Al Jubaer, M. S. Shohrawardi, T. C. Das, and M. S. Siddik, ''Analysing GoLang projects' architecture using code metrics and code smell,'' in *Proc. 1st Int. Workshop Intell. Softw. Automat.*, 2021, pp. 53–63, doi: 10.1007/978-981-16-1045-5_5.

[92] T. Sharma, M. Fragkoulis, and D. Spinellis, ''Does your configuration code smell?'' in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories (MSR)*. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 189–200.

[93] V. Garousi and B. Küçük, ''Smells in software test code: A survey of knowledge in industry and academia,'' *J. Syst. Softw.*, vol. 138, pp. 52–81, Apr. 2018, doi: 10.1016/j.jss.2017.12.013.

[94] R. M. de Mello, A. G. Uchoa, R. F. Oliveira, D. T. M. de Oliveira, B. Fonseca, A. F. Garcia, and F. de Barcellos de Mello, ''Investigating the social representations of code smell identification: A preliminary study,'' in *Proc. IEEE/ACM 12th Int. Workshop Cooperat. Hum. Aspects Softw. Eng. (CHASE)*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, May 2019, pp. 53–60, doi: 10.1109/CHASE.2019.00022.

[95] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, ''Developer-driven code smell prioritization,'' in *Proc. IEEE/ACM 17th Int. Conf. Mining Softw. Repositories (MSR)*, New York, NY, USA, May 2020, pp. 220–231, doi: 10.1145/3379597.3387457.

MUHAMMAD ANIS AL HILMI received the master's degree in electrical and informatics engineering from the Bandung Institute of Technology, in 2015. He is currently an Assistant Professor with the Department of Informatics, Politeknik Negeri Indramayu. He has published journal articles and conference papers related to information security and IT. His research interests include information security and software engineering. He holds a patent and two industrial designs.

ALIFIA PUSPANINGRUM received the master's degree in software engineering from Institut Teknologi Sepuluh Nopember, in 2018. She is currently an Assistant Professor with the Department of Informatics, Politeknik Negeri Indramayu. She has published journal articles and conference papers related to software engineering and artificial intelligence. Her research interests include software engineering and artificial intelligence. She holds an intellectual property right in web application.

DARSIH received the master's degree in information system from Universitas Diponegoro, in 2014. She is currently an Assistant Professor with the Department of Informatics, Politeknik Negeri Indramayu. She has published journal articles and conference papers related to information systems. Her research interests include software engineering and information systems. She holds an intellectual property right in web application.

DANIEL ORANOVA SIAHAAN (Member, IEEE) received the master's degree in software engineering from Technische Universiteit Delft, in 2002, and the Ph.D. degree in software engineering from Technische Universiteit Eindhoven, in 2004. He is currently a Professor with the Department of Informatics, Institut Teknologi Sepuluh Nopember. He has published more than 50 journal articles and conference papers related to software engineering. His research interests include software engineering and requirements engineering. He is also a Society Member of IEEE.

HERNAWATI SUSANTI SAMOSIR (Member, IEEE) received the master's degree in software engineering from Institut Teknologi Sepuluh Nopember, in 2018. She is currently an Assistant Professor with Institut Teknologi Del. She has published journal articles and conference papers related to software engineering. Her research interest includes software engineering.

AMELIA SAHIRA RAHMA received the master's degree in software engineering from Institut Teknologi Sepuluh Nopember, in 2018. She has published journal articles and conference papers related to software engineering. Her research interest includes software engineering.