

A systematic review on software code smells

Mohammed Ghazi Al-Obeidallah¹, Dimah Al-Fraihat²

¹Department of Computer Science and Information Technology, Abu Dhabi University, Abu Dhabi, United Arab Emirates

²Department of Software Engineering, Faculty of Information Technology, Isra University, Amman, Jordan

Article Info

Article history:

Received Mar 28, 2024

Revised Jan 16, 2025

Accepted Mar 3, 2025

Keywords:

Antipatterns

Code refactoring

Code smells

Smell detection

Systematic review

ABSTRACT

This paper provides a systematic review of code smell detection studies published from 2001 to 2023, addressing their significance in identifying underlying issues in software systems. Through stringent inclusion criteria, 116 primary studies were analyzed, focusing on various aspects such as publication venue, code smell categories, subject systems, supported programming languages, evaluation criteria, and detection techniques. The analysis reveals that 50% of the papers were conference proceedings, with 80% utilizing Java-supported techniques and commonly used subject systems like Apache Xerces, GanttProject, and ArgoUML. Metrics-based methods (33%) and search-based approaches (32%) were predominantly employed, with machine learning emerging in 20% and rule-based methods in 15% of the studies. Notably, recent studies have shown an increased adoption of machine learning techniques. The identified code smells include god class, feature envy, long method, and data class, with precision and recall being the most commonly used evaluation metrics. This review aims to inform future research directions and aid the software engineering community in developing novel detection techniques to enhance code quality and system reliability.

This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.



Corresponding Author:

Dimah Al-Fraihat

Department of Software Engineering, Faculty of Information Technology, Isra University

11622, Amman, Jordan

Email: d.fraihat@iu.edu.jo

1. INTRODUCTION

Code smells are structures in the code that indicate violation of design rules. Code smells are not preventing the program from providing its required functionality. However, they indicate weaknesses in software design. Code smells may increase the risk of software failure. The existence of code smells may give a sign about wider design problems, and they affect the software understandability and effectiveness [1]. Since code smells play a key role in shaping the quality of software, several studies were presented to detect their presence in the source code.

Fowler *et al.* [2] listed 22 code smells classified into seven key categories. These categories are bloaters, object-oriented abusers, change preventers, dispensable, couplers, encapsulates, and general category. For example, the size and volume of the smell in bloaters can be unmanageable and out of control. This smell would appear in a method with too many lines of code which has a higher complexity than short methods. Bloaters would appear also in large classes which involve lots of fields and methods. Additionally, it would appear in methods with long parameter list which indicates a sign of code smell.

In contrast, object-oriented abusers involve different bad use of object-oriented structures to develop software programs. For example, the use of switch statement will lead to code duplication. In addition, the use of a temporary field will create problems since the reference of variables cannot be accessed outside its

scope. Avoiding the use of long parameter lists leads to the use of temporary variables. Another popular abuse of object-oriented classes is the “Refuse Bequest” where a sub class partially inherits its parent. Hence, part of the parent methods will be used.

Change preventers code smells make programs difficult to change. Hence, maintenance tasks will be more difficult. The key reason for change prevention is the assignment of more than one function/feature to a class or method. One of the most popular forms of change preventers is divergent change which occurs when a class involves a lot of irrelevant methods. Another code smell related to change preventers is the shotgun surgery where cascading relationship used to connect classes together. Any change in one class will propagate to affect all classes in the same chain.

Conversely, dispensable code smells are unnecessary code that must be removed. For example, code may contain useless classes (lazy class) or unnecessary data fields in a class (data class). Data classes are unable to use their fields and require other classes to use these classes. Another form of dispensable is the code duplication where similar code is repeated several times. This code can be replaced or unified.

Several studies were presented in the literature to address the issues related to various types of code smells. These issues concern code smell detection techniques, code smell impact, code smell causes, and code smell catalogue. Hence, there is a need to appraise and summarize the collective findings and identify the latest developments. While acknowledging the valuable review studies conducted by previous researchers, it should be noted that this study follows a systematic review methodology and employs a rigorous selection and analysis process focusing specifically on the years 2001 to 2023. The timeframe chosen for the review reflects the intention to capture the most recent developments in the field of code smell detection, acknowledging the fast-paced nature of research advances during this period. The objective of this study is to complement previous review research by examining studies conducted in subsequent years and extending the analysis to include the period 2001 to 2023, thus highlighting recent advances and emerging trends in code smell techniques. To the best of the authors’ knowledge, this is the first systematic review paper on code smell detection to cover the years 2001 to 2023. We retrieved studies from reputable publishing venues and databases, and analyze them to answer the following research questions:

RQ1: What is the distribution of studies per year in the context of the code smell detection?

RQ2: What are the categories of publications included in this review research?

RQ3: What is the distribution of the studies included in this review, categorized by the employed code smell detection technique?

RQ4: What programming languages are employed by various code smell detection techniques in the included studies?

RQ5: What subject systems were used to validate code smell detection techniques in the included studies?

RQ6: What code smells were detected by various techniques in the included studies?

RQ7: What evaluation criteria were utilized by code smell detection techniques in the reviewed studies?

We believe that summarizing literature in the code smell detection field will open new research directions and will help the software engineering community to address the necessity of utilizing new detection techniques. The rest of the paper is organized as follows: section 2 presents previous work in surveying code smells and related literature. Section 3 provides an overview of the topic, highlights the impacts of code smells and provides an overview of code smell detection techniques. Research methodology is presented in section 4; the analysis and results are presented in section 5. The paper concludes in section 6. Finally, avenues for future work are suggested in section 7.

2. RELATED WORK

Several systematic literature reviews were presented in the literature. All these reviews focused on comparing code smell detection approaches in terms of different criteria. Our findings in this review are comprehensive, consistent, and complement the conclusions presented by other studies.

The survey study conducted by Roy and Cordy [3] presented the state of the art in clone detection research. The survey provides a view of the existing clone taxonomies, detection approaches and experimental evaluations of clone detection tools. Additionally, Fontana *et al.* [4] presented an experimental evaluation of six code smell detection tools. The key differences between the addressed tools were highlighted in the evaluation.

The review of Rasool and Arshad [5] presented an up-to-date review on the state-of-the-art techniques and tools used for mining code smells from the source code of different software applications. They classified selected code smell detection techniques and tools based on their detection methods and analyzed the results of the selected techniques. Rasool and Arshad focused on Fowler’s 22 code smells [2], [5]. Sabir *et al.* [6] reviewed 78 primary studies published from January 2000 till December 2017. The review of Sabir *et al.* focused on the code smell detection techniques used in object and service- oriented paradigms. Zhang *et al.* [7] reviewed 39 studies related to code smells detection. Their research focused on

“Duplicated Code” whereas some code bad smells such as “Message Chains” received little attention. Zhang *et al.* study showed that very few studies report the impact of using code bad smells. Instead, most studies focused on developing tools and methods to automatically detect code bad smells.

The literature review of Sobrinho *et al.* [8] showed that even though bad smells of different types are studied together, only a small number of studies explored the relations between them. They suggested that there are additional potential relations that warrant further investigation. Sobrinho *et al.* also noted that researchers have different levels of interest, some of them publishing sporadically and others continuously. Further, the review of Sobrinho *et al.* found that the communities studying duplicated code and other types of bad smells are largely separated. Finally, Sobrinho *et al.* observed that some venues are more likely to disseminate knowledge on duplicate code (which often is listed as a conference topic on its own), while others have a more balanced distribution among other smells.

The study of Misbhaudhin and Alshayeb [9] provided an overview of existing research in the field of model refactoring. A total of 3,295 articles, related to the field of unified modeling language (UML) model refactoring, were extracted from well-known electronic databases. A multi-stage selection process was used to ensure proper inclusion of relevant studies for review and analysis. Ninety-four primary studies were eventually selected and analyzed. The results showed that a few quality techniques and approaches have been proposed in this area, but it still has some important open issues and limitations to be addressed in future.

AlDallal [10] focused on the identification of refactoring opportunities where more attention refactoring opportunities are explored. Their results showed “extract class” and “move method” were found to be the most frequently considered refactoring activities. The results show that researchers use six primary existing approaches to identify refactoring opportunities and six approaches to empirically evaluate the identification techniques.

Azeem *et al.* [11] presented a systematic literature review study on machine learning techniques for code smell detection. Azeem *et al.* analyzed papers published between 2000 and 2017. The results showed that god class, long method, functional decomposition, and spaghetti code have been heavily considered in the literature. Decision trees and support vector machines are the most commonly used machine learning algorithms for code smell detection.

The review paper presented by AbuHassan *et al.* [12] identified 145 studies related to smell detection in software design and code. AbuHassan *et al.* addressed several questions related to the analysis of the existing smell detection techniques in terms of abstraction level (design or code), targeted smells, used metrics, implementation, and validation. The results showed that 57% of the studies did not use any performance measures, 41% of them omitted details on the targeted programming language, and the detection techniques were not validated in 14% of these studies.

The study of Reis *et al.* [13] aimed to identify the main code smells detection techniques and tools discussed in the literature, and to analyze to which extent visual techniques have been applied to support code smell detection. The results showed that most used approaches to code smells detection are search-based (30.1%), metric-based (24.1%), and symptom-based approaches (19.3%). Most of the reviewed studies (83.1%) used open-source software, with the java language occupying the first position (77.1%). In terms of code smells, god class (51.8%), feature envy (33.7%), and long method (26.5%) are the most covered ones.

3. BACKGROUND

This section provides comprehensive research background related to code smells and presents the causes of code smells, impact of code smells, and key concepts relevant to the research topic. Researchers focused on investigating the consequences of code smells and how these code smells can be detected to address their impact on software quality. Based on the available literature, the key causes of code smells can be summarized as follows:

- a. Design patterns impact: the relationship between design patterns and code smells is still not well investigated. The implementation of design pattern instances in the source code leads to more coupled classes and increase the number of classes. Some studies suggest that design patterns, in general, reduce the chance of code smells.
- b. Software developers experience: the skills and the experience of software developers play a role in implementing a program that suffers from code smells. The study of [14] reveals that 32% of software developers are aware about code smells.
- c. Lack of code smell detection tools during software development: most of software development tools and frameworks are not supporting automatic code smell detection during the development process. The study of [14] and [15] reveal that the developers focus on the functionality of the software and ignore the signs of code smells during the development process.

Several studies have been presented in the literature to investigate the impact of code smells on software programs [16]–[25]. These studies mainly summarized the impact of code smell into two key consequences. The first consequence is the increase of code bugs or defects in the software. Cloning or duplicating code increases the chances of software bugs [17]. For example, the study of [23] suggested that code duplication did not play a key role in producing bugs in the software. The study also claim that code cloning does not develop new bugs. The study of Li and Shatnawi [20] identified the relationship between code smells and class error probability. The study concluded that shotgun surgery, god class, and god methods have a negative impact on the class error probability.

Maintaining software involves efforts to add new functions or features and modifying the code to fulfil new requirements. The study of [16] showed how the lack of design heuristics affects software maintainability. The study also compared between two versions of an implementation (with god class smell and without god class smell). The study concluded that high cohesion and low coupling in software programs affect the maintainability efforts.

The second impact is the increase in the maintenance effort. Several hypotheses were introduced to relate the impact of code smells with software changeability. The continuous modifications of the program increase the code smell. Smell classes evolved more frequently compared to non-smell classes. The study of [17] investigated the impact of code smells on software change proneness. The study found that long class and Message Chain are the most frequent smells in several releases of Eclipse and Azureus, and each new release introduces some new smells while removing the older smells.

The study of [24] presented an empirical study that investigated inter-smell relations and their effects on the incidence of maintenance problems. By analyzing how professional developers conducted tasks on four different systems, the study found empirical evidence that certain inter-smell relations were associated with problems during maintenance. Another study of [25] investigates the effects of code smells at the activity level. Six professional developers were hired to perform three maintenance tasks on four functionally equivalent Java Systems. Each developer performs two maintenance tasks. The logs of the developers were traced, and an annotation approach was defined to assess if code smells impact maintenance activities. The study showed that different code smells affect different activity effort. The techniques used to detect code smells in the literature [3]–[12], [13] can be grouped into four key categories. Code smells detected by different detection approaches are presented in Table 1. The detection techniques are grouped based on the used detection method into four key categories [26], [27]:

- a. Metrics-based approaches use software metrics such as “lines of code”, “coupling” between objects, “cohesion”, and “depth of inheritance tree” to detect code smells based on certain threshold values. The selection of threshold values affects the overall accuracy of the detection process. Software metrics helped to detect the following code smells: refused bequest, data clumps, shotgun surgery, large class, long method, and lazy class. Since there is no unified benchmark to set the threshold values, this approach is still not practical, in terms of accuracy, to detect code smells.
- b. Search-based approaches: these approaches use different search algorithms to detect code smells in the source code. Heuristic search algorithms were used to extract rules that can be used during the search process.
- c. Rule based approaches: rule-based systems convert the problem into a set of condition action rules. If-then rules are widely used to feed an inference engine equipped with a working knowledge about the problem. Software metrics were used to capture knowledge and the available information. Moreover, rule-based approaches describe code smell symptoms and produce rules that will be translated into detection algorithms to identify code smells.
- d. Machine learning approaches: these approaches use a set of predictors for a machine learning classifier to reach a decision. Machine learning approaches depend on the quality of a balanced dataset and on the quality of the training model. For example, to detect code smells, the training model can learn from standard design practices and compared to developer coding practice [17].

Table 1. Categories of code smells

Category	Code smell
Bloaters	Long parameter list, long method, large class, data clumps, and primitive obsession
Object oriented abusers	Temporary fields, switch statement, refused bequest, parallel inheritance hierarchies, and alternative classes with different interfaces
Change preventers	Shotgun surgery, and divergent change
Dispensable	Data class, lazy class, duplicate code, dead code, speculative generality

4. METHOD

This systematic review research aims to conduct a fair and comprehensive evaluation and interpretation of available research published from 2001 to 2023 in the field of code smell detection. To the best of our knowledge, less than 10 studies were published before 2001 that focus on software code smell. The guidelines suggested by [18], [28], and [29]–[31] were followed in undertaking this systematic review. The first phase of the research involved planning the review, which was further divided into determining the necessity for a review, developing the research questions, and describing the search approach to find relevant research papers. The second phase involved conducting the review, which was subdivided into defining the appropriate research selection criteria, including the inclusion/exclusion criteria, developing the quality evaluation rules to filter research publications, constructing the data extraction strategy to address the study objectives, and then synthesizing the data taken from the publications. The third phase of the research was the reporting phase.

4.1. Planning phase

The area of code smell research is experiencing rapid growth, and there are several code smell detection techniques presented in the literature. Therefore, there is a need to summarize the findings and outcomes of previous research and identify the latest developments in the field. The synthesized information will help in the identification of research patterns and the development of statistics that will in turn shed light on limitations and gaps in the literature, in addition to current and future directions of research. Accordingly, this review will ultimately provide answers to the following research questions:

RQ1: What is the distribution of studies per year in the context of the code smell detection?

RQ2: What are the categories of publications included in this review research?

RQ3: What is the distribution of the studies included in this review, categorized by the employed code smell detection technique?

RQ4: What programming languages are employed by various code smell detection techniques in the included studies?

RQ5: What subject systems were used to validate code smell detection techniques in the included studies?

RQ6: What code smells were detected by various techniques in the included studies?

Our search process is comprehensive and covers the whole spectrum related to the area of the code smell detection techniques. The search queries used to retrieve the related primary studies are as follows:

- a. Search Query 1: code AND smells;
- b. Search Query 2: bad AND smells;
- c. Search Query 3: antipattern;
- d. Search Query 4: (detect OR identify OR recover) AND smells.

We applied our adopted search queries to well-known reputable scientific databases. These databases include Springer, ACM, IEEE, science direct, and Google Scholar. Based on the search terms mentioned above and using the specified period (2001 to 2023), 689 publications were initially retrieved. 193 were identified as duplicates among the different libraries and removed. Hence, 496 publications were kept for the next phase.

4.2. Review phase

Different approaches have been introduced by researchers to detect code smells at the source code level. To select the primary studies that will be included in this review, several inclusion and exclusion criteria were identified. The inclusion criteria can be summarized as follows: i) Publications must be published from 2001 till August 2023; ii) Publications must be published in a journal, conference proceeding, book chapter, workshop or symposium; iii) Publications must propose/discuss at least one code smell detection technique; and iv) Existence of practical experiments at the code level.

We excluded the following studies: i) Publications that are written in a language other than English; ii) Publications that their main focus is not code smells detection; iii) Publications that are books or theses; iv) Publications that are published as reports; and v) Publications that propose code smell detection based on the design level not at the code level.

4.3. Analysis phase

The completed compilation of articles selected for comprehensive evaluation underwent a rigorous examination to extract information addressing the previously stated research inquiries. Initially, we started with 689 articles. After removing duplicates and applying title and abstract filters, as well as inclusion/exclusion criteria, we reviewed a total of 140 articles in full. Ultimately, 116 articles met our criteria, specifically focusing on the existence of practical experiments at the code level. A total of 116 primary studies were included in this review. We performed a manual validation of the retrieved studies to

ensure that the retrieved studies related to our proposed inclusion and exclusion criteria. The findings are detailed in the results section, and the steps in this stage are illustrated in Figure 1.

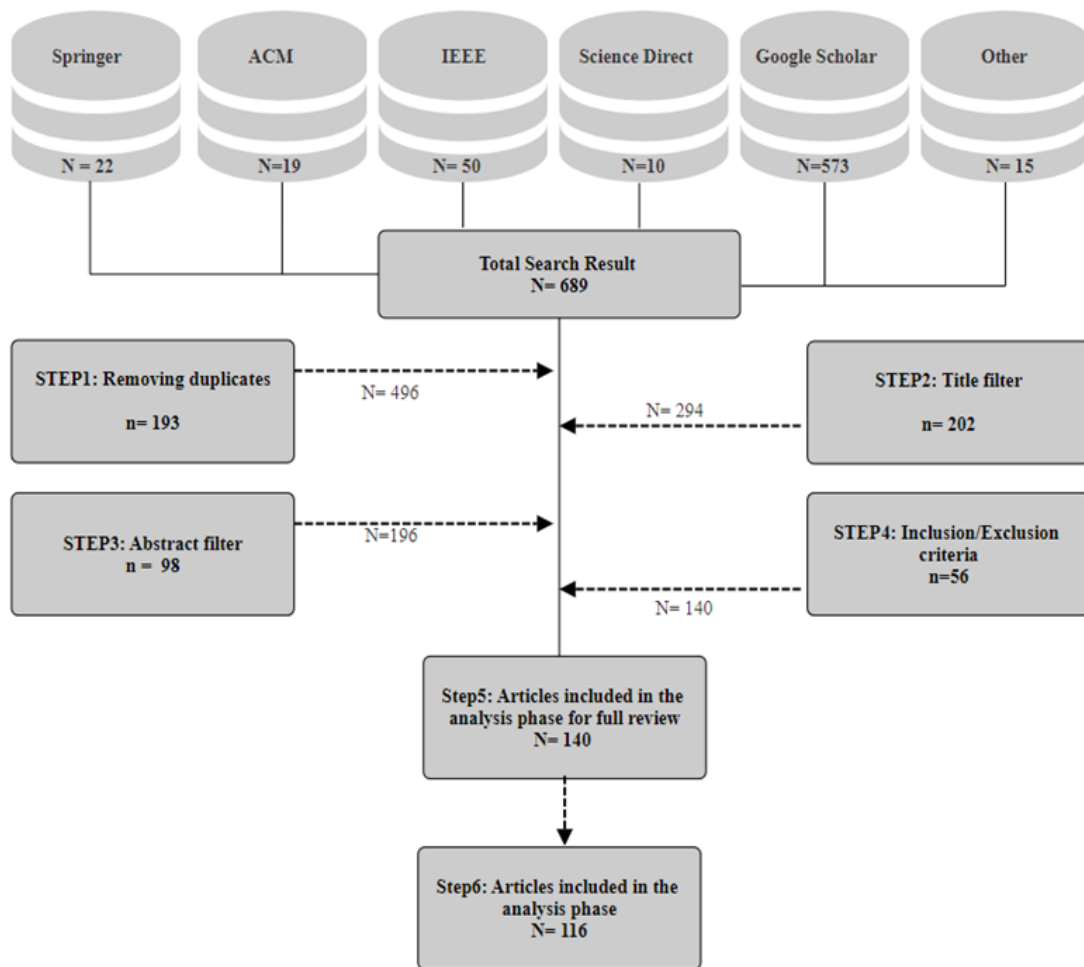


Figure 1. The sequence of actions taken to choose the publications

5. RESULTS AND DISCUSSION

The third stage of this systematic review study involved presenting the results. The final list of research articles comprised a total of 116 publications [32]–[147]. A comprehensive and thorough examination of these papers was conducted to extract relevant information addressing the research questions. The extracted data were quantitatively described, facilitating the identification of patterns in studies conducted between 2001 and 2023. Additionally, the analysis unveiled both commonalities and discrepancies among the studies.

RQ1: What is the distribution of studies per year in the context of the code smell detection?

The retrieved primary studies involve papers from journals, conference proceedings, book chapters, workshops, and symposiums. The number of included primary studies per year is presented in Figure 2. As can be seen from Figure 2, most of the recent publications are Journal papers. In 2017, 11 conference papers were published related to code smell detection. We could not find any papers related to code smell detection published in 2003.

RQ2: What are the categories of publications included in this review research?

Figure 3 shows the type of publications included in this review. As Figure 3 shows, most of the included primary studies are conference proceedings (50% of the primary studies, with a total of 58 studies). We grouped the primary studies published in workshops, book chapters, and symposiums into one category

(Other) with a total of 23 studies. Journal papers represent 30% of our papers' repository with a total of 35 journal papers.

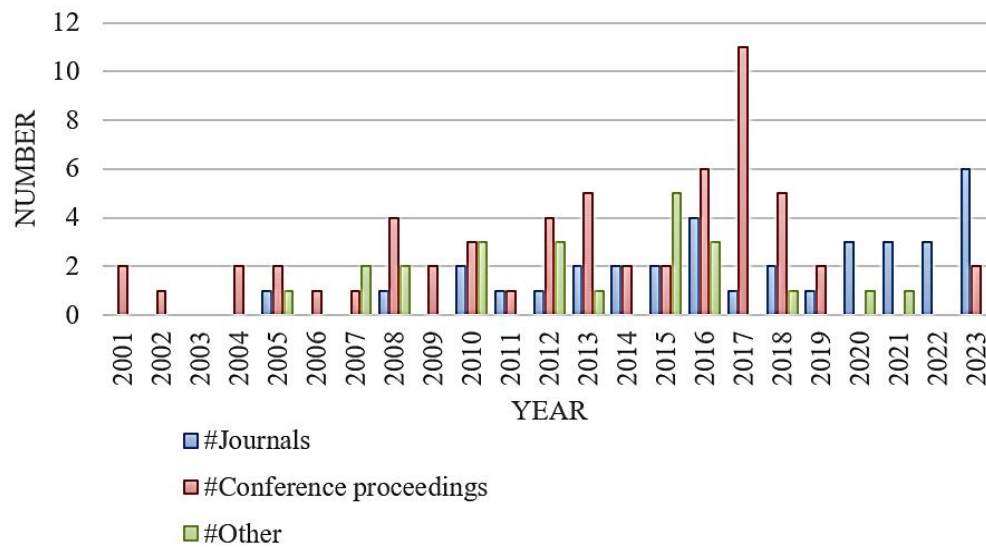


Figure 2. Number of included studies per year

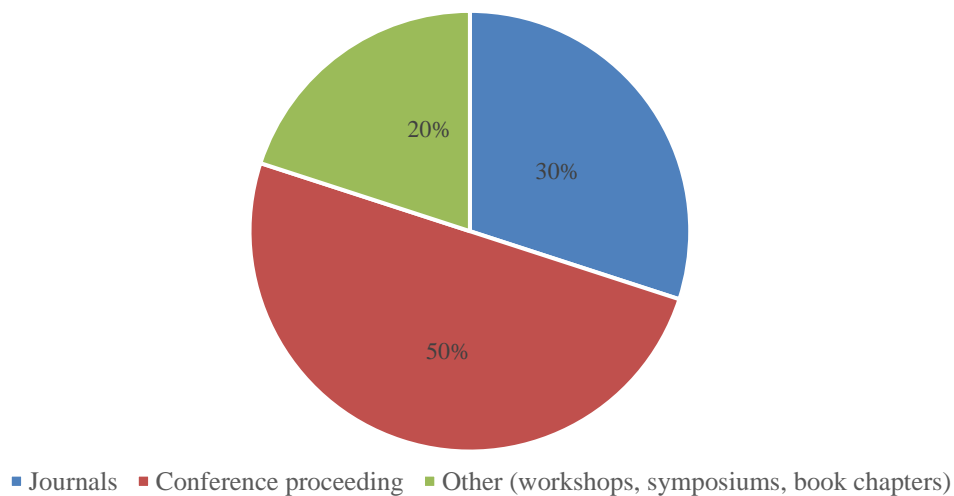


Figure 3. Type of publications

RQ3: What is the distribution of the studies included in this review, categorized by the employed code smell detection technique?

Code smell detection techniques were compared based on the category of the technique, type of detected code smells, supported programming languages, subject systems, and evaluation criteria used to evaluate the technique. Out of the 116 studies included in this analysis, only one study used a manual approach where the detection is done based on human perception [87]. The manual detection process usually takes a long time, and the number of false positive smells are usually high.

Figure 4 shows the percentage of studies included in this review based on the used detection technique. Figure 4 depicts that the predominant approaches in the literature for detecting code smells are metrics-based. In contemporary research, there is a growing utilization of machine learning techniques to identify code smells, displacing other traditional detection methods.

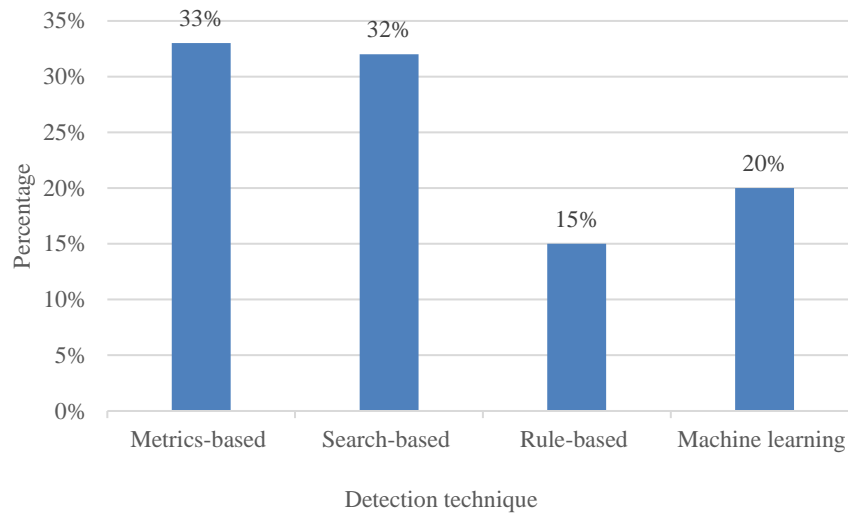


Figure 4. The percentage of studies based on the used detection technique

RQ4: What programming languages are employed by various code smell detection techniques in the included studies?

Based on the analysis of 116 studies, it was noted that Java is the most used language where it is used in 93 studies (80 % of the included studies). Further, three of the recent techniques used Python [91], [119], and [143]. MATLAB is only used in one study [86]. Android is used in the primary studies [111], [114], and [137]. Figure 5 shows the programming languages used by different detection techniques to detect code smells.

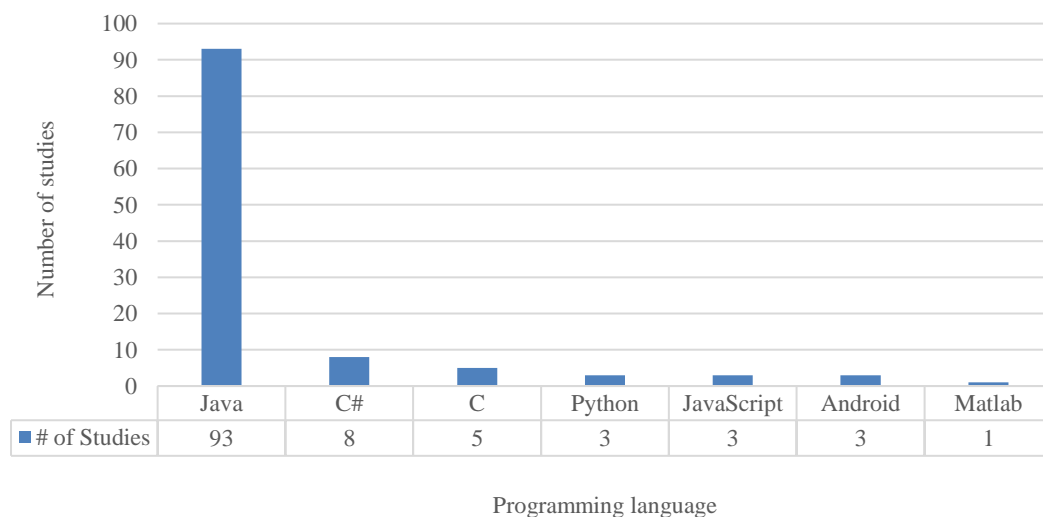


Figure 5. Programming languages used by studies included in this review

RQ5: What subject systems were used to validate code smell detection techniques in the included studies?

Different subject systems were used to validate code smell detection techniques. These subject systems vary in their size, language, and the implemented code smells. Apache Xerces, GanttProject, and ArgoUML are frequently used as subject systems in empirical studies on code smell detection due to their rich set of code smells and extensive use in the software engineering community. Appendix summarizes their common usage.

We noticed that even though subject systems were used by the majority of techniques, such as Apache Xerces, their true positives code smell instances are not explicitly revealed. Hence, a standard

benchmark is required to ease the validation process. This code smells benchmark should reveal the true positive instances of code smells implemented in the subject systems. Table 2 lists the subject systems used only once in literature. We found 60 subject systems appear only in one primary study. The use of these subject systems makes the validation process difficult since there is no standard benchmark that reveals true positive smell instances.

Table 2. Subject systems used in one study

#	Subject System	#	Subject System	#	Subject System
1	20 Java GitHub projects	21	Aspectj	41	Hadoop
2	JasperReports	22	Drjava	42	Hive
3	Velocity Engine	23	Javacc	43	Hsqldb
4	Spring Framework	24	Maven	44	Karaf
5	Struts	25	Xmojo	45	Lucene
6	3 Students' projects	26	131 releases of 13 open-source systems	46	Manifold-cf
7	VideoStore	27	Android Opt Telephony	47	Nutch
8	LANSimulation	28	Android Support	48	Mastodon
9	JExcelAPI	29	Apache Lucene	49	Pig
10	Freeplane	30	Multilabel constructed dataset	50	Qpid
11	Redash	31	Dataset of 281 Java Project	51	Labeled instances of six code smells collected from 74 software systems
12	Xalan	32	Metabase	52	Training dataset involves API calls, Permissions, system calls
13	Poi	33	Joplin	53	Code snippets extracted from eight open-source C# projects
14	LaTeXDraw	34	Freeplane	54	Python dataset built using four open-source Python libraries: Numpy, Django, Matplotlib-, and Scipy
15	aTunes	35	AbdExtractor	55	Grafana
16	MediaPesata	36	Grinder	56	Superset
17	LaTazza	37	Art of Illusion	57	Prometheus
18	Corpus of 106 Python projects	38	JExcelAPI	58	RocketChat
19	30 Open-source Java projects	39	Ant-ivy	59	Ant-design
20	MediaPesata	40	Cassandra	60	Carbon-app

Qualitas corpus was used in four studies [120], [138], [140], and [145]. These studies applied machine learning techniques to detect code smells and used 74 open-source projects to train the model. Figure 6 shows the subject systems used by more than two detection techniques to validate the code smell detection technique. As illustrated in Figure 6, programming mistake detector (PMD), Madeyski Lewowski code quest (MLCQ) dataset, tomcat, areca, weka and struts were used in two primary studies.

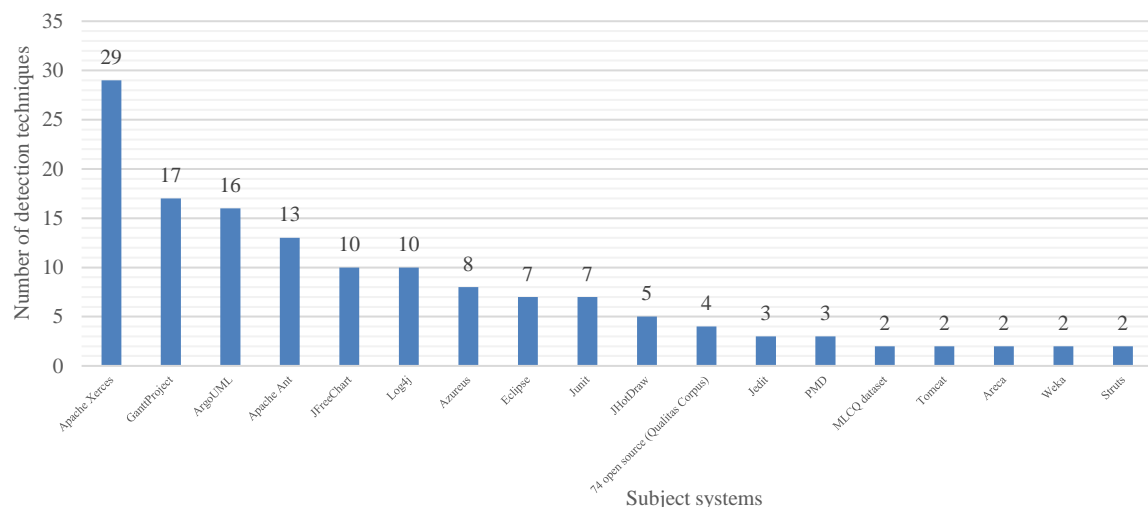


Figure 6. Subject systems used by more than two detection techniques

RQ6: What code smells were detected by various techniques in the included studies?

Based on our investigations of the 116 primary studies, we found that the god class, feature envy, long method, and data class are the most detected smells. This is maybe due to the simple structure of these smells. Shotgun surgery and lazy class are the least detected code smells where they only detected by 10 primary studies. Other types of smells were detected by the detection techniques such as complex method, complex conditional, and multifaceted abstraction in [141]. Figure 7 shows the top code smells detected by the different primary studies.

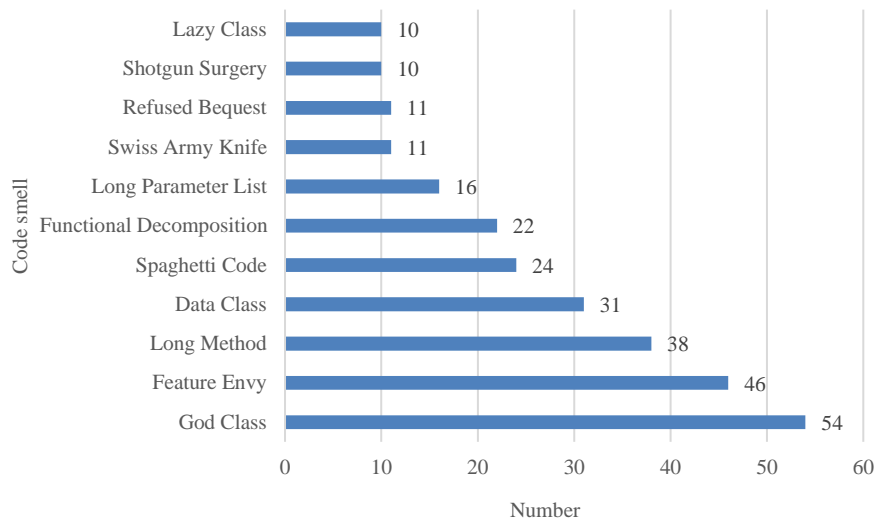


Figure 7. Number of frequent code smells detected by different techniques

RQ7: What evaluation criteria were utilized by code smell detection techniques in the reviewed studies?

To evaluate the performance and the accuracy of the detection techniques, different evaluation methods were employed in the surveyed literature. We found that precision and recall are the most used metrics to evaluate the accuracy of the detection approach where they were used in 69 and 68 studies, respectively. F measure is the third most used evaluation criteria. Figure 8 shows the evaluation criteria used to evaluate the code smell detection techniques. 11 studies did not reveal the evaluation criteria used to evaluate the detection technique.

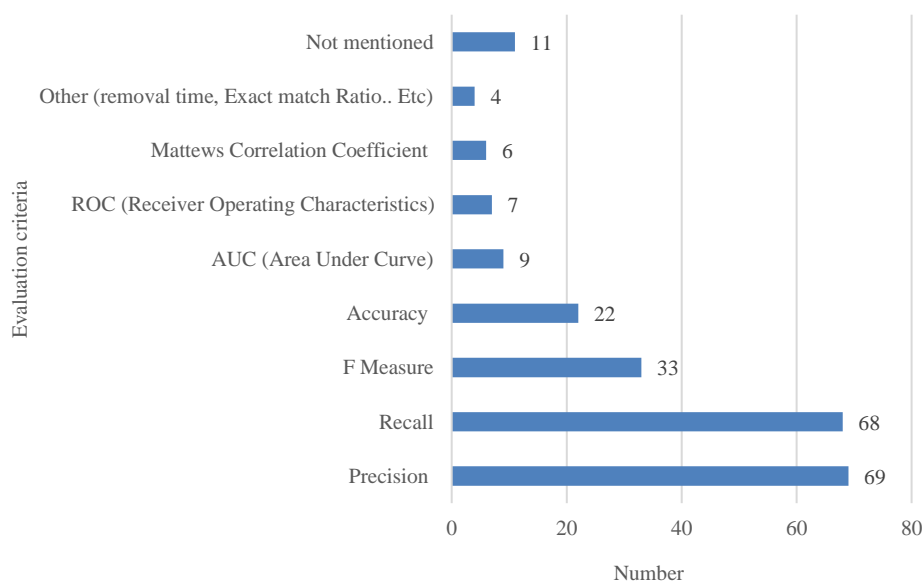


Figure 8. Evaluation criteria used by detection techniques

6. CONCLUSION AND FUTURE WORK

Software code smells are signs of deeper problems in the code. Several studies were presented in the literature to detect these smells from software systems. This paper presented a review of software code smell detection techniques. A final list of 116 primary studies from reputable databases and publishing venues were included in the analysis. We compared software code smell detection techniques in terms of publishing venues over years, supported programming language, detection technique category, subject systems used to validate the technique, detected code smells, and evaluation criteria.

The results show that the majority of the primary studies are published in conference proceedings. Metrics based techniques are the most used techniques to detect code smells. Recent techniques used machine learning to detect code smells. We noticed that most of the detection techniques supported Java programming language. Moreover, the top used subject systems to validate detection techniques are Apache Xerces, GanttProject, and ArgoUML. We found 60 subject systems were used in one primary study. This makes the validation process difficult since there is no standard benchmark to validate the detected smells. Different techniques have different results. To evaluate the proposed detection techniques precision and recall are used by the majority of the detection techniques.

Future work in code smell detection should focus on diversifying subject systems for validation, extending analyses to multiple programming languages, and collaborating to establish standardized benchmarks. Exploring the integration of machine learning alongside traditional metrics-based techniques could enhance detection accuracy. Comparative studies should be conducted to understand variations among detection techniques, while the exploration of alternative evaluation metrics beyond precision and recall is crucial. A longitudinal analysis of trends and changes in code smell detection techniques over time can provide valuable insights. Additionally, efforts should be directed towards developing tools for real-world integration, fostering community collaboration for shared datasets and methodologies, and conducting case studies in industrial settings to assess practical impact and feasibility.

FUNDING INFORMATION

Authors state no funding involved.

AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Mohammed Ghazi Al-Obeidallah	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Dimah Al-Fraihat	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

C : Conceptualization

M : Methodology

So : Software

Va : Validation

Fo : Formal analysis

I : Investigation

R : Resources

D : Data Curation

O : Writing - Original Draft

E : Writing - Review & Editing

Vi : Visualization

Su : Supervision

P : Project administration

Fu : Funding acquisition

Conflict of interest statement

Authors state no conflict of interest.

DATA AVAILABILITY

The data that support the findings of this study are available from the corresponding author, DAF, upon reasonable request.

REFERENCES

- [1] D. Al-Fraihat, Y. Sharrab, A. R. Al-Ghuwairi, N. Sbaih, and A. Qahmash, "Detecting refactoring type of software commit messages based on ensemble machine learning algorithms," *Scientific Reports*, vol. 14, no. 1, Sep. 2024, doi: 10.1038/s41598-024-72307-0.

- [2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D.-R. Roberts, "Improving the design of existing code," *Addison-Wesley*, vol. 6, 1999.
- [3] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, 2007.
- [4] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello, "An experience report on using code smells detection tools," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, Mar. 2011, pp. 450–457, doi: 10.1109/ICSTW.2011.12.
- [5] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, Sep. 2015, doi: 10.1002/smr.1737.
- [6] F. Sabir, F. Palma, G. Rasool, Y. G. Guéhéneuc, and N. Moha, "A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems," *Software - Practice and Experience*, vol. 49, no. 1, pp. 3–39, Oct. 2019, doi: 10.1002/spe.2639.
- [7] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," *Journal of Software Maintenance and Evolution*, vol. 23, no. 3, pp. 179–202, Oct. 2011, doi: 10.1002/smr.521.
- [8] E. V. D. P. Sobrinho, A. De Lucia, and M. D. A. Maia, "A systematic literature review on bad smells-5 W's: Which, when, what, who, where," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 17–66, Jan. 2021, doi: 10.1109/TSE.2018.2880977.
- [9] M. Misbhaudhin and M. Alshayeb, "UML model refactoring: a systematic literature review," *Empirical Software Engineering*, vol. 20, no. 1, pp. 206–251, Oct. 2015, doi: 10.1007/s10664-013-9283-7.
- [10] J. Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, vol. 58, pp. 231–249, 2015, doi: 10.1016/j.infsof.2014.08.002.
- [11] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, Apr. 2019, doi: 10.1016/j.infsof.2018.12.009.
- [12] A. AbuHassan, M. Alshayeb, and L. Ghouti, "Software smell detection techniques: a systematic literature review," *Journal of Software: Evolution and Process*, vol. 33, no. 3, Mar. 2021, doi: 10.1002/smr.2320.
- [13] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow, "Code smells detection and visualization: a systematic literature review," *Archives of Computational Methods in Engineering*, vol. 29, no. 1, pp. 47–94, Jan. 2022, doi: 10.1007/s11831-021-09566-x.
- [14] A. Yamashita and L. Moonen, "Do developers care about code smells? An exploratory survey," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, Oct. 2013, pp. 242–251, doi: 10.1109/WCRE.2013.6671299.
- [15] D. Al-Fraihat, "Evaluating the success of e-learning systems: The case of Moodle LMS at the University of Warwick," Ph.D. dissertation, University of Warwick, 2019.
- [16] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos, "An empirical investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 65, no. 2, pp. 127–139, Feb. 2003, doi: 10.1016/S0164-1212(02)00054-7.
- [17] D. Al-Fraihat, Y. Sharrab, A. R. Al-Ghuwairi, M. Al Elaimat, and M. Alzaidi, "Detecting and resolving feature envy through automated machine learning and move method refactoring," *International Journal of Electrical and Computer Engineering*, vol. 14, no. 2, pp. 2330–2343, Apr. 2024, doi: 10.11591/ijece.v14i2.pp2330-2343.
- [18] D. Al-Fraihat, Y. Sharrab, F. Alzyoud, A. Qahmash, M. Tarawneh, and A. Maaita, "Speech recognition utilizing deep learning: A systematic review of the latest developments," *Human-centric Computing and Information Sciences*, vol. 14, 2024, doi: 10.22967/HICIS.2024.14.015.
- [19] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," *ESEC/FSE'05 - Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13)*, vol. 30, no. 5, pp. 187–196, Sep. 2005, doi: 10.1145/1095430.1081737.
- [20] 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, Jul. 2007, doi: 10.1016/j.jss.2006.10.018.
- [21] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *IEEE International Conference on Software Maintenance, ICSM*, Sep. 2008, pp. 227–236, doi: 10.1109/ICSM.2008.4658071.
- [22] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Proceedings - International Software Metrics Symposium*, 2002, vol. 2002-Janua, pp. 87–94, doi: 10.1109/METRIC.2002.1011328.
- [23] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?," *Empirical Software Engineering*, vol. 17, no. 4–5, pp. 503–530, Dec. 2012, doi: 10.1007/s10664-011-9195-3.
- [24] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings - International Conference on Software Engineering*, May 2013, pp. 682–691, doi: 10.1109/ICSE.2013.6606614.
- [25] Z. Soh, A. Yamashita, F. Khomh, and Y. G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016*, Mar. 2016, vol. 1, pp. 393–402, doi: 10.1109/SANER.2016.103.
- [26] A.-R. Al-Ghuwairi *et al.*, "Visualizing software refactoring using radar charts," *Scientific Reports*, vol. 13, no. 1, Art. no. 19530, Nov. 2023, doi: 10.1038/s41598-023-44281-6.
- [27] M. G. Al-Obeidallah, D. G. Al-Fraihat, A. M. Khasawneh, A. M. Saleh, and H. Addous, "Empirical investigation of the impact of the adapter design pattern on software maintainability," in *2021 International Conference on Information Technology (ICIT)*, Jul. 2021, pp. 206–211, doi: 10.1109/ICIT52682.2021.9491719.
- [28] S. Keele, "Guidelines for performing systematic literature reviews in software engineering," *Technical Report*, 2007.
- [29] D. Al-Fraihat, Y. Sharrab, A. R. Al-Ghuwairi, H. Alshishani, and A. Algarni, "Hyperparameter optimization for software bug prediction using ensemble learning," *IEEE Access*, vol. 12, pp. 51869–51878, 2024, doi: 10.1109/ACCESS.2024.3380024.
- [30] D. Al-Fraihat, Y. Sharrab, A. R. Al-Ghuwairi, H. Alzabut, M. Beshara, and A. Algarni, "Utilizing machine learning algorithms for task allocation in distributed agile software development," *Heliyon*, vol. 10, no. 21, p. e39926, Nov. 2024, doi: 10.1016/j.heliyon.2024.e39926.
- [31] Y. Sharrab, D. Al-Fraihat, M. Tarawneh, and A. Sharieh, "Medicinal plants recognition using deep learning," in *2023 International Conference on Multimedia Computing, Networking and Applications, MCNA 2023*, Jun. 2023, pp. 116–122, doi: 10.1109/MCNA59361.2023.10185880.
- [32] R. Marinescu, "Detecting design flaws via metrics in object-oriented systems," in *Technology of Object-Oriented Languages and*

- Systems*, 2001, no. TOOL, pp. 173–182, doi: 10.1109/tools.2001.941671.
- [33] J. Krinke, “Identifying similar code with program dependence graphs,” in *Reverse Engineering - Working Conference Proceedings*, 2001, pp. 301–309, doi: 10.1109/wcre.2001.957835.
 - [34] E. Van Emden and L. Moonen, “Java quality assurance by detecting code smells,” in *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2002, vol. 2002-Janua, pp. 97–106, doi: 10.1109/WCRE.2002.1173068.
 - [35] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in *IEEE International Conference on Software Maintenance, ICSM*, 2004, pp. 350–359, doi: 10.1109/ICSM.2004.1357820.
 - [36] M. Rieger, S. Ducasse, and M. Lanza, “Insights into system-wide code duplication,” in *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2004, pp. 100–109, doi: 10.1109/WCRE.2004.25.
 - [37] J. Kreimer, “Adaptive detection of design flaws,” *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117–136, Dec. 2005, doi: 10.1016/j.entcs.2005.02.059.
 - [38] B. Walter and B. Pietrzak, “Multi-criteria detection of bad smells in code with UTA method,” in *Lecture Notes in Computer Science*, vol. 3556, Springer Berlin Heidelberg, 2005, pp. 154–161.
 - [39] G. Langelier, H. Sahraoui, and P. Poulin, “Visualization-based analysis of quality for large-scale software systems,” in *20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005*, Nov. 2005, pp. 214–223, doi: 10.1145/1101908.1101941.
 - [40] M. J. Munro, “Product metrics for automatic identification of ‘bad smell’ design problems in Java source-code,” in *Proceedings - International Software Metrics Symposium*, 2005, vol. 2005, pp. 15–24, doi: 10.1109/METRICS.2005.38.
 - [41] N. Moha, Y. G. Guéhéneuc, and P. Leduc, “Automatic generation of detection algorithms for design defects,” in *Proceedings - 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006*, 2006, pp. 297–300, doi: 10.1109/ASE.2006.22.
 - [42] A. Wasylkowski, A. Zeller, and C. Lindig, “Detecting object usage anomalies,” in *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2007*, Sep. 2007, pp. 35–44, doi: 10.1145/1287624.1287632.
 - [43] T. Grba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel, “Using concept analysis to detect co-change patterns,” in *International Workshop on Principles of Software Evolution (IWPSSE)*, Sep. 2007, pp. 83–89, doi: 10.1145/1294948.1294970.
 - [44] C. J. H. Mann, “Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems,” *Kybernetes*, vol. 36, no. 5/6, Jun. 2007, doi: 10.1108/k.2007.06736eae.001.
 - [45] R. Wetzel and M. Lanza, “Visually localizing design problems with disharmony maps,” in *SOFTVIS 2008 - Proceedings of the 4th ACM Symposium on Software Visualization*, Sep. 2008, pp. 155–164, doi: 10.1145/1409720.1409745.
 - [46] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, “JDeodorant: Identification and removal of type-checking bad smells,” in *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, Apr. 2008, pp. 329–331, doi: 10.1109/CSMR.2008.4493342.
 - [47] a a Rao and K. N. Reddy, “Detecting bad smells in object oriented design using design change propagation probability matrix,” *International MultiConference of Engineers and Computer Scientists*, vol. I, pp. 1001–1007, 2008.
 - [48] C. Parnin, C. Görg, and O. Nnadi, “A catalogue of lightweight visualizations to support code smell inspection,” in *SOFTVIS 2008 - Proceedings of the 4th ACM Symposium on Software Visualization*, Sep. 2008, pp. 77–86, doi: 10.1145/1409720.1409733.
 - [49] R. Falke, P. Frenzel, and R. Koschke, “Empirical evaluation of clone detection using syntax suffix trees,” *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, Jul. 2008, doi: 10.1007/s10664-008-9073-9.
 - [50] N. Moha, Y. G. Guéhéneuc, A. F. Le Meur, and L. Duchien, “A domain analysis to specify design defects and generate detection algorithms,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4961, Springer Berlin Heidelberg, 2008, pp. 276–291.
 - [51] K. Dhambri, H. Sahraoui, and P. Poulin, “Visual detection of design anomalies,” in *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, Apr. 2008, pp. 279–283, doi: 10.1109/CSMR.2008.4493326.
 - [52] J. Nödl, H. Neukirchen, and J. Grabowski, “A flexible framework for quality assurance of software artefacts with applications to Java, UML, and TTCN-3 test specifications,” in *Proceedings - 2nd International Conference on Software Testing, Verification, and Validation, ICST 2009*, Apr. 2009, pp. 101–110, doi: 10.1109/ICST.2009.34.
 - [53] F. Khomh, S. Vaucher, Y. G. Guéhéneuc, and H. Sahraoui, “A bayesian approach for the detection of code and design smells,” in *Proceedings - International Conference on Quality Software*, Aug. 2009, pp. 305–314, doi: 10.1109/QSIC.2009.47.
 - [54] S. Bryton, F. Brito E Abreu, and M. Monteiro, “Reducing subjectivity in code smells detection: Experimenting with the Long Method,” in *Proceedings - 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010*, Sep. 2010, pp. 337–342, doi: 10.1109/QUATIC.2010.60.
 - [55] N. Moha, Y. G. Guéhéneuc, L. Duchien, and A. F. Le Meur, “DECOR: A method for the specification and detection of code and design smells,” *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, Jan. 2010, doi: 10.1109/TSE.2009.50.
 - [56] E. Murphy-Hill and A. P. Black, “An interactive ambient visualization for code smells,” in *Proceedings of the 5th international symposium on Software visualization*, Oct. 2010, pp. 5–14, doi: 10.1145/1879211.1879216.
 - [57] G. D. F. Carneiro *et al.*, “Identifying code smells with multiple concern views,” in *Proceedings - 24th Brazilian Symposium on Software Engineering, SBES 2010*, Sep. 2010, pp. 128–137, doi: 10.1109/SBES.2010.21.
 - [58] N. Gruska, A. Wasylkowski, and A. Zeller, “Learning from 6,000 projects: Lightweight cross-project anomaly detection,” in *ISSTA '10 - Proceedings of the 2010 International Symposium on Software Testing and Analysis*, Jul. 2010, pp. 119–129, doi: 10.1145/1831708.1831723.
 - [59] N. Moha, Y. G. Guéhéneuc, A. F. Le Meur, L. Duchien, and A. Tiberghien, “From a domain analysis to the specification and detection of code and design smells,” *Formal Aspects of Computing*, vol. 22, no. 3–4, pp. 345–361, May 2010, doi: 10.1007/s00165-009-0115-x.
 - [60] S. Hassaine, F. Khomh, Y. G. Guéhéneuc, and S. Hamel, “IDS: An immune-inspired approach for the detection of software design smells,” in *Proceedings - 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010*, Sep. 2010, pp. 343–348, doi: 10.1109/QUATIC.2010.61.
 - [61] M. Monperrus, M. Bruch, and M. Mezini, “Detecting missing method calls in object-oriented software,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6183 LNCS, Springer Berlin Heidelberg, 2010, pp. 2–25.
 - [62] F. Khomh, S. Vaucher, Y. G. Guéhéneuc, and H. Sahraoui, “BDTEX: A QGM-based Bayesian approach for the detection of antipatterns,” *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, Apr. 2011, doi: 10.1016/j.jss.2010.11.921.
 - [63] P. Lerthathairat and N. Prompoon, “An approach for source code classification using software metrics and fuzzy logic to improve code quality with refactoring techniques,” in *Communications in Computer and Information Science*, vol. 181 CCIS, no. PART 3,

- Springer Berlin Heidelberg, 2011, pp. 478–492.
- [64] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. G. Guéhenec, and E. Aimeur, “SMURF: A SVM-based incremental anti-pattern detection approach,” in *Proceedings - Working Conference on Reverse Engineering, WCRE*, Oct. 2012, pp. 466–475, doi: 10.1109/WCRE.2012.56.
 - [65] D. Jiang, P. Ma, X. Su, and T. Wang, “Detecting bad smells with weight based distance metrics theory,” in *Proceedings of the 2012 2nd International Conference on Instrumentation and Measurement, Computer, Communication and Control, IMCCC 2012*, Dec. 2012, pp. 299–304, doi: 10.1109/IMCCC.2012.74.
 - [66] R. Mahouachi, M. Kessentini, and K. Ghedira, “A new design defects classification: Marrying detection and correction,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7212 LNCS, Springer Berlin Heidelberg, 2012, pp. 455–470.
 - [67] M. F. Zibran and C. K. Roy, “IDE-based real-time focused search for near-miss clones,” in *Proceedings of the ACM Symposium on Applied Computing*, Mar. 2012, pp. 1235–1242, doi: 10.1145/2245276.2231970.
 - [68] C. Dandois and W. Vanhoof, “Clones in logic programs and how to detect them,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7225 LNCS, Springer Berlin Heidelberg, 2012, pp. 90–105.
 - [69] S. Mekruksavanich, P. P. Yupapin, and P. Muenchaisri, “Analytical learning based on a meta-programming approach for the detection of object-oriented design defects,” *Information Technology Journal*, vol. 11, no. 12, pp. 1677–1686, Nov. 2012, doi: 10.3923/itj.2012.1677.1686.
 - [70] I. Polásek, S. Snopko, and I. Kapustík, “Automatic identification of the anti-patterns using the rule-based approach,” in *2012 IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics, SISY 2012*, Sep. 2012, pp. 283–286, doi: 10.1109/SISY.2012.6339530.
 - [71] A. Maiga *et al.*, “Support vector machines for anti-pattern detection,” in *2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings*, Sep. 2012, pp. 278–281, doi: 10.1145/2351676.2351723.
 - [72] K. E. Rajakumari and T. Jebarajan, “A novel approach to effective detection and analysis of code clones,” in *2013 3rd International Conference on Innovative Computing Technology, INTECH 2013*, Aug. 2013, pp. 287–290, doi: 10.1109/INTECH.2013.6653701.
 - [73] M. Monperrus and M. Mezini, “Detecting missing method calls as violations of the majority rule,” *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 1, pp. 1–25, Feb. 2013, doi: 10.1145/2430536.2430541.
 - [74] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, “Code smell detection: Towards a machine learning-based approach,” in *IEEE International Conference on Software Maintenance, ICSM*, Sep. 2013, pp. 396–399, doi: 10.1109/ICSM.2013.56.
 - [75] E. Murphy-Hill, T. Barik, and A. P. Black, “Interactive ambient visualizations for soft advice,” *Information Visualization*, vol. 12, no. 2, pp. 107–132, Mar. 2013, doi: 10.1177/1473871612469020.
 - [76] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, “Identification of refused bequest code smells,” in *IEEE International Conference on Software Maintenance, ICSM*, Sep. 2013, pp. 392–395, doi: 10.1109/ICSM.2013.55.
 - [77] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Ben Chikha, “Competitive coevolutionary code-smells detection,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8084 LNCS, Springer Berlin Heidelberg, 2013, pp. 50–65.
 - [78] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyanyk, “Detecting bad smells in source code using change history information,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, Nov. 2013, pp. 268–278, doi: 10.1109/ASE.2013.6693086.
 - [79] A. M. Fard and A. Mesbah, “JSNOSE: Detecting javascript code smells,” in *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013*, Sep. 2013, pp. 116–125, doi: 10.1109/SCAM.2013.6648192.
 - [80] S. Kumar and J. K. Chhabra, “Two level dynamic approach for feature envy detection,” in *Proceedings - 5th IEEE International Conference on Computer and Communication Technology, ICCCT 2014*, Sep. 2015, pp. 41–46, doi: 10.1109/ICCCT.2014.7001467.
 - [81] G. de Figueiredo Carneiro and M. G. de Mendonça Neto, “Sourceminer: Towards an extensible multi-perspective software visualization environment,” in *Lecture Notes in Business Information Processing*, vol. 190, Springer International Publishing, 2014, pp. 242–263.
 - [82] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, “Code-smell detection as a bilevel problem,” *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 1, pp. 1–44, Oct. 2014, doi: 10.1145/2675067.
 - [83] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, “A cooperative parallel search-based software engineering approach for code-smells detection,” *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, Sep. 2014, doi: 10.1109/TSE.2014.2331057.
 - [84] G. Czibula, Z. Marian, and I. G. Czibula, “Detecting software design defects using relational association rule mining,” *Knowledge and Information Systems*, vol. 42, no. 3, pp. 545–577, Jan. 2015, doi: 10.1007/s10115-013-0721-z.
 - [85] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro, “Experience report: Evaluating the effectiveness of decision trees for detecting code smells,” in *2015 IEEE 26th International Symposium on Software Reliability Engineering, ISSRE 2015*, Nov. 2016, pp. 261–269, doi: 10.1109/ISSRE.2015.7381819.
 - [86] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyanyk, and A. De Lucia, “Mining version histories for detecting code smells,” *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, May 2015, doi: 10.1109/TSE.2014.2372760.
 - [87] T. Gerlit, Q. M. Tran, and C. Dziobek, “Detection and handling of model smells for MATLAB/Simulink models,” *MASE@MoDELS*, 2015.
 - [88] S. Fu and B. Shen, “Code bad smell detection through evolutionary data mining,” in *International Symposium on Empirical Software Engineering and Measurement*, Oct. 2015, vol. 2015-Novem, pp. 41–49, doi: 10.1109/ESEM.2015.7321194.
 - [89] 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 *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*, Nov. 2016, vol. 2016-Febru, pp. 1–6, doi: 10.1109/SCCC.2015.7416572.
 - [90] B. Walter, B. Matuszyk, and F. A. Fontana, “Including structural factors into the metrics-based code smells detection,” in *ACM International Conference Proceeding Series*, May 2015, vol. 25-29-May-, pp. 1–5, doi: 10.1145/2764979.2764990.
 - [91] F. Palomba, “Textual analysis for code smell detection,” in *Proceedings - International Conference on Software Engineering*, May 2015, vol. 2, pp. 769–771, doi: 10.1109/ICSE.2015.244.
 - [92] M. Hozano, H. Ferreira, I. Silva, B. Fonseca, and E. Costa, “Using developers’ feedback to improve code smell detection,” in *Proceedings of the ACM Symposium on Applied Computing*, Apr. 2015, vol. 13-17-Apri, pp. 1661–1663, doi: 10.1145/2695664.2696059.
 - [93] S. Romano, G. Scanniello, C. Sartiani, and M. Risi, “A graph-based approach to detect unreachable methods in Java software,” in




- Proceedings of the ACM Symposium on Applied Computing*, Apr. 2016, vol. 04-08-Apri, pp. 1538–1541, doi: 10.1145/2851613.2851968.
- [94] 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, Jul. 2017, doi: 10.1007/s13369-016-2238-8.
- [95] S. Peldszus, G. Kulcsár, M. Lochau, and S. Schulze, “Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching,” in *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, Aug. 2016, pp. 578–589, doi: 10.1145/2970276.2970338.
- [96] K. Sirikul and C. Soomlek, “Automated detection of code smells caused by null checking conditions in Java programs,” in *2016 13th International Joint Conference on Computer Science and Software Engineering, JCSSE 2016*, Jul. 2016, pp. 1–7, doi: 10.1109/JCSSE.2016.7748884.
- [97] Z. Chen, L. Chen, W. Ma, and B. Xu, “Detecting code smells in python programs,” in *Proceedings - 2016 International Conference on Software Analysis, Testing and Evolution, SATE 2016*, Nov. 2016, pp. 18–23, doi: 10.1109/SATE.2016.10.
- [98] M. W. Mkaouer, “Interactive code smells detection: An initial investigation,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9962 LNCS, Springer International Publishing, 2016, pp. 281–287.
- [99] F. C. Luiz, B. R. De Oliveira, and F. S. Parreiras, “Machine learning techniques for code smells detection: An empirical experiment on a highly imbalanced setup,” in *ACM International Conference Proceeding Series*, May 2019, pp. 1–8, doi: 10.1145/3330204.3330275.
- [100] 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, Nov. 2016, doi: 10.15866/irecos.v11i11.10590.
- [101] X. Liu and C. Zhang, “DT : a detection tool to automatically detect code smell in software project,” 2016, doi: 10.2991/icmmita-16.2016.126.
- [102] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, “A textual-based technique for smell detection,” in *IEEE International Conference on Program Comprehension*, May 2016, vol. 2016-July, pp. 1–10, doi: 10.1109/ICPC.2016.7503704.
- [103] M. T. Aras and Y. E. Selcuk, “Metric and rule based automated detection of antipatterns in object-oriented software systems,” in *Proceedings - CSIT 2016: 2016 7th International Conference on Computer Science and Information Technology*, Jul. 2016, pp. 1–6, doi: 10.1109/CSIT.2016.7549470.
- [104] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, “Multi-objective code-smells detection using good and bad design examples,” *Software Quality Journal*, vol. 25, no. 2, pp. 529–552, Feb. 2017, doi: 10.1007/s11219-016-9309-7.
- [105] F. Arcelli Fontana, M. V. Mäntylä, M. Zaroni, and A. Marino, “Comparing and experimenting machine learning techniques for code smell detection,” *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, Jun. 2016, doi: 10.1007/s10664-015-9378-4.
- [106] M. Sangeetha and P. Sengottuvelan, “Systematic exhortation of code smell detection using JSmell for Java source code,” in *Proceedings of the International Conference on Inventive Systems and Control, ICISC 2017*, Jan. 2017, pp. 1–5, doi: 10.1109/ICISC.2017.8068719.
- [107] D. K. Kim, “Finding bad code smells with neural network models,” *International Journal of Electrical and Computer Engineering*, vol. 7, no. 6, pp. 3613–3621, Dec. 2017, doi: 10.11591/ijece.v7i6.pp3613-3621.
- [108] L. P. Da Silva Carvalho, R. Novais, L. Do Nascimento Salvador, and M. G. De Mendonça Neto, “An ontology-based approach to analyzing the occurrence of code smells in software,” in *ICEIS 2017 - Proceedings of the 19th International Conference on Enterprise Information Systems*, 2017, vol. 2, pp. 155–165, doi: 10.5220/0006359901550165.
- [109] I. Shoenberger, M. W. Mkaouer, and M. Kessentini, “On the use of smelly examples to detect code smells in Javascript,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10200 LNCS, Springer International Publishing, 2017, pp. 20–34.
- [110] M. Steinbeck, “An arc-based approach for visualization of code smells,” in *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, Feb. 2017, pp. 397–401, doi: 10.1109/SANER.2017.7884641.
- [111] M. Kessentini and A. Ouni, “Detecting Android smells using multi-objective genetic programming,” in *Proceedings - 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft 2017*, May 2017, pp. 122–132, doi: 10.1109/MOBILESoft.2017.29.
- [112] M. Hozano, A. Garcia, N. Antunes, B. Fonseca, and E. Costa, “Smells are sensitive to developers! On the efficiency of (un)guided customized detection,” in *IEEE International Conference on Program Comprehension*, May 2017, pp. 110–120, doi: 10.1109/ICPC.2017.32.
- [113] S. Velioglu and Y. E. Selcuk, “An automated code smell and anti-pattern detection approach,” in *Proceedings - 2017 15th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2017*, Jun. 2017, pp. 271–275, doi: 10.1109/SERA.2017.7965737.
- [114] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, “Lightweight detection of Android-specific code smells: The aDoctor project,” in *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, Feb. 2017, pp. 487–491, doi: 10.1109/SANER.2017.7884659.
- [115] A. Kaur, S. Jain, and S. Goel, “A support vector machine based approach for code smell detection,” in *Proceedings - 2017 International Conference on Machine Learning and Data Science, MLDS 2017*, Dec. 2017, pp. 9–14, doi: 10.1109/MLDS.2017.8.
- [116] N. Ujihara, A. Ouni, T. Ishio, and K. Inoue, “C-JRefRec: Change-based identification of move method refactoring opportunities,” in *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, Feb. 2017, pp. 482–486, doi: 10.1109/SANER.2017.7884658.
- [117] B. M. Merzah and Y. E. Selcuk, “Metric based detection of refused bequest code smell,” in *Proceedings - 9th International Conference on Computational Intelligence and Communication Networks, CICN 2017*, Sep. 2017, vol. 2018-Janua, pp. 53–57, doi: 10.1109/CICN.2017.8319355.
- [118] W. K. Chen, C. H. Liu, and B. H. Li, “A feature envy detection method based on dataflow analysis,” in *Proceedings - International Computer Software and Applications Conference*, Jul. 2018, vol. 2, pp. 14–19, doi: 10.1109/COMPSAC.2018.10196.
- [119] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu, “Understanding metric-based detectable smells in Python software: A comparative study,” *Information and Software Technology*, vol. 94, pp. 14–29, Feb. 2018, doi: 10.1016/j.infsof.2017.09.011.
- [120] M. Hadj-Kacem and N. Bouassida, “A hybrid approach to detect code smells using deep learning,” in *ENASE 2018 - Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2018, vol. 2018-March,

- pp. 137–146, doi: 10.5220/0006709801370146.
- [121] G. Saranya, H. Khanna Nehemiah, A. Kannan, and V. Nithya, “Model level code smell detection using EGAPSO based on similarity measures,” *Alexandria Engineering Journal*, vol. 57, no. 3, pp. 1631–1642, Sep. 2018, doi: 10.1016/j.aej.2017.07.006.
 - [122] M. Škipina, J. Slivka, N. Luburić, and A. Kovačević, “Automatic detection of feature envy and data class code smells using machine learning,” *Expert Systems with Applications*, vol. 243, Mar. 2024, doi: 10.1016/j.eswa.2023.122855.
 - [123] H. Mumtaz, F. Beck, and D. Weiskopf, “Detecting bad smells in software systems with linked multivariate visualizations,” in *Proceedings - 6th IEEE Working Conference on Software Visualization, VISSOFT 2018*, Sep. 2018, pp. 12–20, doi: 10.1109/VISSOFT.2018.00010.
 - [124] S. Romano and G. Scanniello, “Exploring the use of rapid type analysis for detecting the dead method smell in Java code,” in *Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018*, Aug. 2018, pp. 167–174, doi: 10.1109/SEAA.2018.00035.
 - [125] H. Liu, Z. Xu, and Y. Zou, “Deep learning based feature envy detection,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Sep. 2018, pp. 385–396, doi: 10.1145/3238147.3238166.
 - [126] A. K. Das, S. Yadav, and S. Dhal, “Detecting code smells using deep learning,” in *IEEE Region 10 Annual International Conference, Proceedings/TENCON*, Oct. 2019, vol. 2019–Octob, pp. 2081–2086, doi: 10.1109/TENCON.2019.8929628.
 - [127] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, “Comparing heuristic and machine learning approaches for metric-based code smell detection,” in *IEEE International Conference on Program Comprehension*, May 2019, vol. 2019–May, pp. 93–104, doi: 10.1109/ICPC.2019.00023.
 - [128] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, “Deep learning based code smell detection,” *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1811–1837, 2021, doi: 10.1109/TSE.2019.2936376.
 - [129] 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 Computing and Applications*, vol. 32, no. 11, pp. 7009–7027, Apr. 2020, doi: 10.1007/s00521-019-04175-z.
 - [130] A. Barbez, F. Khomh, and Y. G. Guéhéneuc, “A machine-learning based ensemble method for anti-patterns detection,” *Journal of Systems and Software*, vol. 161, p. 110486, Mar. 2020, doi: 10.1016/j.jss.2019.110486.
 - [131] T. Guggulothu and S. A. Moiz, “Code smell detection using multi-label classification approach,” *Software Quality Journal*, vol. 28, no. 3, pp. 1063–1086, Apr. 2020, doi: 10.1007/s11219-020-09498-y.
 - [132] H. Grodzicka, A. Ziobrowski, Z. Łakomiak, and L. Kawa Michałand Madeyski, “Code smell prediction employing machine learning meets emerging java language constructs,” in *Lecture Notes on Data Engineering and Communications Technologies*, vol. 40, Springer International Publishing, 2020, pp. 137–167.
 - [133] M. M. Draz, M. S. Farhan, S. N. Abdulkader, and M. G. Gafar, “Code smell detection using whale optimization algorithm,” *Computers, Materials & Continua*, vol. 68, no. 2, pp. 1919–1935, 2021, doi: 10.32604/cmc.2021.015586.
 - [134] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, M. Makhoulouf, and L. Ben Said, “Code smell detection and identification in imbalanced environments,” *Expert Systems with Applications*, vol. 166, p. 114076, Mar. 2021, doi: 10.1016/j.eswa.2020.114076.
 - [135] M. De Stefano, F. Pecorelli, F. Palomba, and A. De Lucia, “Comparing within-and cross-project machine learning algorithms for code smell detection,” in *MaLTESQuE 2021 - Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution, co-located with ESEC/FSE 2021*, Aug. 2021, pp. 1–6, doi: 10.1145/3472674.3473978.
 - [136] A. Alazba and H. Aljamaan, “Code smell detection using feature selection and stacking ensemble: An empirical investigation,” *Information and Software Technology*, vol. 138, p. 106648, Oct. 2021, doi: 10.1016/j.infsof.2021.106648.
 - [137] E. Amer and S. El-Sappagh, “Robust deep learning early alarm prediction model based on the behavioural smell for android malware,” *Computers and Security*, vol. 116, p. 102670, May 2022, doi: 10.1016/j.cose.2022.102670.
 - [138] S. Jain and A. Saha, “Rank-based univariate feature selection methods on machine learning classifiers for code smell detection,” *Evolutionary Intelligence*, vol. 15, no. 1, pp. 609–638, Jan. 2022, doi: 10.1007/s12065-020-00536-z.
 - [139] A. Kovačević *et al.*, “Automatic detection of long method and god class code smells through neural source code embeddings,” *Expert Systems with Applications*, vol. 204, p. 117607, Oct. 2022, doi: 10.1016/j.eswa.2022.117607.
 - [140] R. S. Rao, S. Dewangan, A. Mishra, and M. Gupta, “A study of dealing class imbalance problem with machine learning methods for code smell severity detection using PCA-based feature selection technique,” *Scientific Reports*, vol. 13, no. 1, Sep. 2023, doi: 10.1038/s41598-023-43380-8.
 - [141] A. Ho, A. M. T. Bui, P. T. Nguyen, and A. Di Salle, “Fusion of deep convolutional and LSTM recurrent neural networks for automated detection of code smells,” in *ACM International Conference Proceeding Series*, Jun. 2023, pp. 229–234, doi: 10.1145/3593434.3593476.
 - [142] A. Kovačević *et al.*, “Automatic detection of code smells using metrics and CodeT5 embeddings: a case study in C#,” *Neural Computing and Applications*, vol. 36, no. 16, pp. 9203–9220, Aug. 2024, doi: 10.1007/s00521-024-09551-y.
 - [143] R. Sandouka and H. Aljamaan, “Python code smells detection using conventional machine learning models,” *PeerJ Computer Science*, vol. 9, p. e1370, May 2023, doi: 10.7717/peerj-cs.1370.
 - [144] S. Alawadi *et al.*, “FedCSD: A federated learning based approach for code-smell detection,” *IEEE Access*, vol. 12, pp. 44888–44904, 2024, doi: 10.1109/ACCESS.2024.3380167.
 - [145] N. A. A. Khleel and K. Nehéz, “Detection of code smells using machine learning techniques combined with data-balancing methods,” *International Journal of Advances in Intelligent Informatics*, vol. 9, no. 3, pp. 402–417, Nov. 2023, doi: 10.26555/ijain.v9i3.981.
 - [146] F. Ferreira and M. T. Valente, “Detecting code smells in React-based Web apps,” *Information and Software Technology*, vol. 155, p. 107111, Mar. 2023, doi: 10.1016/j.infsof.2022.107111.
 - [147] P. Sukkasem and C. Soomlek, “Enhanced machine learning-based code smell detection through hyper-parameter optimization,” in *Proceedings of JCSSE 2023 - 20th International Joint Conference on Computer Science and Software Engineering*, Jun. 2023, pp. 297–302, doi: 10.1109/JCSSE58229.2023.10202124.




APPENDIX

Subject System	Description
Apache Xerces	"Apache Xerces is a widely-used open-source XML parser library by the Apache Software Foundation. It supports parsing, validating, and manipulating XML documents in Java and C++, adhering to W3C standards. Xerces provides both DOM and SAX APIs, robust error handling, and efficient performance for XML processing in diverse software engineering applications."
GanttProject	"GanttProject is an open-source project management tool designed to create and manage Gantt charts. It helps users plan, schedule, and track tasks, milestones, and resources in projects. GanttProject features include task dependencies, resource allocation, Gantt chart generation, and export options for sharing project plans. It is widely used for project scheduling and management in various industries."
ArgoUML	"ArgoUML is an open-source UML modeling tool used for visualizing and designing software systems using the Unified Modeling Language (UML). It provides a user-friendly interface for creating diagrams such as class diagrams, use case diagrams, sequence diagrams, and more. ArgoUML supports code generation from UML diagrams and facilitates collaboration through version control systems. It is widely utilized in software engineering for modeling and documenting software architectures and designs."
Apache Ant	"A Java library and build tool specifically designed for Java applications. It is used in studies due to its extensive use in real-world applications and the richness of its code smells. It has been used in 15 different detection techniques."
JFreeChart	"A powerful Java chart library specialized in generating professional-quality charts. It is included in studies because of its well-documented codebase and the variety of code smells it exhibits. It has been used in 13 different detection techniques."
Log4j	"Log4j is a popular Java logging framework that helps developers record application runtime information, errors, and debugging details. It offers flexible configuration options through properties or XML files, supports multiple logging levels, various output destinations (like console or files), and customizable log message formatting. Log4j is valued for its simplicity, performance, and widespread adoption in Java applications for effective logging and debugging."
Azureus	"Azureus, now known as Vuze, is a popular BitTorrent client written in Java. It allows users to download and share files using the BitTorrent protocol, which enables efficient peer-to-peer file transfer. Azureus/Vuze provides features such as torrent management, bandwidth prioritization, RSS subscription support for automatic downloads, and a user-friendly interface for monitoring download progress and managing downloaded files. Its complex codebase and frequent updates make it a valuable subject for code smell research. It has been widely used for its comprehensive set of features and cross-platform compatibility."
Eclipse	"An integrated development environment (IDE) used in studies for its extensive plugin architecture and large codebase. It has been used in 7 different detection techniques."
JUnit	"JUnit is a vital Java testing framework that automates unit testing with intuitive annotations and robust assertion methods, ensuring code reliability and quality across development stages. It provides tools for developers to write and run repeatable tests to ensure the correctness of their Java code. JUnit supports annotations like @Test for identifying test methods, various assertion methods for verifying expected outcomes, and setup/teardown methods (@Before, @After) for test setup and cleanup tasks. It integrates seamlessly with build tools like Maven and Gradle, making it a fundamental tool for test-driven development (TDD)."
JHotDraw	"JHotDraw is an open-sourc Java library used for creating graphical editors and diagramming applications. It provides a framework with reusable components and design patterns specifically tailored for building interactive graphical user interfaces (GUIs). JHotDraw simplifies the development of custom drawing editors by offering ready-made tools for handling graphical elements, user interactions, and undo-redo functionality. It is widely utilized in software engineering for developing visual modeling tools, diagram editors, and other applications requiring sophisticated graphical interfaces in Java-based environments."
PMD	"PMD is a source code analyzer for Java, widely used in software development to identify potential code issues, such as bugs, performance bottlenecks, unused variables, and complex code structures. It helps improve code quality by highlighting areas for refactoring and optimization, promoting best practices and maintainability in Java projects."
Tomcat	"Tomcat is an open-source web server and servlet container developed by the Apache Software Foundation. It provides a robust environment for running Java Servlets and JavaServer Pages (JSP) applications on various platforms. Tomcat implements the Java Servlet and JavaServer Pages specifications, providing a lightweight and efficient container for deploying web applications. It is widely used in production environments due to its reliability, scalability, and extensive community support."
Weka	"Weka is a prominent open-source software suite used for machine learning and data mining tasks. Written in Java, it provides a rich collection of algorithms for classification, regression, clustering, association rule mining, and feature selection. Weka includes a user-friendly graphical interface for easy experimentation with machine learning techniques, making it accessible to both beginners and experts. It supports data preprocessing, model evaluation, and visualization, making it a versatile tool for developing and deploying machine learning solutions in various software engineering applications."
Struts	"Apache Struts is an open-source Java web application framework that follows the MVC pattern. It simplifies development with components like Action classes, Form beans, and tag libraries for UI creation. Struts include built-in validation and integrates well with JSP, Servlets, and other Java frameworks, making it popular for scalable and maintainable web applications."

BIOGRAPHIES OF AUTHORS

Mohammed Ghazi Al-Obeidallah    received his Ph.D. in computer science/software engineering from the University of Brighton, UK. Currently, he is working as an assistant professor at the College of Engineering at Abu Dhabi University, Abu Dhabi, UAE. His main research interests are design patterns, reverse engineering, requirements engineering, and privacy. He can be contacted at email: Mohammed.obeidallah@adu.ac.ae.



Dimah Al-Fraihat    received her Ph.D. in computer science/software engineering from the University of Warwick, UK. She is an associate professor at the Department of Software Engineering, Faculty of Information Technology, Isra University, Jordan. Her research interests include refactoring, design patterns, software testing, requirements engineering, documentation, computer-based applications, and deep learning. She can be contacted at: d.fraihat@iu.edu.jo.