Received: 9 December 2017 | Revised: 26 July 2018 | Accepted: 6 August 2018

DOI: 10.1002/spe.2639

SURVEY PAPER

WILEY

A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems

Fatima Sabir[1,2] | Francis Palma[3] | Ghulam Rasool[1] | Yann-Gaël Guéhéneuc[4] | Naouel Moha[5]

[1]Department of Computer Science, COMSATS University Islamabad, Lahore, Pakistan

[2]Sharif College of Engineering and Technology, UET Lahore, Pakistan

[3]Department of Computer Science, Linnaeus University, Kalmar, Sweden

[4]Department of Computer Science, Concordia University, Montréal, Quebec, Canada

[5]Department of Computer Science, University of Québec in Montréal, Montréal, Quebec, Canada

Correspondence
Francis Palma, Department of Computer Science, Linnaeus University, Kalmar, Sweden.
Email: francispalmaphd@gmail.com

Summary

This systematic literature review paper investigates the key techniques employed to identify smells in different paradigms of software engineering from object-oriented (OO) to service-oriented (SO). In this review, we want to identify commonalities and differences in the identification of smells in OO and SO systems. Our research method relies on an automatic search from the relevant digital libraries to find the studies published since January 2000 on smells until December 2017. We have conducted a pilot and author-based search that allows us to select the 78 most relevant studies after applying inclusion and exclusion criteria. We evaluated the studies based on the smell detection techniques and the evolution of different methodologies in OO and SO. Among the 78 relevant studies selected, we have identified six different studies in which linguistic source code analysis received less attention from the researchers as compared to the static source code analysis. Smells like the *yo-yo problem*, *unnamed coupling*, *intensive coupling*, and *interface bloat* received considerably less attention in the literature. We also identified a catalog of 30 smells infrequently reported for SO systems and that require further attention. Moreover, a suite of 20 smells reported for SO systems can also be detected using static source code metrics in OO. Finally, our review highlighted three major research trends that are further subdivided into 20 research patterns initiating the detection of smells toward their correction.

KEYWORDS

antipatterns, design smells, object-oriented (OO) systems, service-oriented (SO) systems, smells

1 | INTRODUCTION

Software systems are becoming increasingly complex due to the amount of and frequent changes in user requirements. Researchers and developers adopted approaches to address the complexity of software systems and to implement user requirements using structured and object-oriented (OO) software development. Object-oriented software development focuses on the principles of modularity and reusability. It is possible to obtain flexibility via good-quality OO design and standardized solutions, such as design patterns.[1] Developers implement design patterns according to the software requirements that are part of software architecture. A software architecture describes software elements and their

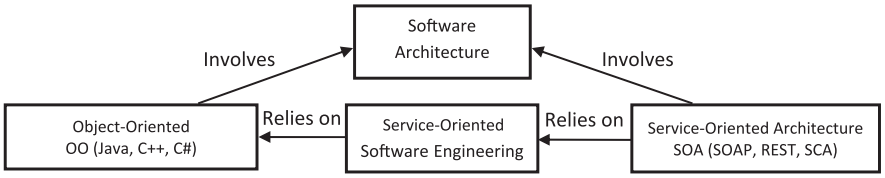

FIGURE 1 The software architecture and programming paradigms

relationship[2] at different levels of abstraction and with various forms, like classes and methods (OO) or services and servers (service-oriented architecture [SOA]).[3,4] Figure 1 shows the relationship of software architecture with the OO and SOA.

SOA is a software engineering paradigm that provides the foundation for the development of low-cost, rapid, and simple components for a distributed environment.[5,6] SOA relies on different types of services and may depend on the OO code to build complex applications.[5] The static behavior of the OO code and the dynamic nature of SOA are a challenge when maintaining the quality of service (QoS) of service-oriented (SO) systems.[5]

The interest and significance of the SO paradigm are ever increasing. Surveys forecasted that about 80% of the applications are expected to be service and RESTful (Representational State Transfer) based by 2016.* Thus, microservices are going to play a big role in enterprise-level application development. In today's distributed computing era, not only is distributed computing at its top notch but also is distributed development getting more reliable and viable development methodology. This helps the service computing paradigm dominate over the traditional development and computing paradigms. However, there are still a significant number of enterprise systems still being developed and maintained using other paradigms, for example, the OO paradigm.

Software requirements can be functional and nonfunctional. Although software developers are usually well aware of functional and nonfunctional requirements, they tend to ignore proper design guidelines during the various stages of software development under time pressure. This may lead to the introduction of smells in their systems. Software smells are poor solutions to coding and design problems. Bad smells are introduced in software systems because the developers have little to no idea of the system under design or they have very limited time to implement good design and coding practices. As the system grows, smells may lead to serious problems related to maintenance and technical debts.[7] Fowler et al defined 22 code smells in OO and their various refactoring solutions.[8]

A recent study explains well the trade-off between delivering acceptable but immature systems under the constraint of *shorter time to delivery*.[9] This study also investigates why and when the code starts to "smell bad." Releasing immature systems may lead to maintenance problems for the systems.[10] A high-quality documentation is also recommended to achieve and maintain design principles that may prevent and/or remove smells.[11] Other studies report experiments performed to assess the impact of good design on code comprehension,[12] the effect of team size on source code quality,[13] and human judgments on source code,[14] whereas studies group smells into code smells, design smells, architectural smells, and antipatterns. Smells are often reported as antipatterns[15,16] or as code/design smells,[17] which opens a debate for researchers to have a consensus on the categorization of smells.

A number of studies reported antipatterns and code smells in different types of services, eg, antipatterns in SOAP services and WSDL (Web Service Description Language) interfaces,[18-20] in REST,[21,22] and in SCA (Service Component Architecture).[23] The data sets used in these studies for their experimental validations relied on OO code. In addition, the studies performed by Mateos et al.[19,20] use contract-first and code-first techniques for the detection of smells from the Web service registries.

A literature review may not necessarily be exhaustive but should be up to date and should include all major works on a topic. The main purpose of our systematic literature review (SLR) is learning about research on bad smells in two main programming paradigms (eg, OO and SO) and answers to related research questions (eg, RQ1 to RQ5 in this SLR). We believe that a wider view of the state-of-the-art research work on bad smells provides researchers and practitioners with an opportunity to gain a broader view of the contributions and gaps in the literature. To the best of our knowledge, there is no such wide SLR study that emphasized on OO and SO, the two most common paradigms, by bringing these two paradigms together, and then investigated studies to identify potential research opportunities and existing gaps.

Since the approaches employed for the identification of smells in SO systems are also based on static and dynamic analyses, they may use approaches associated with OO smell identification. A detailed discussion on all the existing approaches is redundant here and out of the scope of this paper because reviews and comparisons of various detection approaches

*https://searchmicroservices.techtarget.com/feature/SOA-must-keep-pace-with-connected-everything-trend-in-2016

already exist in the literature.[24-27] We are interested in the evolution of the OO approaches used for the identification of antipatterns for SOA systems. We want to show the connection between the two paradigms to provide research directions for the reuse of OO approaches for the identification of smells in SO systems.

However, previous studies selected smells based on a single term query, like "code smell"[25,26] or refactoring opportunities for code smells.[24,27] Therefore, an exhaustive discussion of all types of smells is missing in the literature. We want to provide a complete and exhaustive state of the research on smells. Thus, it is essential to cover all terms associated with smells like "code," "design," "architecture," and "antipatterns." Getting rid of any terms for the selection of primary studies may not fulfill our criteria, as we define in Section 3, regarding the catalogs of smells reported by researchers in previous studies. Consequently, we carry and report an SLR that focuses on the smells and their evolution in OO and SOA and on the research trends that are covered well in OO approaches but received less attention in the SOA. We also report existing research gaps for both OO and SOA smells. We gather and analyze a set of 78 highly relevant studies addressing six state-of-the-art approaches for the detection of smells.

We extend the scope of this review for OO and SO systems by also including correction approaches after the year 2014, thus complementing a previous SLR that reported on the correction of OO smells until the year 2014.[27] Studies that focus on the refactored pieces of code are out of the scope of our SLR[28,29] because, in this study, we only consider the research works that deal with the detection of smells. This SLR may assist researchers and practitioners investigating the issues that received less attention in the literature regarding OO and SOA code smells and may lead to a new research trend by shifting the research direction for undiscovered smells that can be detected by applying existing techniques or by using novel techniques for both OO and SOA. This SLR will also help new researchers comprehend the smells and the various techniques reported on their detection.

The remainder of this paper is organized as follows: Section 2 reviews previously published research on several types of smells. Section 3 describes the method of conducting an SLR along with the review protocol followed in this study. Section 4 reports the results of the questions examined in this SLR. Section 5 discusses open issues, whereas Section 6 concludes our findings and highlights future research trends.

2 | RELATED WORK

Over the last two decades, research on smells has gained increased attention from software engineering practitioners and researchers. There are some studies reported for smells, and researchers have performed various surveys and review studies related to smells. A recent review reported different techniques for mining source code repository[26] and presented results after an analysis of the various systems used for the detection of code smells. The review in the work of Rasool and Arshad[26] presented various techniques already available for the detection of code smells along with variations in the results of these techniques implemented by different tools for the detection of smells. Moreover, the review in the aforementioned work[26] focuses only on the code smells and excludes antipatterns, design smells, and architectural smells. Furthermore, this review only focused on the detection of code smells from OO. We in this study, however, focus on all types of smells as well as smells in the SO paradigm. Another review reported software defect prediction strategies available until the year 2012.[30] All the literature mentioned above either covered a single paradigm[24-26] or the smell like *code clone*[31,32] or *design defect*[30] until the year 2012. Only one review is reported for code smells[26] in the year 2015 but again focused on smells at the code level and their detection techniques.

The state-of-the-art research works on smells reported in the literature by using various techniques from the areas of artificial intelligence, machine learning, and image processing. The relationships among the smells are classified into two categories: smells within classes and smells spreading across classes.[33] The work of Mäntylä et al[34] discusses the correlations among different smells by knowing the frequency with which these pairs appear in the same modules. A recent study proposed an approach for the detection and resolution of various kinds of smells by simplifying their detection algorithms.[35] We notice from the survey that an inappropriate sequence of refactoring may cause the introduction of defects or smells.[35] These smells are also known as antipatterns, and refactoring is applied to remove these antipatterns.[36] Therefore, it is required to check which type of smells is correlated across various paradigms and what will be the action to remove or refactor those smells from different paradigms.

Researchers presented various surveys and literature reviews in the field of software refactoring to remove smells. Mens and Tourwé[37] performed a comprehensive survey in software refactoring. Zhang et al[25] performed an SLR on code smells and discussed different refactoring approaches used for the correction of code smells after reviewing 39 studies. Another literature review presented by Wangberg[24] analyzed both code smells and refactoring after examining 46 studies.

TABLE 1 Overview of existing reviews

Ref	Time Span Covered	BS Domain (General)	BS Domain Specific	Studies Reported	Review Method	Focus
[26]	1999-2015	CS	OO	46	SLR	Tools, techniques, languages used by tools
[31]	Up to 2011	CC	OO	213	SLR	Method, tools for clone detection
[32]	Up to 2007	CC	OO/SPL/AOS	Exact studies not reported	Literature survey	Taxonomy of clone detection techniques and tools
[30]	2000-2010	SWF	OO	36	SLR	Fault prediction in units of software systems
[39]	2010-2012	SPL smells	SPL	74	SLR	Proposed techniques for SPL
[37]	1996-2003	BS	OO	Exact studies not reported	Literature survey	Refactoring activities and their roles are discussed
[24]	2000-2009	CS/DS	OO	46	SLR	Methodological, empirical contribution of code smell *w.r.t.* refactoring
[38]	2001-2012	BS/DS/AP	OO	94	SLR	Model-driven approaches to smells and their effects on model quality
[27]	2001-2013	BS	OO	47	SLR	Refactoring activities and opportunities
This study	2000-2017	BS/CS/AP/DS/AS	OO/SO/REST/SOAP	78	SLR	Focus on smell's evolution, state-of-the-art approaches and research trends in OO and SO

Abbreviations: AOS, Aspect Oriented Software Engineering; AP, antipatterns; AS, architectural smells; BS, Bad Smells; CC, Code Clones; CS, code smells; DS, design smells; OO, object-oriented; REST, Representational State Transfer; SLR, systematic literature review; SO, software-oriented; SPL, software product line; SWF, software fault.

This literature review is more beneficial in terms of empirical studies as well as refactoring opportunities that have an impact on code quality.[24] Another SLR reported existing approaches to refactoring the UML (Unified Modeling Language) model.[38] Refactoring options for a software product line is also reported in an SLR.[39] A recent study by Al Dallal[27] shows that *extract class* and *move method* are the most reported techniques for the correction of code smells.

No such SLR covers all paradigms evolved from the year 2000 until 2017 ranging from smells to code smells, and then from design smells to antipatterns for different software engineering paradigms. It is worth mentioning that different state-of-the-art techniques are also evolved across paradigms and can be applied to various research problems. The SLR presented in this paper examines overall state-of-the-art approaches for smells not only for OO but also for SO systems. We added relevant studies on refactoring techniques that are not part of the recent SLR on refactoring.[27]

We also report the research strategies that evolved from the year 2000 until 2017 across paradigms like the use of particular algorithms, source code metrics, natural language processing, and machine learning. Moreover, we also report the smells that gain maximum attention after examining the state-of-the-art approaches. This SLR will also help to know about the current state-of-the-art approaches for software engineering and discusses which smells are still uncovered in different research trends reported from the year 2000 until 2017. Table 1 summarizes the existing SLR for OO and SO paradigms.

As Table 1 shows, existing reviews discuss either refactoring approaches or detection approaches. The search terms associated with these approaches are mostly based on code smells. No such review discussed detection techniques and their evolution that may help researchers investigate smells for SO systems. Moreover, refactoring approaches also focus on either techniques used for refactoring[27] or modeling techniques used for refactoring. Furthermore, research regarding the impact of smells on different issues like maintenance, fault proneness, and the lexical impact of code is uncovered. All these areas are comparatively new and mostly reported after the year 2013 for OO and SO systems. Most of these reviews focus on code smells and do not consider the state-of-the-art techniques for architectural smells and antipatterns. This may give the reader an incomplete review that discovered some smells reported as architectural smells or antipatterns. We are unable to find any review that focuses on SO systems and techniques used for those that also evolve in OO and SO paradigms. Previous studies have focused on classifications but do not discuss research trends that may help new researchers initiate investigations on these smells.

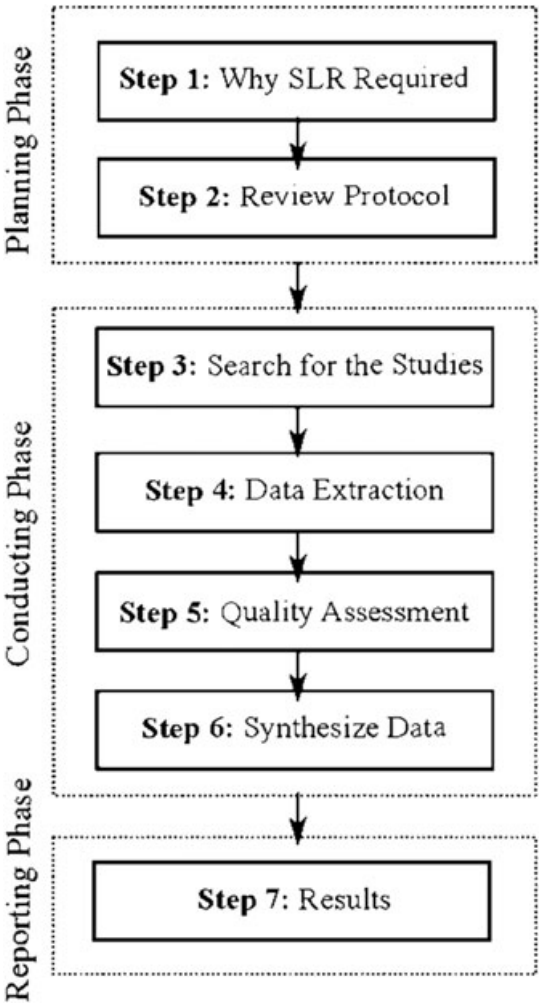

FIGURE 2 Steps followed for a systematic literature review (SLR)

3 | RESEARCH METHOD

This SLR reports the existing state-of-the-art approaches on smells from different software engineering paradigms. Brereton et al suggested software engineering researchers apply evidence-based software engineering.[40] The evidence-based research was primarily introduced in the medical domain because expert opinion–based medical service is not as reliable as advice-based health care services. In addition, to collect all relevant facts on research questions, performing an SLR may also help practitioners find existing research gaps. We follow the guidelines proposed by Brereton et al[40] to perform this SLR in three main steps: planning, conducting, and reporting as shown in Figure 2.

This section describes the protocol we follow to perform this review. We also ensure to reduce the chances of search bias. The protocol includes the selection of most appropriate research questions, rules for the study selection criteria, identification of different studies, classification of studies, classification of dimensions for the attributes, and, finally, the results of data extraction and analysis.

3.1 | Planning the SLR

The main goal of evidence-based software engineering is to collect the most relevant evidence from research and investigate the findings of evidence to evaluate research problems. The state-of-the-art smell techniques are evolved in OO and SO paradigms. Identification of the existing review that is available for this paradigm is reported in Section 2. However, none of the previously published reviews are similar to the review presented in this study. Most of the reviews were based on code smells from the OO paradigm. In this SLR, we used the following terms to search for the primary studies.

TABLE 2 Research questions

ID	Research Question	Motivation
RQ1	What are the classifications of the state-of-the-art techniques employed in the detection of smells?	Identification of smell detection techniques followed by their classifications.
RQ2	How the state-of-the-art approaches evolved across different paradigms starting from object-oriented to service-oriented?	Evolution of specific techniques in object- and service-oriented systems.
RQ3	What are the smells that are reported for a specific paradigm?	Identification of unique smells for a specific paradigm.
RQ4	What is the correlation between smells across the paradigms?	Smells that are repeatedly reported for different paradigms.
RQ5	What are the trends in research for smells from the year 2000 to 2017?	Research trends followed in the domain of smells.

((‘*smells*’ **OR** ‘*code smells*’ **OR** ‘*design smells*’ **OR** ‘*architectural smells*’ **OR** ‘*antipatterns*’ **OR** ‘*antipattern*’ **OR** ‘*anti-pattern*’ **OR** ‘*anti-patterns*’) **AND** (‘*OO*’ **OR** ‘*services*’ **OR** ‘*SOC*’ **OR** ‘*SBS*’ **OR** ‘SOA’ **OR** ‘*service-based system*’ **OR** ‘*object-oriented system*’))

The search string is searched from the keywords, abstract, and title of each study from the year 2000 until December 2017. Table 3 reports the results of our search.

***Review protocol*:** In the following, we show the general criteria followed in this study to provide a more consistent and focused review. We specify the research questions with the help of the following PICOC (Population, Intervention, Comparison, Outcome, Context) criteria[41]:

- *Population:* OO software engineering (OOSE), SO computing, SO systems, services, REST, SOAP, WSDL;
- *Interventions:* smells, design smells, architectural smells, code smells, antipatterns, anti-pattern, anti-patterns;
- *Comparison:* a holistic comparison of the population to analyze the impact of recent research on smells, solutions, methods, and techniques;
- *Outcomes:* a classification of state-of-the-art smell techniques that are used to identify or correct smells across paradigms;
- *Context:* an exclusive focus on evidence collected from the state-of-the-art techniques on smells.

Through this SLR, we try to answer five research questions as stated in Table 2.

3.2 | Conducting the SLR

This section presents the review protocol required to perform our SLR. We search for the relevant literature to conduct the SLR.

3.2.1 | Search process for studies

An effective search string is essential to select the most relevant studies. There is no such clear consensus on the types of smell as design, code, and antipatterns. Therefore, we first go through the relevant reviews presented for smells to avoid any overlapping, and then, we expand this review for OO and SO systems. We also check the most relevant keyword for the review and check their synonyms, hyponyms, and alternatives. We rely on the Boolean operators like “AND,” “OR,” and the wildcard characters (*) to formulate our search string. As we want to cover all types of smells starting from the term code smell to design smell, then to architectural smell, and, finally, to antipatterns, we therefore use each search term associated with smells starting with the help of the wildcard character (*) and “AND” operators to include the relationship between population and intervention. General terms related to smells were searched from different digital libraries along with keywords and full term-based search. Table 3 reports the result of each term associated with smells.

3.2.2 | Study selection

To select the most relevant research studies, we applied a three-step process.

Step 1: We extracted 13 769 studies resulting from the generic keyword-based search strings from different digital libraries. The keyword-based search also reports the articles from requirement engineering and performance antipatterns. Initially

TABLE 3 Number of studies found in selected digital libraries after a general term search

Sl No	Term Search	IEEE	ACM	Science Direct	Wiley	Springer	Total
1	Code smell, code smells, code flaws	195	1586	86	1	32	1900
2	Antipattern/anti-pattern/antipatterns	48	1719	125	0	1	1893
3	Design smell, design smells, design flaws	135	6550	84	1	8	6778
4	Architectural smell/architectural smells	27	2048	382	0	0	2457
5	Smells	58	54	100	49	480	741
Total		**463**	**11 957**	**777**	**51**	**521**	**13 769**

Abbreviations: ACM, Association for Computing Machinery; IEEE, Institute of Electrical and Electronics Engineers.

extracted studies are further refined for the domain of software engineering, resulting in 2669 studies left in the pool for review. A majority of the studies are removed from the ACM (Association for Computing Machinery) library because terms associated with the research strings are also available in the domain other than software engineering.

Step 2: The collection of studies selected in Step 1 is further refined manually by covering index terms, abstract, title, and their application domains (OO and SO). This process removes all studies from the domain of requirement engineering and Android applications containing various terms related to smells. Duplication is removed among research studies from the selected databases. The resultant provided 540 studies out of 2669 based on their matching definitions of smells related to design, code, and architecture.

Step 3: Furthermore, studies are filtered following some *exclusion* and *inclusion* criteria. Only the studies from well-known conferences are kept, and the rest is discarded. The *inclusion* criterion is based on the following.

- Journal articles are selected related to the domain of OO analysis, software maintenance, reverse engineering, information and software technology, SOA, and Web service.
- Top-level conferences are selected when related to software maintenance, reverse engineering, OO technology, evidence-based software engineering, and SO computing.
- In this SLR, we include all studies associated with the term smells (eg, code, design, antipatterns, and architectural smells).
- Contextual data for each study are provided in the Excel sheet available online.§§§§§

On the other hand, the *exclusion* criterion includes the following.

- Articles of short length (less than five pages).
- Book chapters are not included.
- Workshop articles and lecture notes are not included.
- Software performance antipatterns and software requirement antipatterns are not added due to their irrelevancy to our target domain as we are working on the evolution of smells in OO, and services and requirement and performance could not be part of the evolution from one system to another.
- Research works published as a technical report.
- Research studies related to code clone, duplicate code, and copy-paste programming are not added since reviews exist for them.
- Smells related to android systems are not added since, in this SLR, we are covering only the paradigm of OO and SO.
- Research studies that discussed single smell are also not added because we want to know which smells are mostly discussed by tools, industry, and academia.

After applying Step 3, only 75 studies are left that satisfy the abovementioned inclusion and exclusion criteria. Finally, the snowballing method[42] is applied to check the reference list of the selected studies to minimize the chance of removal of any relevant studies. Therefore, in an additional activity, 78 studies are selected in Step 4. Snowballing provides an additional three studies[43-45] mostly cited in different research studies and not included in selected searched databases. Figure 3 shows the representation of study selection criteria.

The identification of smells is performed following an incremental process. In the first phase, we start with a primary study and collect information on all reported smells. We then follow the process across all the primary studies for different domains, and, finally, we get a pool of smells for a specific domain. In the second iteration, we run the process for identification of smells and check whether these smells are already "reported" or "detected" or "corrected" in the area other than OO; if yes, then we add those smells into the correlated smell section to check what types of smells are evolved

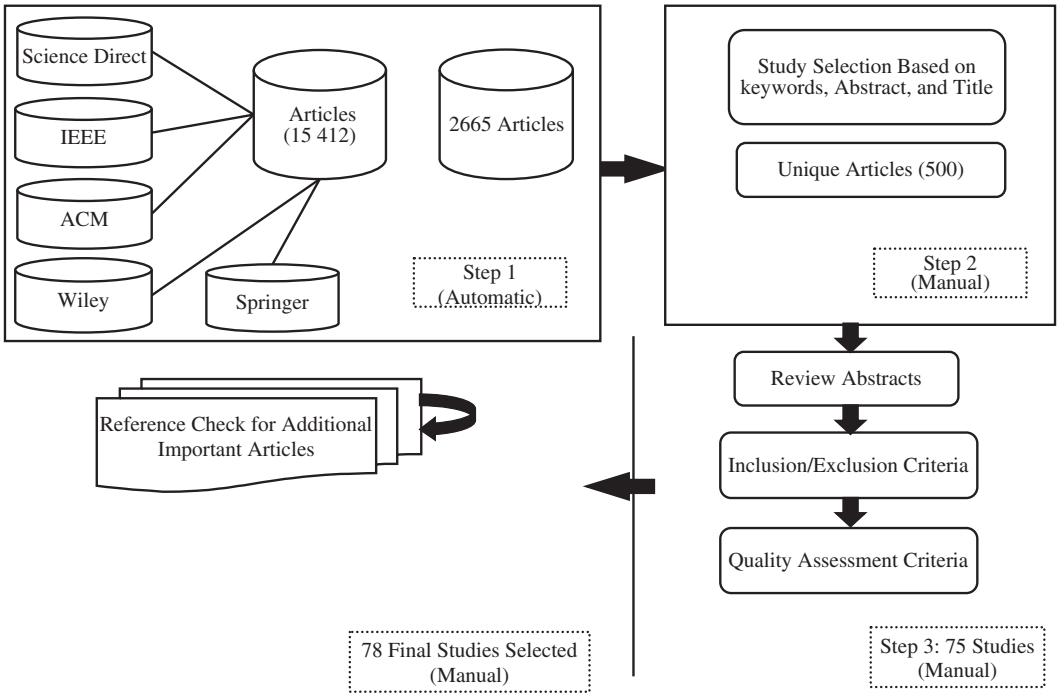

FIGURE 3 Study selection criteria. ACM, Association for Computing Machinery; IEEE, Institute of Electrical and Electronics Engineers

across paradigms. If we are not able to find that smell as "corrected" for SO systems, then we report this smell as "not corrected." A similar iteration is followed for techniques evolved in OO and SO to check the approach followed in a sequence from the year 2000 until 2017. This iteration will also help identify trends in the research on smells.

3.2.3 | Quality assessment

A number of techniques have been widely used by the academia for the quality assessment of studies in an SLR, for instance, Dybå and Dingsøyr[46] used the *Critical Appraisal Skill Program* and Al Dallal[27] used Kitchenham guidelines for quality assessment. In this SLR, we followed the PICOC[47] and Brereton et al[40] guidelines to prepare the quality checklist as listed in Table 4. The checklist can further be divided into four basic quality aspects, including (1) the criteria to design a study, (2) the method that describes the setup of the study, and (3) how the study is performed, the final conclusion is drawn from the study, and implication contributions for the academia or industry. The quality assessment is based on the response of "Yes," "No," and "Not Applicable." Our SLR focused on the OO and SO paradigms. Therefore, some of the quality assessment questions are not applicable to some studies and lie in the response group of "Not Applicable." Our 78 primary studies are evaluated based on the quality checklist questions mentioned in Table 4.

The results derived in the quality assessment step demonstrate that the majority of the studies (ie, 74) clearly mentioned about the bad smell identification techniques in contrast to the four studies that did not illustrate any discussion on the identification techniques. These studies performed SLRs[26,27] or discussed the applicability of machine learning techniques for the identification of bad smells.[15,48] All of the selected primary studies are classified completely based on the bad smell identification techniques as they are further divided into two main categories and six subcategories. We also found 22 primary studies that evolved from one domain to another and three primary studies[15,26,27] that are not applicable to the bad smell evolution, as they belong to both the literature review and machine learning techniques. There are 55 studies that we investigated, the complete analysis of which is discussed in Section 3. Most of the primary studies justified their research methods. However, we noticed that some of the studies did not compare their results with state-of-the-art techniques and belong to the domain of SO software engineering. These are the earliest research papers that performed the detection of bad smells in the service domain and partially validated that their results as service-based systems are not open-source and that it is difficult to perform the validations. Among the 78 primary studies, a maximum number of studies from the OO paradigm, particularly on the detection of code smells, ie, 35 studies plus 15 other mixed studies, contributed to the existing literature in a similar way as mentioned in RQ3 and RQ4. We also found some unique studies from the domain of RESTful APIs (application programming interfaces) and on the impact of bad smells on software maintenance that

TABLE 4 Quality assessment criteria and results of primary studies

Design	Questions to be Asked	Yes	No	NA
QA1	Did the study state bad smell identification techniques clearly?	74	4	0
QA2	Are the primary studies classified based on the bad smell identification techniques?	78	0	0
QA3	Did the research methodology evolve from one domain (OO) to another?	20	55	3
QA4	Did the study classify based on the types/domain of bad smells?	75	0	3
Conduct				
QA5	Did the data under analysis generated completely address the problem statement?	56	0	22
QA6	Did the study evaluate the proposed method and are the results explained well?	56	22	0
Analysis				
QA7	Are the data sets clearly mentioned in primary studies?	64	9	5
QA8	Did the study apply any accuracy measures for evaluation?	26	35	17
QA9	Did the study explain the methods of statistical measure implementation?	43	35	0
QA10	Was the statistical technique of evaluation justified?	35	40	3
QA11	Was the method used for the research justified?	77	0	1
QA12	Did the study clearly state the purpose of analysis?	78	0	0
Conclusion				
QA13	Did the primary study explain well all mentioned research questions?	50	15	13
QA14	Did the study compare their results with state-of-the-art techniques?	56	20	3
QA15	Did the study explain negative findings or discuss limitations?	72	2	4
QA16	Did the study explain well the validity threats?	71	3	4
Implications				
QA17	Did the primary study contribute to existing techniques?	50	8	20
QA18	Did the primary study identify any new area of research?	18	26	34
QA19	Did the researchers discuss their findings in terms of contributions toward academia and industry?	56	20	3

Abbreviation: OO, object-oriented.

provided new notions to the existing techniques that provide information on RQ4 and RQ5. The quality assessment criteria for QA16, QA17, and QA18 suggest meaningful information for the practitioners and academia that we discuss in detail in Section 5. The quality assessment criteria for QA8, QA9, and QA10 belong to the validations of the tools/techniques used in the primary studies. Moreover, QA14, QA15, and QA16 are also not applicable for some studies,[26,27] as they discussed their findings from the literature. Table 4 shows our quality assessment criteria and the findings based on those criteria. The detailed evaluation of the primary studies is available online.[§§§§§]

3.2.4 | Data extraction and analysis

We extracted data in Excel in a consistent format as presented in our online appendix,[§§§§§] where we also present detailed results. Data are extracted based on the research questions. We focus on the types of smells, ie, code, design, architectural, and antipatterns, as they appear in the title, abstract, or index terms of an article. Our classification technique is based on static, dynamic, empirical, methodological, and linguistic source code analysis. Research trends are collected after examining the sequence of related research patterns over the past 17 years. Data extracted are evaluated, and disagreement was discussed until the conclusive results are achieved. Many selected studies did not answer all the questions available in our data extraction form. Table 5 presents the data extraction sheet designed for each research question.

3.2.5 | Data synthesis

Quality criteria are based on the inclusion-exclusion criteria as defined above. Metadata analysis is performed after reviewing the studies completely. Metadata analysis for each research question is clearly examined, and the answer is recorded in an Excel sheet as presented online.[§§§§§] We have verified basic contextual information reported for each research question. A number of approaches are available for data synthesis, some of which maintain the qualitative form of the evidence such as meta-ethnography, whereas some involve converting qualitative findings into a quantitative form such

TABLE 5 Data extraction sheet

Search Criteria	Data Item	Description
General	Identification number	Reference number assigned to the article
	Bibliography	Year, Title, Source/Research Group
	Type of the article	Conference Paper, Journal Paper/Tech Report
	Study aims	Summarizing notes about each study
	Study designs	Experimental, Case Study, Survey, Review
RQ1	Behavioral source code analysis	Source code analysis that uses source code metrics to examine source code behavior
	Dynamic source code analysis	Analyzing the interrelationship of program entities after the execution of and checking the behavior of the program
	Algorithm-based analysis	Studies that use a specific algorithm to detect smells from the source code
	Empirical source code analysis	Studies that report the results of already established tools to empirically evaluate the research problems and address some new findings
	Methodological analysis	Implements the already proposed methodology in a new way to either correct or detect smells. These types of analysis compare the results before and after the implementation of any specific method.
	Linguistic source code analysis	Checking the internal code quality like naming conventions of methods, classes, etc
RQ2	Evolution of research for smells in OO and SO	Analysis of multiple research methodologies constantly repeated from paradigm to paradigm
RQ3	Smells reported for a specific paradigm	Unique identification of smells for OO, SOAP services, REST, SCA, AOP, SPL
RQ4	Evolution of smells in OO and SO	Identification of smells that are reported for OO, but also later found in SO systems
RQ5	Trends in research	Unique trends for smell Detection, Correction, Maintenance, etc

Abbreviations: AOP, Aspect Oriented Programming; OO, object-oriented; REST, Representational State Transfer; SCA, Service Component Architecture; SO, software-oriented; SPL, software product line.

as content analysis.[49] Basic quality criteria for selecting studies discussed above are based on the guideline provided by Kitchenham et al.[50] There are different terms reported for the smells in the literature, but most of the smells are reported as code smells (22 research studies), and only nine studies are reported as antipatterns. Data addressing our five research questions are extracted from the 78 most relevant studies that satisfy all the quality criteria, including the PICOC and the inclusion/exclusion checklist. Our goal is to collect the most relevant data from the studies selected to analyze the state-of-the-art approaches in OO and SO paradigms. To investigate the key questions, three sets of data were extracted from 78 studies.

1. *Context data* showing the context of each study, such as the source of data, experimental evaluation, application area, and programming languages, are noted.
2. *Qualitative data* analysis based on the cause-effect relationship or reporting new ideas by using different properties of the system under analysis.[51]

    a. *Ethnographic studies* discuss the role of particular issues and describe the cause-effect relationship like the discussion on the literature review and findings from the literature to describe new opportunities like code smell mining[26] or refactoring opportunities[27] or empirical evidence from history.[52,53]
    b. *Grounded theory* used induction or observation and uses interviews, surveys, or observations.[54,55]
    c. *Phenomenology* focuses on the "subjective reality" of an event or perceived by the study population as reported in other works.[56-58]

3. *Quantitative data* extracted from the studies based on the predictive performance of the model or approach reported in the study. The data are divided categorically, and the variable used to represent the result is mostly continuous. However, some of the studies reported their results in both forms.

   a. *Categorical studies* report their results predicting whether the smells are detected, corrected, or maintained in the system under analysis. These results are reported using accuracy measures like precision or recall. In total, 26 studies used accuracy measures like precision and recall in this SLR.
   b. *Continuous studies* reported their results by using similar measures like the mean standard error or measuring the difference between expected and observed results like chi-square, correlation, logistic regression, and ranking form. We found 21 studies that fall under this category, as they report their results by using statistical techniques to validate their research model and present their findings. The most widely used technique for continuous studies is correlation analysis (six studies) followed by regression analysis. Table 6 shows the complete list of studies in this category with the techniques applied.

The distribution of studies with respect to the types of data they used is shown in Figure 4. We also show the frequency of terms related to smells, eg, "antipatterns," "architecture," "unpleasant smell," "code smell," "design smell," and "code and design" from the literature in Table 7.

It is important to consider that 22 studies are reported as "Others" because they used a mixed approach and did not report their results by using any measures. These types of studies have mostly focused on the key concepts that are proposed but were not tested or validated. The complete information about these studies is provided online.[§§§§§]

TABLE 6 Techniques for continuous studies

Ref No	Technique Used
[35]	Fisher's exact test
[59]	Mean, Median, SD, Correlation
[20,60,61]	Correlation
[62,63]	Wilcoxon signed-rank test
[64]	Correlation
[65,66]	Chi-square test
[53]	Regression analysis
[67]	Cliff's D and Kruskal-Wallis test, Holm's Mann
[45]	Cohen's kappa, Fleiss' kappa
[68]	Correlation analysis, Regression
[69]	Wilcoxon signed-rank test, Mean, Standard deviation, Median
[68]	Logistic regression model
[70]	Min, Median, Mean, Mode
[71]	Proportion, Odds ratio
[72]	PCA, Logistic regression
[43]	Logistic regression model, Odds ratio
[73]	Fisher's exact test, Odds ratio, Chi-squared

Abbreviations: PCA, principal component analysis; SD, standard deviation.

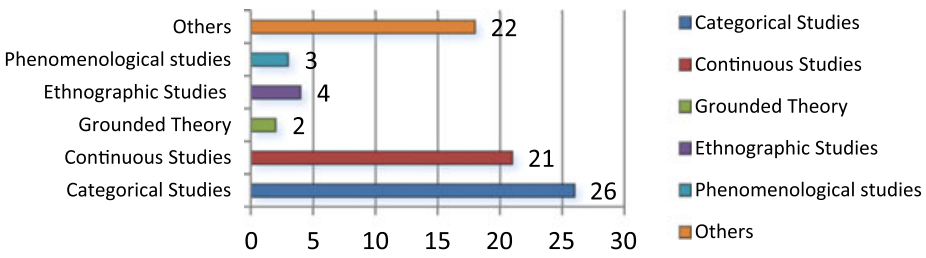

FIGURE 4 Distribution of studies with respect to the data analysis techniques [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 7 Frequency of smell terms in studies

Sl No	Smell Type	Frequency	%	Reference
1	Antipatterns	11	14.1	35,43,48,54,56,73-78
2	Architecture	15	19.23	18-23,59,60,62,79-82,83,84
3	Unpleasant smell	7	8.75	14,45,55,64,85-87
4	Code smell	24	30	26,27,41,52,53,58,61,63,68,69,72,88-99
5	Design smell	18	23.5	15,16,44,57,65,67,70,71,100-109
6	Code and design	3	3.85	7,81,110
	Total	**78**	**100**	

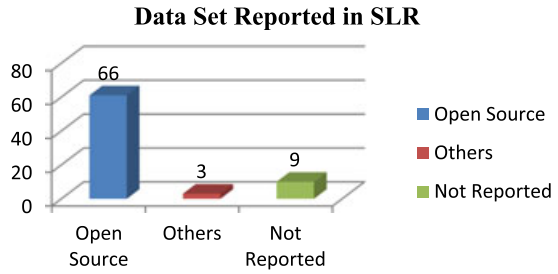

FIGURE 5 Data source used for the state-of-the-art research on smells. SLR, systematic literature review [Colour figure can be viewed at wileyonlinelibrary.com]

Data synthesis is combined with the data extraction form to analyze the quantitative as well as qualitative data fully. The data extraction form as reported in Table 5 provides complete information regarding each research question along with the data synthesis reported in Figure 4. The information presented helps to look for the most applied statistical methodology used by the industry for categorical, continuous, and qualitative studies. In total, 22 studies used a mixed approach based on the quantitative and qualitative information. Some of them only discuss the concept or novel approach and present findings or benefits of their approach.

Moreover, it is also observed that most of the studies have validated their research model on open-source systems. Therefore, most of the results in the area of smells can be compared or tested by analyzing similar open-source systems. Figure 5 reports 66 primary studies that use open-source systems and three other studies that use proprietary systems, ie, other than open-source systems. We also found nine studies that do not rely on any target systems, open-source or proprietary, to validate their results.

The validation criteria reported by the studies are either based on accuracy as a measure or by using statistical techniques. We have found the highest number of research studies between the years 2013 and 2015 (18 studies). Moreover, the research trend has been moved from static source code analysis to dynamic source code analysis, applying machine learning, artificial intelligence, and genetic algorithms. Furthermore, smell detection has not gained much attention for SO systems. Figure 6 shows the distribution of studies over the years. Fowler et al[8] introduced the concept of code smells in 1999, and the first paper reporting smells was published in 2001,[111] followed by four studies in the year 2004 and two studies in 2006. These are conference studies, and we are unable to find any journal studies in those years that fulfill the criteria of selection for studies. The problem of identifying smells in the source code began to attract more research attention in the year 2010 with an average of more than six research studies per year. This observation highlights researchers' interest and the importance of smells after the year 2009.

The research studies were published in 39 different venues. Over half of the studies are published in conferences, and the rest are published in various journals. A slight shift from conference to journal articles shows the importance of considering both conference and journal articles in this systematic review. Also, considering either journal articles or conference articles will create a research bias and may provide the readers with an incomplete literature review. Furthermore, researchers are attracted more toward the conference of reverse engineering, software maintenance, whereas for services, SO computing attracted more researchers.

There is less number of studies reported for SO software engineering compared to OO. This shows that research is now shifting toward SO systems due to the high demand for Web services. Most of the studies for SOAP services are reported in the International Journal of Web and Grid Services (four studies) and IEEE Transactions on Service-Oriented Computing

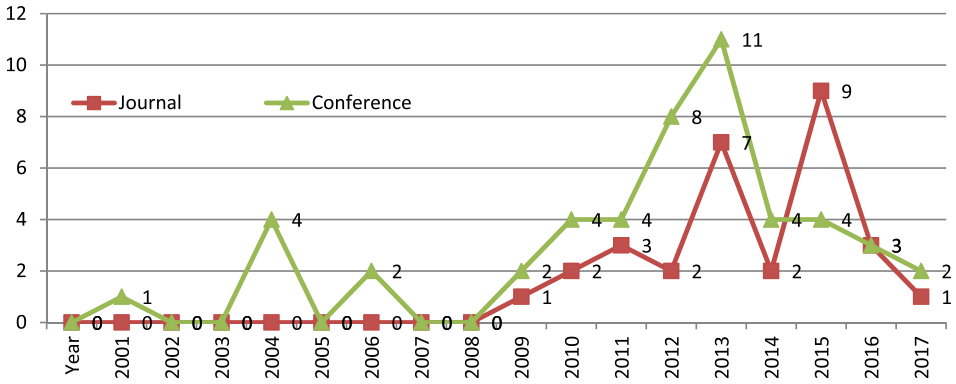

FIGURE 6 Year-wise distribution of studies [Colour figure can be viewed at wileyonlinelibrary.com]

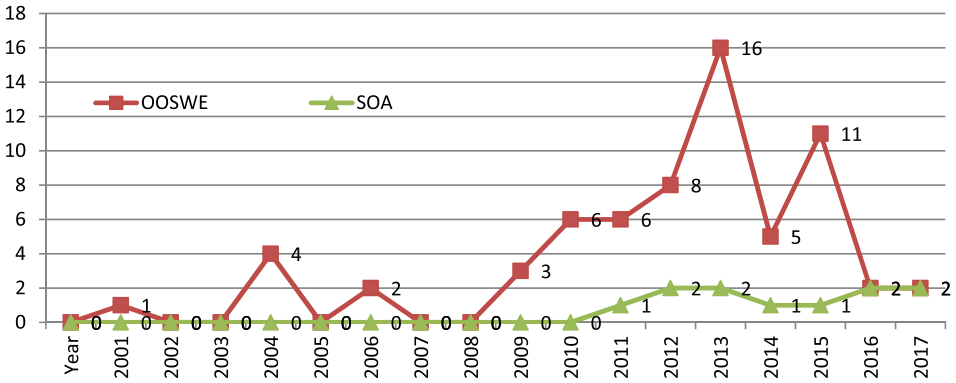

FIGURE 7 Year-wise distribution of studies with respect to different paradigms. OOSWE, Object Oriented Software Engineering; SOA, service-oriented architecture [Colour figure can be viewed at wileyonlinelibrary.com]

(one article). Moreover, we are unable to find any journal paper on REST service smells as well as the correction of smells for REST services. This observation shows that this is a highly active area of research for the new researchers. Figure 7 presents studies over the past 16 years for the paradigms of OO and SO.

The authors of all the above studies are from academia and mainly working in research groups with support from industry. Therefore, on the basis of this study's selection criteria, no strong evidence was found that gives strong implication whether research on smells is primarily conducted by the industry or the academic community. We also found some references where academia solves the industry problems after collecting information from industry blogs like J2EE[†] and INFOQ[‡],[18,22,80] and that provide tool support that solves the reported problems by industry blogs.

4 | RESULTS AND DISCUSSION

RQ1: What are the classifications of the state-of-the-art techniques employed in the detection of smells?

The classification of the detection techniques for smells is also reported in a recent review,[26] but it is focused only on the smells discussed by Fowler et al.[8] This focus creates a bias because smells other than Fowler's are not reported. Moreover, the selection of studies covers January 2000 until December 2017.

Another SLR used the term "smell" for selecting studies, but focused only on studies that discussed multiple refactoring operations.[27] It focused only on the refactoring for removing smells and not on the impacts of smells on system performance and multiple approaches used for the detection of smells. We also found studies that discussed model-driven approaches for smells until the year 2011[39] due to their research associated with the detection of smells. Moreover, we

[†]www.j2ee.com
[‡]www.infoq.com

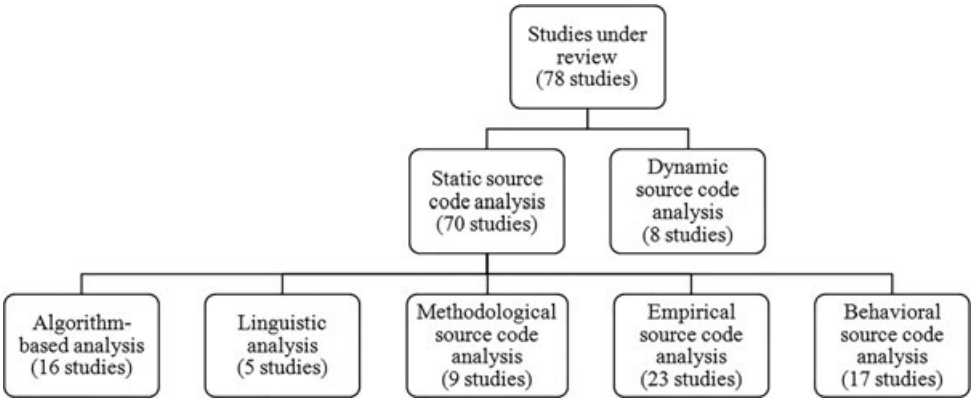

FIGURE 8 Distribution of primary studies based on source code analysis

also discussed studies that introduced correction approaches in addition to the detection techniques that were not considered in the most recent review.[26] We divided the studies based on source code analysis and not based on the symptoms associated with smells as reported in the work of Rasool and Arshad.[26] The classification is divided into (1) static source code analysis and (2) dynamic source code analysis.

Static source code analysis is a technique that examines the properties of smells and their impact without executing systems. In contrast, the dynamic source code analysis examines the cause-and-effect relationship of smells during system execution.

In the following, we group primary studies selected for this review into two major types: (1) dynamic source code analysis and (2) static source code analysis. Most of the studies focused on static source code analysis as we found 70 primary studies under this category compared to eight for dynamic source code analysis. Figure 8 reports the total number of studies reported for each type of analysis.

1. Static source code analysis

We further divide the group of static source code analysis into six subcategories as shown in Figure 7.

a. Behavioral source code analysis

Software systems undergo various changes, and a study[28] reports that 98% of the literature on change impact analysis is related to code analysis in comparison to 17% of the studies to architectural changes. We group multiple primary studies into *behavioral source code analysis* that examines the program behavior without executing the source code. The behavioral source code analysis uses different metrics to check program behavior like cohesion, coupling, depth of inheritance, and lines of code using various source code metrics.

The detection of smells is not possible without any intermediary representation. This representation is used to extract useful information from the application and to apply source code metrics to check the detection of smells. Therefore, behavioral source code analysis is based on either informal description of flaws[102] or using the textual description of rules with the help of Domain Specific Language (DSL).[17] Source code parsing is also applied using different parsers[20,59,60,83] that make an intermediary representation of source code and apply source code metrics directly to this intermediary representation. Some code smells are also reported in the literature after examining the version control histories and then apply source code metrics.[92] These types of studies help industry and academia to investigate the problems for a system under analysis to improve its quality.

We also observed that metamodel[101] and ontology[110] are other forms of intermediary representations on which source code metrics are applied to detect smells. Moreover, smell identification could be done by specifying them either in the form of textual description[64] that helps improve the WSDL document describing SO systems and OO systems[100,103,106] or in the form of UML specifications[75] to improve the source code quality. Table 8 highlights the conditions used by the primary studies for smell detection. It provides the primary studies reported for OO and SO systems.

b. Empirical source code analysis

Empirical source code analysis has been the technique to get information by using already established tools or approaches directly or indirectly. Empirical source code analysis is analyzed either quantitatively or qualitatively.[113] The evidence collected from different techniques will help researchers answer questions that are clearly defined and collected from different problem domains.[113]

TABLE 8 Studies used behavioral source code analysis

Ref No	Smell Type	Precondition	Post Condition
102	DS	Informal description of flaws	Source code metrics
92	CS	Change history extractor using SVN[§] and CVS[¶]	Code smell detector applies for each smell type
17	CS/DS	Textual description is used for DSL to generate detection algorithms	Source code metrics
112	CS/DS	Parsing using JFLEX[#] and Java Cup[‖]	Source code metrics
88	CS/DS	None	Source code metrics
101	DS	Metamodel	Source code metrics
110	CS, AP	Ontology	Source code metrics
44,64,100,103,106	DS, BS	Smell properties are applied	Source code metrics
75	AP	UML specification is defined for each AP	Correction approaches are defined
20,59,60,83	AS, AP	Java to WSDL file	Source code metrics

Abbreviations: AP, antipatterns; AS, architectural smells; CS, code smells; CVS, Concurrent Versions System; DS, design smells; DSL, Domain Specific Language; SVN, Sub Versioning Number; UML, Unified Modeling Language; WSDL, Web Service Description Language.

[§]https://subversion.apache.org/

[¶]http://www.nongnu.org/cvs/

[#]http://jflex.de/

[‖]http://czt.sourceforge.net/dev/java-cup/

Some smell detection techniques use empirical evidence collected by different already established tools to check the associations among various versions of OO systems[29,35,62] or an SLR to check the most relevant information presented for either detection of code smells[26,27,72] or correction of code smells[109] or code decay.[52]

Empirical source code analysis mostly investigates the cause-effect relationship like smell versus maintenance effort,[53,91] best practices of services in an open-source platform,[77] class performance,[68] change proneness of antipatterns and clones,[73] refactoring suggestions[58,67,68] after smell detection, or statistical techniques to check the relationship between already presented smell and design patterns.[69] Table 9 summarizes the extracted information of 23 studies that rely on the empirical source code analysis techniques.

c. Algorithm-based source code analysis

Some smell detection methods use more than one detection technique, like source code metrics. Some use genetic algorithms,[71,94,96] machine learning techniques,[15,87] or image processing.[57] These algorithms help solve the problem of using a fixed threshold for the detection of smells[71] and correction of smells using development history.[96] Moreover, detection results from different repositories are used to implement machine learning techniques.[15,87] These studies also open an opportunity for the detection of smells and correction using machine learning algorithms with good precision and recall as reported in our online appendix.[§§§§§] Table 10 summarizes the information of all studies that use different algorithms.

d. Methodology-based source code analysis

Methodology-based source code analysis is used to implement an existing methodology in an alternative manner for either detecting or correcting smells with the end goal of improving detection accuracy.

This type of source code analysis can be used to define a unique technique relying on already available tools or algorithms to detect occurrences of smells. The studies in this aspect mainly focus on the quality of the source code before and after implementation of specific methodologies, like performance comparison of different queries using some source code metrics[74] or using different tools for smell detection.[97]

Moreover, the NSGA-II (nondominated sorting genetic algorithm) algorithm is also used to check source code quality before and after applying algorithms for the correction of smells.[84] The effect of smells on system defects was also studied by considering five types of smells.[41] This study showed that the *switch statement* has more influence on defects in comparison to other smells under study. The studies reporting empirical source code analysis are presented in Table 11.

e. Linguistic source code analysis

This technique of source code analysis through linguistic quality assessment started in early 2015. There are only three studies published in the OO paradigm that investigated linguistic antipatterns. These antipatterns are erroneously introduced in the code when using wrong naming conventions of methods, classes, and variable names. The detection of linguistic antipatterns is a new area of research receiving growing attention in the software engineering (SE) research community. We were unable to find any study that reports the effect of linguistic antipatterns on system performance. Moreover, it is required to implement correction approaches for linguistic antipatterns.

TABLE 9 Empirical source code analysis techniques

Ref	Smell Types	Precondition	Post Condition
29	AP	Mining the source code	Twelve versions of Eclipse[**] and nine versions of ArgoUML[††]are used to mine the repositories for antipatterns and code smells
16	AP	Change proneness and null pointer exception in classes are analyzed	Results are validated in 11 releases of Eclipse
35	AP	DÉCOR[‡‡] along with the PTIDEJ[‖] tool suite to check static dependencies	Macocha[§§] mines version control systems for checking association among antipatterns
62	AS	SO system FraSCAti[¶¶]is used for evaluation purposes	Python script is used to check the commit with changes and without changes
93	CS	Mimec[56] is used to record log	Twelve code smells detected in the premaintenance version using Borland together and InCode
53	CS	Borland together and InCode[##] for the detection of code smell	Mimec[56] is used to check the developer's activity, and then, maintenance effort is analyzed
91	CS	Three developers are hired to perform maintenance tasks	Regression analysis is carried out to measure effort
43	CS	Changes are counted using CVS from Eclipse	Logistic regression is applied to correlate the presence of antipatterns with the change process
67	AP	Antipatterns are detected, the MaDUM[114] matrix is used to check the cost	Refactoring options are suggested to remove smells
65	DS	Design smells are analyzed *w.r.t.* flaws classes	Design defects vs flaws classes
56	AP	DÉCOR is used for the detection of code smells	Odds ratio and Fisher's exact test are applied to check the difference between mutated and nonmutated antipatterns
76	AP	A set of guidelines is defined for WSDL	Easy SOC plug-in developed for detection
61	CS	Manual analysis of smell frequency and source code metrics for the detection of smells	Correlation is used to check the further relationship
89	CS	Eclipse plug-in is developed to check the location of source code	Naïve Bayes and association rule mining are applied to check the bug relationship in code
58	DS	InCode[‖‖] as an Eclipse plug-in to check the quality of the code	Refactoring suggestions for code quality
45	CS	Different-smell detector is used to check smells in code	Kappa[***] statistics are applied to check results
68	BS	Mining the evolution of three open-source projects	Quantitative analysis of the classes participating in refactoring
69	CS	Nine design patterns and seven code smells are analyzed	Correlation, mean, and median are used to answer the research problem
26,27,52	CS	Kitchenham guideline to collect relevant literature	Reported code smell detection and correction approaches
73	AP	DÉCOR for antipattern detection and CC finder for clone detection	Cochange analysis and fault-proneness identification with the Macocha model
77	AP	A literature review of services collected	Analysis of the Google Cloud platform, Stack overflow, OCCI for best practices

Abbreviations: AP, antipatterns; AS, architectural smell; CS, code smell; CVS, Concurrent Versions System; DS, design smell; OCCI, Open Cloud Computing Interface; MaDUM, Minimal Data Members Usage Matrix; PTIDEJ, Pattern Trace Identification, Detection, and Enhancement in Java; SOC, Service Oriented Computing; SC, source code; SO, software-oriented; WSDL, Web Service Description Language.

[**]https://eclipse.org

[††]http://argouml.tigris.org/

[‡‡]http://ptidej.net/tools

[§§]http://www.ptidej.net/

[¶¶]http://frascati.ow2.org/

[##]https://marketplace.eclipse.org/content/incode-helium

[‖‖]https://marketplace.eclipse.org/content/incode-helium

[***]https://en.wikipedia.org/wiki/Cohen's_kappa

TABLE 10 Algorithm-based source code analysis

Ref No	Smell Type	Algorithm Domain	Precondition	Post Condition
[15]	AP	ML	60 source code metric results are used as training data sets from PROMISE[†††]	SVM classifier is applied using WEKA[‡‡‡]
[80]	AS	GA	Base examples are collected from different Web service search engines	Different combinations of source code metrics are applied
[71]	CS	GA	GA is used for dynamic threshold adaptation	Source code metrics are retrieved from different tools
[54]	AP	-	Perl Script along with the PTIDEJ tool is used to compute metrics from 12 versions of Eclipse and nine versions of ArgoUML	DÉCOR is used to detect antipatterns
[90]	CS	-	The parallel evolutionary algorithm is used	Source code metric thresholds are set using the GA algorithm
[104]	AP	Mathematics	Antipatterns are formally defined	Prolog rules are applied for detection
[48]	AP	CG	A set of metrics and their values are used to generate B-spline	The similarity of the signature is computed to detect antipatterns
[94]	CS	GA	GA is implemented with the help of source code metrics	NSGA-II[125] is used for the correction of defects
[108]	DS	BMS	Initialization, training, memory selection cell	Source code metric selection for detection
[86]	BS		Set of reference code for refactoring	GA is applied
[57]	DS	IP	Input is the name of the method, association type among classes	Similarity scoring and bit-vector algorithm are applied to check smells
[81]	AP	GA	The PE-A algorithm is applied	Different combinations of best threshold source code metrics for detection
[87]	BS	ML	Source code metric results from seven different software repositories	Naïve Bayes, Logistic, IB1, IB-k, VFI, J48, and random forest applied
[96]	CS	GA	Source code metrics	NSGA-II is applied to check, detect, and correct smells from development history
[115]	CS	ML	78 systems are studied, and different source code metrics are analyzed	Combinations of different machine learning algorithms are applied to check the smells for each system under analysis
[78]	AP	ML/GA	Source code metrics are used	An evolutionary algorithm for antipattern detection

Abbreviations: AP, antipatterns; AS, architectural smells; BMS, Biomedical Sciences; CG, conjugate gradient; CS, code smells; DS, design smells; GA, genetic algorithm; IB1, instance-based learning algorithm; IB-k, instance-based k-nearest neighbor; IP, image processing; ML, machine learning; NSGA-II, non-dominated sorting genetic algorithm; PE-A, parallel evolutionary algorithm; SVM, support vector machine; VFI, voting feature intervals.

[†††]http://promise.site.uottawa.ca/SERepository/

[‡‡‡]http://www.cs.waikato.ac.nz/ml/weka/

These techniques used several types of parsers like the Stanford Natural Language Parser to detect parts of speeches. The LAPD (linguistic antipattern detector)[70] tool is proposed for the detection of linguistic antipatterns based on the natural language processing (NLP) parser to detect similarity between class names, variables, and methods by manually implementing smell detection techniques to detect linguistic antipatterns. A recent study reported the effect of linguistic antipatterns on change proneness.[109] Table 12 highlights different conditions used for linguistic source code analysis.

2. Dynamic source code analysis

The smell detection techniques that rely on the dynamic source code analysis techniques mainly analyze the execution states of the systems after their execution under real execution scenarios. Most architectural smell detection techniques use dynamic source code analysis and belong to the paradigm of SOA,[18,21-23,82] relying on DSL that helps generate algorithms along with a service interface, using FraSCAti[11] runtime support for static, dynamic, and lexical analyses. The dynamic source code analysis also used dynamic threshold adaptation instead of fixed thresholds for smell detection.[99]

TABLE 11 Methodological source code analysis

Ref No	Smell Type	Precondition	Post Condition
[74]	AP	Execution and transformation of queries to make an ASG	Create an EMF representation of the ASG for performance comparison
[97]	AP	J-Deodorant,§§§ Check style,¶¶¶ and InCode are used for the detection of smells	Extract Class, Encapsulate Fields, and Move Method refactoring is applied using Jason 1.3.10### and Eclipse Kepler‖‖‖‖
[84]	AS	Source code metrics and multiobjective optimization approach	NSGA-II is implemented to check source code quality before and after refactoring
[79]	AS	Migration strategies are defined for legacy application to SO systems	Refactoring of WSDL document is applied, combining different thresholds of SC metrics
[105]	DS	The CLIO approach is used to detect structural and change coupling	Modularity violation is calculated by comparing structural and change coupling
[95]	CS	The tool is developed that runs in the background to monitor changes	Monitor invokes smell detection tool and warns developers
[85]	BS	Pattern-based definitions are presented based on the symptoms of smells	A survey is conducted to get consensus on revised and improved definitions
[14]	BS	A two-part Web-based questionnaire is developed to get an opinion from developers about smells	Different options are analyzed like an evaluation of developer perception, demographic effects, and experience of developers *w.r.t.* the code smells
[41]	CS	Negative binomial regression is run to check the faults in investigated systems	Different suggestions are passed that helps researchers for refactoring

Abbreviations: AP, antipatterns; AS, architectural smell; ASG, abstract syntax graph; CS, code smell; DS, design smell; EMF, Eclipse Modeling Framework; NSGA-II, nondominated sorting genetic algorithm; SC, source code; SO, software-oriented; WSDL, Web Service Description Language.

§§§https://marketplace.eclipse.org/content/jdeodorant

¶¶¶checkstyle.sourceforge.net/

###https://sourceforge.net/projects/jason/files/jason/version%201.3.10/

‖‖‖‖www.eclipse.org/downloads/packages/release/Kepler/SR

A genetic algorithm along with tuning machine is applied to check the results with inferred settings, default settings, and with a tuning algorithm.[99] We found eight studies that reported dynamic source code analysis for OO. The complete information about the different techniques implemented using dynamic source code analysis is presented in Table 13.

Summary on RQ1: Research on smells analyzes the target systems by applying source code–level metrics that help investigate systems by using lexical properties. Research in the domain of smells also empirically validates findings using statistical measures after investigating the cause-effect relationship with some independent variables, like the numbers of defects. A recent shift toward the use of different algorithms from machine learning as well as artificial intelligence also helps detect design smells and may improve the performance of detection techniques. These algorithms are reported for the detection of smells in both OO and SO paradigms. We have identified a few studies that reported the use of lexical analysis[70,71] and dynamic source code analysis.[22]

To conclude, while the concern is about the classification of state-of-the-art approaches, it is worthy to note that much work has been done on the static and dynamic analyses of OO and SO systems; however, very few contributions have been made by checking the linguistic quality in these systems. In particular, very little research effort has been made to ensure the quality of the identifiers for methods/classes/interfaces, services/operations, and parameters/messages, ie, if the OO and SO systems are of sound quality in terms of linguistic quality. Thus, more attention is required not only on the static and dynamic attributes but also on the linguistic/semantic quality of such systems.

RQ2: How did the state-of-the-art approaches evolve across different paradigms starting from OO to SO?

There are a number of different detection and correction techniques that crossed domains. These approaches mostly focused on source code analysis and evolved from detection to correction in OO and SOA. We extracted data from 78 primary studies and presented the extracted results in an Excel sheet available online.§§§§§ The attributes selected for the extraction for each study are reported in Section 3. The analysis results give us a clear idea about the evolution of the state-of-the-art approaches in OO and SOA. We divided the research methodology of 78 primary studies in five different categories. As we are interested in OO and SOA, we divided the research techniques reported for OO-related primary

TABLE 12 Linguistic source code analysis

Ref	Paradigm	Precondition	Post Condition
[70]	OO	1. Check methods, attributes, and leading comments using the Stanford Natural Language Parser 2. The semantic relation is analyzed using WordNet[****] and implemented as an Eclipse plug-in	Seven open-source system archives are used to check linguistic antipatterns using an LAPD, then online questionnaires are designed to check the developers' perception toward linguistic antipatterns (LAPs)
[109]	OO	Linguistic antipatterns are defined	Case examples are given to analyze the LAPs
[22]	SO	Syntactic and semantic similarities are studies using the Stanford Parser[††††] and a WordNet lexical database	The DOLAR[‡‡‡‡] tool is developed to detect linguistic antipatterns from REST APIs
[71]	OO	Lexical and design smells are detected in 30 releases of three projects: ANT,[§§§§] ArgoUML,[¶¶¶¶]and Apache[####]	Fault proneness is checked for design smells vs lexical smells
[72]	OO	Structural metrics are applied	Principal component analysis along with different statistic measures is used to evaluate the subject system

Abbreviations: API, application programming interface; LAPs, linguistic antipatterns; LAPD, linguistic antipattern detector; OO, object-oriented; REST, Representational State Transfer; SO, software-oriented.

[****]https://wordnet.princeton.edu/

[††††]http://nlp.stanford.edu:8080/parser/

[‡‡‡‡]http://sofa.uqam.ca

[§§§§]http://ant.apache.org/

[¶¶¶¶]https://sourceforge.net/projects/argouml/

[####]http://www.apache.org

TABLE 13 Dynamic source code analysis techniques

Ref No	Smell Type	Precondition	Post Condition
[18]	AS	DSL is used along with algorithm generation to map rules	Static, dynamic, and lexical source code metrics
[21]	AS	DSL is used along with a service interface to invoke the services by using FraSCAti and Apache CXF[‖‖‖‖‖‖‖] runtime support	Wrapping REST API with FraSCAti SCA analysis
[22]	AS	DSL is used along with a service interface to invoke the services by using FraSCAti and Apache CXF runtime support	WordNet, Core NLP is used to analyze lexical properties
[23]	AS	DSL along with FraSCAti runtime support	Source code metrics
[82]	AS	Association rule mining to check the association among execution of services	Source code metrics
[63]	CS	Detect smells on the client side	Check detected smells on the server side
[98]	CS	Mining the source code through SVN	The SrcML[*****] toolkit and a MARKOS[†††††] code analyzer are used
[99]	CS	Tuning machine is applied on an inferring set to check the most appropriated thresholds	Smells are checked and refactored after applying dynamic threshold adaptation

Abbreviations: API, application programming interface; AS, architectural smells; CS, code smells; DSL, Domain Specific Language; NLP, natural language processing; REST, Representational State Transfer; SCA, Service Component Architecture; SVN, Sub Versioning Number.

[‖‖‖‖‖‖‖]https://github.com/apache/cxf

[*****]http://www.srcml.org/about-srcml.html

[†††††]http://markosproject.sourceforge.net/downloads/

studies that also crossed to SOA. The detection and correction approaches used source code metrics or source code analyses as the primary techniques, further combined with other research techniques for the identification of smells. As shown in the following, we use pre- and post-conditions because techniques primarily used behavioral analysis of systems as a prerequisite.[116]

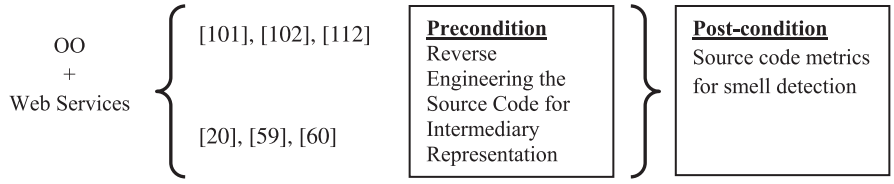

FIGURE 9 Source code metrics in object-oriented (OO) and software-oriented

a. Source code metrics

Source code metrics quantify the application features in the OO design knowledge base. These metrics are selected based on the OO design principles. Moreover, these principles are the core of OO design that further classifies knowledge based on their definitions and different rules used for these definitions. There should be concrete knowledge about the selection of suitable metrics to check if these metrics are a valid indicator of detected smell or not. However, most of the source code metrics are not applied directly to the source code. A literature review indicates that parsing is the activity mostly used to get the intermediary representation of source code, and then, source code metrics are applied to check various quality indicators for the applications.[101,102,112] It is also observed that OO source code metrics are used for SO systems to check the quality of the services by detecting several types of defects in services or in their interfaces.[20,59,60] Figure 9 reports the condition used for source code metric–based evolution.

The primary studies used source code metrics for OOSE to evolve into SO software engineering. All studies used an intermediate representation of source code for the detection of smells across two paradigms.

b. Mining the source code using SVN or CVS

There are a number of studies that report the detection of smells through mining source code using version control systems. Software developers often rely on subversion to keep track of the current and historical versions of files like source code, Web pages, and documentation. Software version history is often used to check the relationship between different quality indicators, with respect to the system performance and solution, ie, refactoring, for a specific problem. These types of studies often use development history of the various releases of the system to check the relationship between two different variables like smells vs maintenance effort[53,91] or smell vs quality of code after refactoring[68] or smell vs change history information about the different versions of the systems[92] by examining the history using the Sub Versioning Number (SVN) or the Concurrent Versions System (CVS) after collecting commits for each change. Approaches also use versioning history with the algorithm called HIST (Historical Information for Smell deTection)[92,98] and function as follows.

1. Versioning systems are used to extract changes in source code.
2. The locations of the changes from versioning systems are given as input.
3. A change history extractor like the SVN or CVS is used to mine the versioning systems, reporting the complete information change. This is performed by comparing the folder and snapshot of change. The SrcML[‡‡‡‡‡] toolkit is used to parse the source code to find cases of change. Then, the code smell detector is applied for smells.

c. Domain Specific Language

Domain analysis is a process that uses specific information required to develop software systems in such a way that is making the desired system reusable for the creation of a new system.[117] The DSLs that are proposed for code, design, and architectural smells are based on the following steps.

1. Key characteristics of smells from the literature are gathered, and rules to discover them within code or design are designed manually.
2. The next step is to check the measurable properties by using low-, high-, and medium-threshold implementation.
3. Lexical properties using WordNet are examined. Moreover, the properties can be combined using set operators like Union (UNION) and Intersection (INTER) to build more complex detection rules.
4. Classification of the key characteristics is used to divide the properties further.
5. Finally, a DSL is proposed to describe smells in terms of their measurable, structural, and lexical properties via a rule card using a set of operators.

[‡‡‡‡‡]http://www.srcml.org/tools/index.html

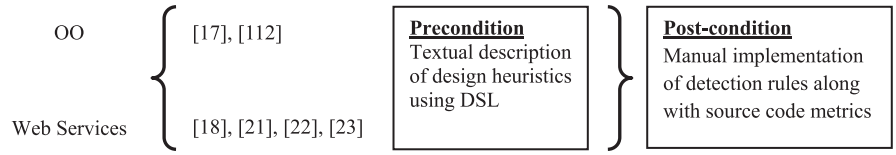

FIGURE 10 Domain Specific Language (DSL) evolves in object-oriented (OO) and software-oriented

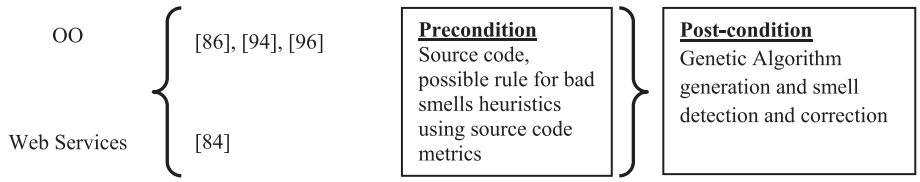

FIGURE 11 Genetic algorithm evolves in object-oriented (OO) and software-oriented

We observed from the SLR of code and design smells that methods based on a DSL mostly rely on the Backus-Naur form for the specification of smells and the boxplot statistical technique for adjusting the threshold values of source code metrics. This technique evolved from OO[17,112] to SO[21-23,40] as shown in Figure 10.

d. Genetic algorithm

Smell detection and correction approaches also use genetic algorithms to improve system quality by detecting smells as well as suggesting refactoring opportunities to correct them.[84,86,94,96] The main benefits of using such approaches are as follows.

1. Genetic algorithms only require defect examples and not different defect types.
2. It is not required to write the detection and correction rules.
3. Metrics with related threshold values are not required, which may cause problems in the case of different thresholds reported in the literature.
4. The effort required to perform refactoring is also considered for the detection and correction of smells.
5. Already discovered smell results are used as learning examples.
6. Derived detection rules are used to select the best refactoring solutions from a list.
7. Refactoring solutions provide suggestions of the best alternative on the set of defects.
8. Mutation and crossover operators are applied with given probabilities, the resultant is evaluated using a fitness function, and the process is repeated until the stop criteria are met.
9. The algorithm, called NSGA-II, having the precision for the detection of smells and correction of about 87% both for the OO[86,94,96] as well as Web services.[84] Figure 11 shows the relevant studies based on the genetic algorithm.

e. Parallel evolutionary algorithm

The evolutionary algorithms (EAs) and the particle swarm optimization process are used to reduce the computational complexity of the search process. The algorithm is based on the following main features.

1. Parallelization allows speeding up the search process.
2. Exchanging information between different search methods.
3. Using several types of EAs reduces the sensitivity of different parameters used for the detection of smells.
4. Iterations are independent of the problem.
5. The parallelization process uses a single solution from the search space. The solution uses a set of detection rules that help detect a specific type of code smell.
6. Parallelization is used to generate a detection rule, and then, the genetic algorithm is used for the detection. Finally, the set of the best candidates as a solution is selected.
7. The parallel evolutionary algorithm (PE-A) was reported primarily for OO[90] and then adapted for Web service antipattern detection.[81] The criteria for implementing the PE-A are presented in Figure 12.

Summary on RQ2: After examining the studies from the last 17 years, we identified five different approaches that evolved in OO and SO. These approaches are now quite matured with the ability to provide highly accurate detection results for both OO and SO paradigms. We investigated their steps involved in the identification or correction of smells.

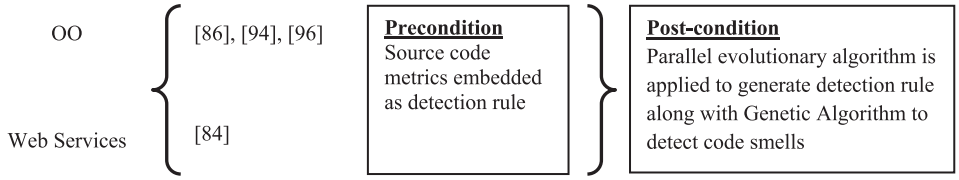

FIGURE 12 The parallel evolutionary algorithm evolves in object-oriented (OO) and software-oriented

However, in the literature, the technique related to mining the source code using versioning systems is still not applied for Web services. We did not find any study that discusses the effect of smells across the different versions of service interface APIs. Source code metrics are the only technique used repeatedly; however, much work is still needed to be done for Web services by introducing some novel metrics, which would help investigate the QoS issues for Web services.

To conclude, while the concern is the evolution of approaches from OO to SO systems, we found that the use of optimization algorithms is common across the paradigms. In addition, for the OO paradigms, it was largely feasible to analyze source code metrics than the SO systems, which is due to the availability of the source code of OO systems with ease as compared to SO systems. For the same reason, mining the SVN/CVS is a lot more possible in the OO paradigm. All these analyses in the SO paradigm are possible only at the interface level. Practitioners should take initiatives to open their source code to academia.

RQ3: What are the smells that are studied for a specific paradigm?

a. Smells reported in OO

A key argument for investigating smells is that certain smells are emphasized more in the literature than others. Moreover, there are different terms provided in the literature for smells like code smells, design smells, architectural smells, and lexical smells that may confuse researchers on which category a smell belongs. The term "code smell" was first introduced by Fowler et al[8] with corresponding refactoring opportunities. Later, Brown et al[36] introduced the term "antipatterns" and divided them into three categories: software development, architectural, and project management antipatterns. Therefore, the smells were later reported in the literature as design, architectural, and code smells and commonly referred to as *smells*.

We search the relevant literature on bad software smells and identify various smells that are reported as code smells, design smells, architectural smells, and antipatterns. Most of the relevant studies reported and analyzed *Feature Envy* as code- and design-level smells. To the best of our knowledge, we did not find this smell reported as architectural smells. Table 14 describes the number of reported smells and their categorization as bad, code, design, and architectural smells in the literature. In Table 14, the Frequency column shows that *Feature Envy* gains utmost attention from researchers. In contrast, much study is still required to be done for the detection of smells like the *yo-yo problem*, *unnamed coupling*, *extensive coupling*, and so on, which gained less attention so far in the software engineering (SE) research community.

If we consider the category of smells defined by Fowler et al[8] and the antipatterns as defined by Brown et al,[36] then the total number of smells comes to 46. If we examine the literature review from the year 2000 until 2017, we find, in total, 22 smells among the ones defined by Fowler et al.[8] In addition, we were unable to find few smells as reported by Brown et al.[36] Table 14 lists all the smells reported as code, design, and architectural smells. However, no studies were found exploring the smells like *Dead End*, *Reinvent the Wheel*, *Primitive Obsession*, *Inappropriate Intimacy*, *Golden Hammer*, and *Incomplete Library Class*. Moreover, there is no template described for code smells as reported for antipatterns in the literature.[36] The correction of code smells might improve the understandability and maintainability of the source code. However, one can remove the antipatterns at the design level, which may lessen the number of smells at the code level. Therefore, to improve the system quality, one should remove both antipatterns and code-level smells, which exist at the design and code levels, respectively.

Mäntylä et al[34] and Wake[33] proposed a classification for smells. Moha et al[17] divided the code smells and antipatterns as inter- and intraclass smells based on structural, lexical, and measurable properties. Another classification of smells was reported in the literature that divides the code smell detection approaches into seven broad categories.[90] However, this categorization[90] is based on the approaches used to handle smells and not based on the properties of the smells. In this paper, we categorize the smells reported in the literature based on the properties associated with each smell and follow the criteria defined by Mäntylä et al.[34] We collect the relevant definition and properties of each smell from the literature and then divide those smells in different classifications like code smells, design smells, antipatterns, and architectural smells. Moreover, Mäntylä et al[34] focused only on the code smells defined by Fowler et al,[8] and not on the antipattern

TABLE 14 Smells reported in the literature from the object-oriented paradigm

Sl No	Smell Name	Type of Smell	Ref No	Frequency
1	Feature Envy	BS, CS, DS	17,44,45,52-55,58,61,65,68,69,72,76,86,87,90,92,93,97,98,100,101,106,109	25
4	God Class	AS, BS, CS, DS	17,45,52-54,58,61,65,69,75,76,88,93,100-102,104,106	19
2	Blob	BS, CS, DS, AS	16,29,35,43,48,67,68,71,73,84,90,92,94,96,98,103,108,110,112	18
3	Data Class	CS, DS	52,53,58,61,65,69,76,88-90,93,95,96,101,102,104,106,109	18
5	Long Parameter List	AP, BS, CS, DS	14,16,29,35,45,67,68,71,73,76,86,87,89,90,95,103,109,112	17
6	Spaghetti Code	AP, AS, CS, DS	16,29,35,43,67,68,71,84,90,94,96,103,108,110,112	15
7	Shotgun Surgery	CS, DS	17,53,61,67,71,76,90,92,93,98,100,101,103,106,109	15
8	Duplicated Code	BS, CS, DS	14,45,53,54,57,58,61,67,71,76,88,93,95,103,109	15
9	Large Class	AP, BS, CS, DS	14,16,29,35,44,45,57,76,86,89,95,103,109,110,112	15
11	Long Method	AP, BS, CS, DS	16,29,35,45,68,73,76,87,89,95,103,108-110,112	15
10	Speculative Generality	AP, BS, CS, DS	16,29,35,43,61,67,68,84,85,96,108,109,112	13
12	Lazy Class	AP, BS, CS, DS	16,29,35,44,68,72,86,87,90,109,110,112	12
13	Refused Parent Bequest	CS, DS	17,53,54,61,68,73,93,100,101,106,109,112	11
14	Functional Decomposition	AS, CS, DS	29,43,67,71,84,90,94,96,103,108,112	11
15	Message Chain	BS, CS, DS	41,67-69,71,73,85,87,103,109	9
16	Data Clump	BS, CS, DS	41,53,69,85,93,97,109,110	8
17	Swiss Army Knife	AP, CS, DS	16,29,35,67,71,103,112	7
18	Divergent Change	CS, C&D, DS	67,71,92,98,103,109	6
19	Switch Statement	BS, CS	85,87,89,95,99,109	6
20	Comment	CS, C&D, DS	67,71,103,109,110	5
21	Parallel Inheritance	CS	89,92,98,109	4
22	Misplaced Class	CS, DS	53,93,101,106	4
23	Class Data Should be Private	AP, CS, DS	16,29,35,68,73	4
24	Poltergeist	AS, CS, DS	57,75,104,110	4
25	God Method	CS	53,54,93	3
26	Anti-Singleton	AP, DS	16,29,35,73	3
27	Complex Class	AP, DS	16,29,35,73	3
28	Middle Man	BS, CS	41,87,109,110	4
29	Brain Class	CS, DS	52,61,65	3
30	Public Fields	CS	95,97	2
31	Schizophrenic Class	CS	61,69	2
32	God Package	DS	101,106	2
33	Wide Subsystem Interface	DS	101,106	2
34	Decorator	BS, DS	45,106	2
35	Global Variables	C&D, DS	67,71	2
36	No Polymorphism	C&D, DS	67,71	2
37	Procedure Class	C&D, DS	67,71	2
38	Brain Method	CS	61	1
39	Common Methods in Sibling Class	CS	95	1
40	Extensive Coupling	CS	61	1
41	External Duplication	CS	69	1
42	Idle Cut Point	CS	72	1
43	Intensive Coupling	CS	61	1
44	Redundant Cut Point	CS	72	1
45	Traditional Breaker	CS	61	1
46	Adapter	DS	93	1
47	Code Clone	DS	105	1
48	Cyclic Inheritance	DS	57	1
49	Cyclic Dependency	DS	105	1
50	Delegated	DS	44	1

(Continues)

TABLE 14 (Continued)

51	Interface Bloat	DS	[104]	1
52	Missing Association Class	DS	[57]	1
53	Observer	DS	[104]	1
54	Poor Inheritance Hierarchy	DS	[105]	1
55	Unnamed Coupling	DS	[105]	1
56	Yo-yo Problem	DS	[104]	1

Abbreviations: AP, antipatterns; AS, architectural smell; C&D, Code and Design; CS, code smell; DS, design smell.

properties as defined by Brown et al.[36] Therefore, we also use the classification of design smells and antipatterns reported in the work of Ganea et al.[58] The partition of smells according to the classification reported in the literature is discussed in the following.

***The Bloater*:** Bloater describes something in the source code that has grown rapidly and, thus, not possible to handle effectively. The smells in this category are *Blob*, *God Class*, *God Method*, *Data Clump*, *Long Method*, *Large Class*, *Primitive Obsession*, *Long Parameter List*, *Complex Class*, and *God Package*. It is very difficult to modify or maintain large codes that further transformed into the *Long Method*, *Large Class*, or *God Class*. This is also true for the *Long Parameter List* and *Data Clump* as they are often found with a long list of parameters. The *God Package* smell is only reported in two studies,[101,106] whereas the *God Method* is found in three studies.[53,54,93] The *Data Clump* smell is also reported in several studies.[53,69,85,93,97,109,110] The smells that are reported by the maximum number of studies include *Blob* and *God Class*, 18 and 19 times, respectively. The complete list of references that studied the Bloater group of smells is presented in Table 14.

***Object-Oriented Abusers*:** The smells in the OO abuser category include *Switch Statements*, *Temporary Field*, *Refused Bequest*, *Alternative Classes with Different Interfaces*, *Parallel Inheritance Hierarchies and Poor Inheritance Hierarchies*, *Class Data Should be Private*, *Global Variables*, *No Polymorphism*, *Procedural Class*, *Public Fields*, *Missing Association*, *Cyclic Inheritance*, *Idle Cut Point*, *Redundant Cut Point*, *Traditional Breaker*, *Adapter*, *Code Clone*, *External Duplication*, and *Cyclic Dependency*. This categorization is often related to the smells where the solution does not fully utilize all the benefits of OO design. The *Refused Bequest* smell is based on this definition because it violates the rule of inheritance design, which is one of the fundamental principles of OO design. Moreover, the *Alternative Classes with Different Interfaces* smell also suffers from the common interface for closely related classes. This shows an example of misusing the OO principles. Similarly, the *Violation of Polymorphism*, *Use of Public Data Members*, and *Class with Missing and No Associations* also fall in the abusers category. However, these smells are not reported in Fowler's catalog of smells. Several studies under this category of smells ignored *Parent Bequest*,[17,53,54,61,68,93,100,101,106,109,112] whereas the *No Polymorphism* and *Procedural Class* smells were studied in only two studies.[67,71] Similarly, the *Cyclic Inheritance* smell is reported in only one primary study that clearly shows a research gap for this smell.

***The Encapsulators*:** The encapsulators often deal with the communication mechanism or encapsulation. The smells in this category are *Message Chains*, *Common Method in Sibling Class*, and *Poltergeist*. These types of smells are often interdependent, where the removal of one smell may cause the introduction of another smell when it is removed. The potential solution for this category of smells is to restructure the class hierarchy by moving a method to another class. However, care must be taken such that the move does not introduce the *Common Method in Sibling Class* smell. The encapsulator smells are mostly based on how the object, data, and operations are accessed. The studies that reported the smells in the category of encapsulators mostly consider the *Message Chain* smell,[67-69,71,85,87,103,109] the *Middle Man* smell is reported three times in the literature,[87,109,110] and the *Common Method in Sibling Class* is reported only once.[95] More studies are required in this category.

***The Coupler*:** These types of smells are strongly related to some properties of the class that may hinder the reusability of the software. The *Schizophrenic Class*, *Message Chain*, *Middle Man*, *Incomplete Library Class Feature Envy*, *Inappropriate Intimacy*, *Intensive Coupling*, *Extensive Coupling*, and *Unnamed Coupling* smells belong to this category. These smells are largely related to the property of coupling and often misuse or overuse the coupling. The research on smells reported *Feature Envy* as a maximum number of smells detected, corrected, and/or considered for maintenance (ie, 25 times) as compared to *Intensive/Extensive Coupling*.[61]

***The Design Rule Abusers*:** These types of smells violate the rules to design the classes or overall programs. These types of smells are erroneously introduced by the programmers in a way that they might consider them as patterns (ie, good practice), but later, they turn into antipatterns (ie, poor practice). Design rule abusers can be further divided as the use of wrong programming approaches like *Boat Anchor*, *Lava Flow*, or *Wrong Methodology* by using *Copy-Paste*

TABLE 15 Smells that are reported repeatedly in the services literature

Sl No	Smell Name	Reference No	Frequency
1	God Object Web Service	17,18,23,62,81,82,100,101,106	9
2	Low-Cohesive Operation	18-20,59,60,81,83	7
3	Ambiguous Names	19,20,59,60,80,83	6
4	Chatty Service	18,23,62,80-82	6
5	Data Web Service	18,23,62,80-82	6
6	Duplicated Web Service	18,23,62,81,82	5
7	Enclosed Data Model	19,20,59,60,83	5
8	Redundant Data Model	19,20,59,60,83	5
9	Whatever Types	19,20,59,60,83	5
10	Empty Messages	20,59,60,83	4
11	Bloated Service	23,62,82	3
12	Bottleneck Service	23,62,82	3
13	Nobody Home	23,62,82	3
14	Sand Pile	23,62,82	3
15	Service Chain	23,62,82	3
16	Stove Pipe	23,62,82	3
17	The Knot	23,62,82	3
18	CRUDy Interface	18,81	2
19	Fine-Grained Web Service	18,81	2

Programming, *Golden Hammer*, *Defactoring*, *Spaghetti Code*, *Anti-Singleton*, *Misplaced Class*, *Wide Subsys Interface*, and the *Yo-yo Problem*. The trend toward the smells as defined by Brown et al[36] is not observed much as we found that only the *Spaghetti Code* is reported in a substantial number of studies[16,29,35,43,67,68,71,84,90,94,96,103,108,110,112] as compared to the *Yo-yo Problem*.[104] However, we are unable to find any relevant study that reports *Boat Anchor* and *Lava Flow* smells.

The Lexical Abuser: Fowler et al[8] defined code smells in code comments that are smells when the comments do not contain information corresponding to the source code and its behavior, which Moha et al[17] later reported as lexical smells if they do not match with the internal code behavior. Recently, a study reports the catalog of lexical smells based on the internal code structure.[70] This catalog considers the method, class naming conventions, as well as method return types to define lexical smells.[70] The complete list of these smells is available online.[§§§§§]

Table 14 reports the name of the smells along with their category as reported in the corresponding research article. The studies reported in this SLR include 58 smells. However, we do not add *Code Clone* or *Copy Code* to this category as it is not included in our review protocol. In Table 14, we report 56 smells along with their frequency.

In Table 14, we can observe that the smells reported by Brown et al[36] as antipatterns still did not receive significant attention from the researchers and were not studied thoroughly in the literature. Moreover, a number of new smells are introduced in the literature like *Code Clone*, *Unnamed Coupling*, and *God Package* in addition to the smells defined by Fowler et al.[8]

b. Smells reported in SO systems

1. *Smells that received more attention:* SOA is a promising architectural style that facilitates the development of low-cost, reliable, and flexible services usable or accessible over the Internet.[47] This architectural style can be implemented using technologies like REST, SOAP, SCA, RPC, and J2EE. The detection of smells in the services is a quite new but challenging area that is receiving increased attention in the SE research community. There are a number of smells that are detected in different SOA technologies like SOAP,[18,81] REST,[21,22] and SCA.[23] Moreover, we also found several studies that reported smells for the WSDL file that is the core specification of the SOAP Web services. Table 15 highlights the frequency for a specific smell reported in the SO systems literature. It is also to be noted that there are more than ten smells reported only once in the services domain like *Breaking Self-Descriptiveness*, *Content Negotiation*, and *Ignoring MIME Type*.[21] The REST architectural style is the area where we identify the gap for the detection of various service antipatterns in SO systems.

2. *Smells that received less attention:* We also found several smells from different SOA technologies that are reported only once in the literature. These smells are detected based on static and dynamic source code analyses by computing different

[§§§§§] http://research.ciitlahore.edu.pk/Groups/SERC/SOA.aspx

TABLE 16 Smells reported for the first time in the services literature

Ref	BSD	CN	CvCLRN	DNU	DOR	DSSS	FH	HvNHN	IC	IMT	ISC	ILC	LDFINT	LFDS	LFEBAOD
19											√	√			
79	√			√	√	√							√	√	√
80															
21		√					√	√		√	√				
22			√					√							

Abbreviations: BSD, Breaking Self-Descriptiveness; CN, Content Negotiation; CvCLRN, Contextualized vs Contextless Resource Name; DNU, Detected Not Used; DOR, Detect Operation that Receive; DSSS, Detect Semantically Similar Services and Operations; FH, Forgetting Hypermedia; HvNHN, Hierarchal vs Non-Hierarchical Node; IC, Ignoring Cache; ILC, Inappropriate or Lacking Comments; IMT, Ignoring MIME Type; ISC, Ignoring Self-Descriptiveness; LDFINT, Look for Data Types For Inconsistent Names and Types; LFDS, Look For Data types that subsumes other Data types; LFEBAOD, Look For Error Information Being Exchanged as Output Data.

TABLE 17 Smells reported for the first time in the services literature

Ref	LFRDD	LSDASI	LFCWSDL	MC	MS	NS	RPT	RC	SINS	SPN	TVA	TTG	TTP	UCFISM	VCU
19							√							√	
79	√	√	√						√						
80					√	√									
21				√				√				√	√		
22										√	√				√

Abbreviations: MC, Misusing Cookies; LFRDD, Look for Redundant Data type Definition; LSDASI, Look for Shard dependencies among Service Implementation; LFCWSDL, Look for Comments in WSDL; MS, Multi Service; NS, Nano Service; RPT, Redundant Port Type; RC, Response Cache; SINS, Share Inappropriate Names for Service elements; SPN, Singularized vs Pluralized Nodes; TTG, Tunneling Through Get; TTP, Tunneling Through Post; TVA, Tidy vs Amorphous URI; UCFISM, Undercover Fault Information within Standard Message; VCU, Verb less vs Cruddy URI.

properties and detected instances of each antipattern in the related services. Most of these antipatterns are related to REST services and reported after the year 2014. Tables 16 and 17 report the findings of those smells that received less attention from SOA technologies.

Summary on RQ3: This research question about the smells (reported for a specific domain) is useful for the future research directions. Our findings suggest that most of the study is performed to investigate *God Class*, *Feature Envy*, *Data Class*, and *Blob*. In total, researchers discussed 56 smells. Researchers used static or dynamic source code analysis for the identification of smells, but attention must be given to those smells that still took less attention like *Unnamed Coupling*, *Poor Inheritance Hierarchy*, and the *Yo-yo Problem*. Similarly, there is a need for an investigation on smells that received less attention in the SO paradigm. Most of the smells belong to REST services and still did not receive much attention, as they need dynamic analysis of the service interface. The techniques used in the OO for static source code analysis are not applicable to REST services because the method for using a class in OO and the method for consuming services in the SO paradigm are not conceptually similar. Most of the attention is given to smell detection for the WSDL interface for Web services. The SOAP services are also analyzed using dynamic properties like availability, throughput, and response time.

To conclude, this research question ideally focuses on the wide range of smells, studied in the literature in OO and SO systems. Overall, the OO domains include a large set of smells; however, not all of them have been analyzed for their automatic detection. In particular, a few smells like *Feature Envy*, *God Class*, *Blob*, *Data Class*, *Long Parameter List*, *Spaghetti Code*, *Shotgun Surgery*, *Duplicated Code*, *Large Class*, and *Long Method* received more attention than others. However, there are still 45 other smells as reported in Table 14 that did not receive much attention from the researchers. As compared to OO smells, the studies' list of smells in the SO literature is very small, ie, only 19 smells. As the software concept has already diverted into the Web era that is service based, researchers should put more effort on SO smells to ensure high-quality service-based systems.

RQ4: What is the correlation between smells across the paradigms?

The studies in the literature considered different smells that are evolved in OO and other domains of software engineering. These smells are also studied and analyzed in the paradigm of SOA. This research question will cover the evolution history of source code measures used for the identification of smells in OO and SO and the smells evolved in OO and SOA. The OO uses classes as compared to services that use interfaces. However, services used by the clients are also embedded using classes and methods; thus, there is a need to study the evolution of smells in OO and SO.

Moreover, SO systems can be implemented by using various technologies, and there are different tools, eg, Java2WSDL, used to generate a representation of services from OO code.[19,20,79] SOAP services can be implemented using the code-first or contract-first approach,[20,79] and OO source code metrics can be reused to detect antipatterns for the code-first approach. If OO code is smelly, then the interface generated using an automated tool will be also smelly.[20] Therefore, source code metrics are highly used by researchers to extract smells from Web services.

Intermediate representations are useful for the identification of smells both for SOA and OO and enable researchers to extract properties of smells. SOA relies on services that generally gather and implement low-cohesive operations in comparison to OO where cohesion must be high for a class or sets of methods. LCOM (Lack of COhesion among Method) may be used for both OO and SOA, but the threshold values may vary as reported in previous studies.[18,81]

Evolution of smells across the paradigms: Out of the 78 most relevant studies investigated in this paper, we identified four studies that belong to the OO paradigm and reported code comments as the smell that evolves to SO and reported by two SO primary studies.[19,79] It is worth mentioning that source code measures used for the identification of smells for SO systems are also reported for OOSE. The primary studies that reported the smell *Comments* across different paradigms are shown in Table 18. The detailed descriptions of these smells with the approaches used for their detection are reported online.[§§§§§] Code comments are the property used by the developers in both OO and SOA. However, the property used for the identification of *COMMENT* may have different thresholds.

TABLE 18 Smells evolved in object-oriented (OO) and software-oriented

Sl No	Smell Name	OO Ref No	Services Ref No
1	Comments	67,71,103,109,110	19,79

TABLE 19 Source code metrics used for the detection of service smells

Sl No	Smell Name	Ref No	Frequency	Metrics Used
1	God Object Web Service	17,18,23,81,100,101,62,82,106	9	COH, NOD, RT, Av
2	Low-Cohesive Operation	18-20,59,60,81,83	7	NOD, ANIO, WMC, LCOM3
3	Ambiguous Names	19,20,59,78,80,83	6	ALS, RGTS, NVMS, NVOS
4	Chatty Service	18,23,62,78,80-82	6	COH, ANAO, NOD, RT, Av
5	Data Web Service	18,23,62,78,80-82	6	COH, ANPT, ANAO
6	Duplicated Web Service	18,23,62,81,82	5	ARIM, ANIO
7	Enclosed Data Model	19,20,59,60,83	5	CBO
8	Redundant Data Model	19,20,59,60,83	5	WMC
9	Whatever Types	19,20,59,60,83	5	ATC
10	Empty Messages	20,59,60,83	4	WMC
11	Bloated Service	23,62,82	3	NOI, NMD, TNP, COH
12	Bottleneck Service	23,62,82	3	CPL, Av, RT
13	Nobody Home	23,62,82	3	NIR, NMI
14	Sand Pile	23,62,82	3	NIR, NMI
15	Service Chain	23,62,82	3	NTMI, Av
16	Stove Pipe	23,62,82	3	NUM, NMD, ANIM
17	The Knot	23,62,82	3	CPL, COH, Av, RT
18	CRUDy Interface	18,81	2	NCO, ANAO, NOD, RT, Av
19	Fine-Grained Web Service	18,81	2	NOD, CPL, COH
20	Chatty Service	78	2	COH, CBO

Abbreviations: ALS, Average Length of Signature; ANAO, Average Number of Accessor Operations; ANIO, Average Number of Identical Operations; ANIM, Average Number of Identical Messages; ANP, Average Number of Parameters in Operations; ANPT, Average Number of Primitive Types; ARIM, Average Ratio of Identical Message; ATC, Abstract Type Count; Av, Availability; CBO, Coupling Between Objects; COH, Cohesion; CPL, Coupling; LCOM3, Lack of Cohesion Method 3; NCO, Number of CRUDy Operations; NIR, Number of Incoming References; NMD, Number of Messages Declared; NMI, Number of Method Invocation; NOD, Number Of Operations Declared; NOI, Number Of Identical Operations; NTMI, Number of Transitive Method Invocation; NUM, Number of Utility Methods; NVMS, Number of Verbs in Method Signature; NVOS, Number of Verbs in Operation Signature; RGTS, Ratio of General Terms in Signature; RT, Response Time; WMC, Weighted Method Complexity.

Source code metrics used for smell detection in OO and SO: SOA is an emerging and a new challenging area that is gaining increased research attention. The smells reported for the services paradigm initiated to be introduced after the year 2010. It is worth mentioning that the smells reported for this paradigm also rely on source code metrics that are primarily used by OO smells. However, it is noted that smells from SO systems are based on static, dynamic, and linguistic analysis of source code, documentation, and service interfaces, eg, WSDL files. Table 19 shows the source code metrics used for the detection of smells in services in the literature.

The metrics used for the identification of 20 smells reported for SOA used source code metrics that have been previously reported for OO. For example, identification of the *God Object Web Service* used cohesion metrics (eg, LCOM3) and operation identification metrics (eg, NOD) that are also used for the identification of smells in OO,[26] but the threshold values may vary for SOA in comparison to OO. The LCOM3 value must be high for OO,[7,26] but it should be low for SOA.[118,119]

Summary on RQ4: In answering RQ4 on the smells and source code measures that evolved in OO and SO, we identified many source code metrics that were used for the identification of OO smells but later reused for the identification of smells related to SO systems. More smells could be investigated and detected after examining the complete list of information already available in blogs, websites, and books related to SO systems using the source code–level metrics that belong to the widely known CKMJ[¶¶¶¶¶] suite to discover and even define new antipatterns in the SO paradigm. Some studies also used source code metrics as a prerequisite for smell identification,[35,45,56,65] and then, different statistical measures are applied to investigate the effect of smells on subject systems. Therefore, we also used these source code measures to investigate the effect of smells on different versions of SO systems, defect prediction, and maintenance. We are more interested in investigating the approach used for the smell identification rather than the characteristics of the smells because OO and SOA smells are not directly comparable. For example, for the *Multi Service* antipattern in SO systems and the *God Class* antipattern in OO systems, their presence is at different granularity levels. Thus, the detection methodology may be the same, but conditions used to implement these approaches vary like threshold values for measures, implementation of metrics at the code level or at the interface level, and the presentation of intermediary source code representation.[19,20]

To conclude, while studying the correlation between smells across the paradigms, our SLR found that researchers tried to map service interfaces in SO with the classes/interfaces in OO and operations in SO with the methods in OO, for example, the *God Object Web Service* in SO vs the *God Class* in OO and *Data Web Service* in SO vs the *Data Class* in OO. Although this mapping may be useful for reusing some of the OO metrics and smells in SO paradigms, such mapping does not always hold. In particular, consider the fact that analysis in SO cannot be performed at the fine-grained statement level like in OO systems. This limitation by large hinders the qualitative and quantitative analyses of service artifacts in the SO paradigm.

RQ5: What are the trends in research on smells from January 2000 to December 2017?

Our study collected relevant research studies based on the Kitchenham guidelines[50] as described in Section 3. We have observed a clear research trend for the detection and correction techniques of smells. The research started in the year 2000 on the detection of smells in OO systems and slowly moving toward the correction of detected and reported smells.

On the basis of our literature review, we can group research trends on smells into three principal categories, ie, detection, correction, and the impact of smells. Figure 13 reports the main research trends in the domain of smells along with their subdomains. Figure 13 provides general trends associated with smells. These research trends are still not reported for SO smells. Figure 13 also reports a new trend toward the impact of smells on software system evolution also reported for SO smells.[78] Table 20 summarizes the trend of the research on the smells. Major studies focused on the detection by using source code metrics with the help of different algorithms. Most of the primary studies that use the genetic algorithm and development history for the detection of smells also reply mainly on source code metrics.

Summary on RQ5: In answering RQ5 on the research trends in the domain of smells, we found a corpus of 34 research studies on the detection of smells. We observed (1) the studies on the relationship between smells and change proneness and maintenance and (2) the studies on intersmell relationships, both started after the year 2010. A new research gap was filled by introducing smells in SOAP and REST services. Moreover, there is no study published on the correction of REST antipatterns. The NLP techniques are also used for the detection and correction of smells in OO[68,102] to the detection of REST linguistic antipatterns.[22] The intersmell relationships and the impact of smells on the maintenance of services in the SO paradigm have not yet been studied.

To conclude, while investigating the research trends on bad smells, we found that significant effort has been made in detecting bad smells; however, very little research has been conducted in refactoring those smells or in recommending solutions to the detected smells. One reason could be while detecting smells in both OO and SO paradigms, perhaps, it is

[¶¶¶¶¶] https://www.dmst.aueb.gr/dds/sw/ckjm/doc/indexw.html

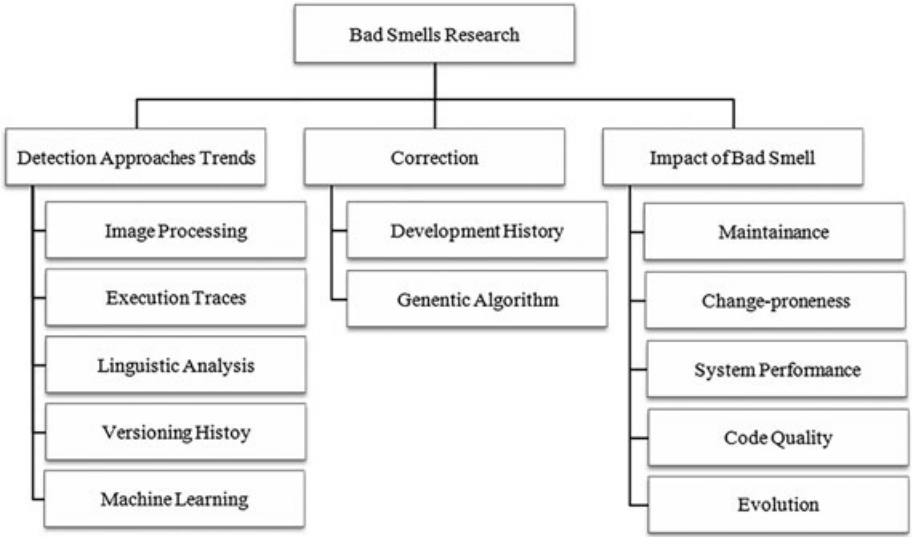

FIGURE 13 General classification of research trend

TABLE 20 Trends in research from the years 2000 to 2017

Trend Name	Frequency	Year	Ref No
Smell detection	34	2001-2015	16,27,29,48,54,58,62-65,70,72,74,82,83,87-90,93-96,101,103-110
Smells vs maintenance	2	2013	53,91
Smell detection using machine learning	2	2011-2012	15,57
Detection of WSDL antipatterns	2	2011-2015	59,60,79-81
Smells and code quality	2	2012, 2015	67,71
Detection of antipatterns from services	2	2013-2014	18,20
Linguistic antipatterns	2	2015-2017	71,109
Smell impact on software changeability	1	2009	43
Smell detection using image processing	1	2010	56
Ontological relationship of smells	1	2010	110
Antipattern detection by mining execution traces	1	2013	17
Antipatterns and fault proneness	1	2013	35
Impact of smell on system quality	1	2013	93
Change proneness of service pattern and antipatterns	1	2014	23
Smell correction using the development history	1	2015	14
Detection of REST linguistic antipatterns	1	2015	22
WSDL refactoring	1	2015	19
Performance comparison of smell detection techniques	1	2015	100
Source code versioning for code smell detection	1	2015	92
Evolution of code smells	1	2017	78

Abbreviations: REST, Representational State Transfer; WSDL, Web Service Description Language.

a little too late to detect and even correct those smells after a while at the time when the systems are already in operation. Therefore, an IDE-based (Integrated Development Environment) smell detection and correction framework is required so that developers can ensure good-quality software artifacts since the early development stage.

5 | DISCUSSION AND OPEN ISSUES

In total, we reviewed 78 highly relevant studies out of the collection of 506 studies published between January 2000 and December 2017. Previous surveys focused either on only code clones[123,124] (ie, a type of smell) or smells or refactoring

activities in the OO paradigm. After we collected the most relevant studies related to the evolution of smells in OO and SO, we analyzed techniques applied to detect smells. Most of the studies[18,19,60,79-82] reporting the detection of smells for SO systems are at the interface level, unlike in the OO paradigm where analyses are mainly done at the source code level. Moreover, while few studies reported the detection of smells in SO systems at the architectural level,[21,81,112] the correction of these architectural smells is not yet studied and requires further research.

Fowler et al[8] identified 22 code smells and suggested their refactoring opportunities. Since then, the research on smells had been gaining increased attention, and different research studies were published from the year 2000 until 2017. However, the term "code smells" later reported in the literature included various forms of code, design, and architectural smells. The development and architectural antipatterns reported in the work of Brown et al[36] also focus on Fowler code smells. Antipatterns like the *Blob* and *Functional Decomposition* are reported for the first time as antipatterns by Brown et al,[36] but later, they are also described as code and design smells. Table 14 shows the *Blob* as code, design, and architectural smells and antipatterns. Therefore, the SE research community does not have a clear consensus on smells. Moreover, the detection approaches for these smells have also varied results due to the availability of multiple source code metrics for the same systems. For example, one can measure the cohesion by using LCOM, LCOM1, LCOM2, or LCOM3.[120]

Moreover, the research community does not always validate their results using precision or recall. We found 15 studies[14,16,26,35,44,45,53,61,68,69,75,76,85,106,110] where the precision and recall to measure the accuracy could not be applied because these studies focused on cause-effect relationships, ie, examples include the impact of smells on the developers' maintenance effort, change proneness, and so on. One study measured the performance of their research model by using correlation analysis.[63] However, in total, 13 studies did not validate their results at all.

Apache is an open-source ecosystem, and Xerces is an Apache project that has various libraries for source code parsing. More than 90% of the plug-ins reported for the validation of the detection techniques are for Eclipse. We provided our complete findings and detailed discussion on those tools online.§§§§§

While moving toward smell detection in SO systems, most of the studies[20,59,60] are dependent on the techniques to develop the SO systems as the code-first or contract-first approaches. To the best of our knowledge, only one research paper studied smells in RESTful APIs. However, the APIs used for the study are not open-source, and the definitions of smells are mostly focused on QoS issues. Future research should include the impact of smells on different versions of APIs to investigate the evolving smells.

The research on smells is quite mature in terms of a number of studies as well as different state-of-the-art techniques for the OO paradigm. However, the detection of smells in SO systems was introduced only after the year 2010. It is difficult to validate the detection or correction techniques of smells for SO systems because open-source SO systems are not greatly available, unlike OO systems. In addition, there is no technical support available from the industry to validate the proposed techniques and their results. This research on the impact of smells will also help in confirming the benefits of design patterns in SO systems as opposed to the antipatterns. The area is still open for researchers to study different state-of-the-art smell detection and correction techniques for the improvement of the quality of REST and SOAP systems.

A large number of the studies reported negative impacts of smells on software systems. Few studies have investigated the occurrences of smells across different versions of software systems.[53,68,91,92] More investigation should be carried out to measure the cause-effect relationship of design patterns vs antipatterns (ie, smells) as well as their impact on the system's overall performance over the long run. The studies focus more on the detection (ie, 39 studies in total) as compared to the maintenance effort. Some of the studies also use version control systems for investigating smells.[92] We did not find any study investigating more than ten smells by using SVN or CVS.[54,88,96,98] Recently, multiple studies have concluded that smells have an adverse impact on software quality.[58,61,68]

Validation criteria for studies

Several studies validated their proposed detection techniques.[30,81,82] Mostly, the journal articles used specific validation techniques to validate their results about specific approaches. Several studies (see, eg, the work of Salehie et al[103]) used accuracy measures, like precision and recall, to validate their techniques. We also found 20 studies that measured cause-effect relationships, like the impacts of smells on systems' maintenance[53,93,94] and code quality.[58,68] Moreover, the most widely used technique for the assessment of the impacts of smells is based on correlation analysis (eg, five studies) as compared to the Wilcoxon signed-rank test and regression analysis (eg, three studies each). The reference lists of these studies are presented in Table 6.

The implications for research and academia

An SLR provides directions for researchers who want to understand the research trends in a specific field. This SLR provides an updated state-of-the-art approach on smells initially for the OO domain and then expands the scope for the

SO paradigm. It also provides guidelines for practitioners working in the software industry to apply the smell detection/correction techniques during software development. The major implications for the academic community working in the area of bad smells are as follows.

- The results of the study are evident in showing that methods based on the static source code analysis received more attention as compared to the dynamic source code analysis. The software industry is revolutionized by adopting various techniques for software development. Therefore, it will be more appropriate, providing complete execution traces online, which will help practitioners pay more attention toward the improvement of the QoS.
- Researchers used algorithms from the area of machine learning and artificial intelligence for the identification of bad smells. The practitioners can change their coding practices and rely on these algorithms for the identification and correction of bad smells, which is having issues in improving detection precision and recall.
- RESTful API providers do not provide the complete version history as per software engineering guidelines.[121] This hinders practitioners to investigate the cause of the evolution of bad smells in service-based systems that continuously evolve. A complete change-log of services related to the SO paradigm should be made available online, which may help new developers to avoid common design/coding mistakes, ie, bad smells. Thus, this will help the industry to further improve the availability and performance issues for service-based systems.
- There is a rapid shift toward the use of industry blogs, especially from INFOQ and J2EE for the identification of new research trends. It will be more helpful if the industry provides some practical standards/guidelines on design patterns along with hands-on examples for service-based systems that may help academia contending with the de facto industry standards.
- There exists an extensive quantity of validation studies reported for OO systems as, often, their versions and codes are publicly available, ie, open source, but we did not find such an extensive validation for bad smells in Web APIs and service-based systems. This is because the service providers often do not provide server-side code retrieval opportunity to the academia working in the SO paradigm. The more recent service providers in the market can benefit from complete version history that can help client APIs to investigate their technical difficulties in their services more resourcefully and capably.
- As the SE research community could not find the most appropriate classification of smells, it is essential to develop an automatic oracle for all types of smells, which can be useful for the industry and academics to apply standardized coding practices.

Nevertheless, it is a difficult and time-consuming task to manually classify and detect smells. Therefore, we believe that an expert advice is required from both the academia and industry to provide a catalog on the categories of smells and their detection approaches for those defined by Brown et al[36] and Fowler et al.[8]

For the researchers, a number of prospects are still open: an automatic detection technique of smells as part of an IDE can be implemented for Eclipse, Visual Studio, or other mainstream development IDEs for developers and designers to avoid smells. In this way, software development tools can rely on the smell detector in helping the developers to design and implement higher-quality systems.

A comprehensive, industrial smell management tool with fully automated detection support and friendly visualization of the variants of smells, as and when they propagate through code segments, would help developers. Existing smell detection techniques can be reorganized so that developers can be more assisted in the detection of smells. An *origin analysis* of smells should also be studied to locate the root cause of smells in systems across their multiple versions.

Threats to validity

The validity of this study mainly concerns the relevant research questions and their conclusion, ie, the relationship between the conclusion and findings.[122]

We try to maximize *internal validity* by applying all the terms associated with smells as reported in Table 14. We also try to maximize internal validity by checking the quality of references with the help of two independent researchers. However, we might have missed some studies that used other terms associated with smells or reported bad smells for Android or iOS development.

To maximize *construct validity*, after reading the abstracts and recording all results, we reported measures in an Excel sheet. We followed the guidelines proposed by Brereton et al[40] and Kitchenham et al.[50] Yet, we performed manual analyses, and the relevancy of different terms associated with smell may limit our findings.

Finally, *external validity* concerns whether we can generalize our results to other studies that may pertain to Android or iOS development. However, in this study, we focus only on two paradigms, eg, OO and SOA, and we keep the other paradigms as our future work.

Moreover, the main threat for this study is the number of terms associated with this SLR. For example, "smells" can be described as "antipattern" or "anti-pattern" or design smell or architectural anomalies or architectural degradation, and so on. We tried to be particularly cautious in data extraction. We searched manually the strings "architectural degradation" or "code anomalies" or "design anomalies." Therefore, although we cover two paradigms in OO and SOA, we might be missing a few relevant studies. Data extraction was carried by two independent researchers who also ensured that the kept/discarded studies are strictly satisfying the PICOC and our inclusion/exclusion criteria as discussed in Section 3. However, other quality checklists can be applied that may increase or decrease the research bias and may lead to different results.

6 | CONCLUSION AND FUTURE WORK

Smells are classified in the literature as code smells, design smells, architectural smells, and antipatterns. We analyzed the most relevant research studies published between January 2000 and December 2017 in different online libraries and investigated five key research questions.

RQ1: *What are the classifications of the state-of-the-art techniques employed in the detection of smells?*

Findings: Mainly two techniques are used in the literature: (1) static source code analysis (eg, behavioral source code analysis, empirical source code analysis, algorithm-based source code analysis, methodology-based source code analysis, and linguistic source code analysis) and (2) dynamic source code analysis based on dynamic threshold adaptation, eg, using a genetic algorithm, instead of fixed thresholds for smell detection.

RQ2: *How did the state-of-the-art approaches evolve across different paradigms starting from OO to SO?*

Findings: A number of different detection techniques that crossed domains are based on (1) source code metrics, (2) mining the source code using SVN or CVS, (3) DSL, (4) genetic algorithm, and (5) PE-A.

RQ3: *What are the smells that are studied for a specific paradigm?*

Findings: In OO, most of the relevant studies reported and analyzed *Feature Envy* as code- and design-level smell. In contrast, much study is still required to be done for the detection of smells like the *Yo-yo Problem*, *Unnamed Coupling*, *Extensive Coupling*, and so on, which gained less attention so far in the SE research community. In SO, antipatterns/smells that received most attention include *God Object Web Service*, *Low-Cohesive Operation*, *Ambiguous Names*, *Chatty Service*, and *Data Web Service*.

RQ4: *What is the correlation between smells across the paradigms?*

Findings: The metrics used for the identification of 20 smells reported for SOA used source code metrics that have also been previously reported for OO. The identification of the *God Object Web Service* used cohesion metrics (eg, LCOM3) and operation identification metrics (eg, NOD) that are also used for the identification of smells in OO. However, the threshold values may vary for SOA in comparison to OO.

RQ5: *What are the trends in research on smells from January 2000 to December 2017?*

Findings: We found a corpus of 34 research studies on the detection of smells. A new research gap was filled by introducing smells in SOAP and REST services. Moreover, there is no study published on the correction of REST antipatterns. The NLP techniques are also used for the detection and correction of smells in OO to the detection of REST linguistic antipatterns.

We identified several issues that should be considered and receive more attention from researchers. We also found several related research activities that must be explored. We advised researchers to pay more attention to linguistic smells that are gaining popularity in the last five years. Moreover, research on the correction of lexical smells requires further investigation. The intersmell relationship for lexical smells and performance evaluations of lexical design patterns vs antipatterns are yet to be studied. Refactoring is a major area that is well researched for OO smells but not yet for SCA and REST smells due to their complex nature. Developers' maintenance effort for smells in SO systems is still not addressed in the recent studies. We were also unable to find any antipatterns detected in Java Enterprise systems, although their detections were performed on SCA systems.[23]

ORCID

Francis Palma http://orcid.org/0000-0001-7092-2244

REFERENCES

1. Meyer B. *Object-Oriented Software Construction*. 2nd ed. New York, NY: Prentice Hall; 1988.
2. Bass L, Clements P, Kazman R. *Software Architecture in Practice*. 2nd ed. Boston, MA: Addison-Wesley; 2003.
3. Breivold HP, Larsson M. Component-based and service-oriented software engineering: key concepts and principles. In: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO); 2007; Lubeck, Germany.
4. Baghdadi Y. Service-oriented software engineering: a guidance framework for service engineering methods. *Int J Syst Serv-Oriented Eng*. 2015;5(2):1-19.
5. Papazoglou MP, Traverso P, Dustdar S, Leymann F. Service-oriented computing: a research roadmap. *Int J Coop Inf Syst*. 2008;17(02):223-255.
6. Papazoglou MP. Service-oriented computing: concepts, characteristics and directions. In: Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE); 2003; Rome, Italy.
7. Booch G. *Object-Oriented Analysis & Design With Applications*. London, UK: Pearson Education; 2006.
8. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Professional; 1999.
9. Tufano M, Palomba F, Bavota G, et al. When and why your code starts to smell bad. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1 (ICSE); 2015; Florence, Italy.
10. Izurieta C. *Decay and Grime Buildup in Evolving Object Oriented Design Patterns* [dissertation]. Fort Collins, CO: Colorado State University; 2009.
11. Gravino C, Risi M, Scanniello G, Tortora G. Does the documentation of design pattern instances impact on source code comprehension? Results From Two Controlled Experiments. Paper presented at: 2011 18th Working Conference on Reverse Engineering (WCRE); 2011; Limerick, Ireland.
12. Prechelt L, Unger-Lamprecht B, Philippsen M, Tichy WF. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Trans Softw Eng*. 2002;28:595-606.
13. Biffl S, Gutjahr W. Influence of team size and defect detection technique on inspection effectiveness. In: Proceedings of the 7th International Symposium on Software Metrics (METRICS); 2001; London, UK.
14. Mäntylä MV, Vanhanen J, Lassenius C. Bad smells-humans as code critics. In: Proceedings of the 20th IEEE International Conference on Software Maintenance; 2004; Chicago, IL.
15. Maiga A, Ali N, Bhattacharya N, Sabane A, Gueheneuc Y-G, Aimeur E. SMURF: a SVM-based incremental anti-pattern detection approach. Paper presented at: 2012 19th Working Conference on Reverse Engineering (WCRE); 2012; Kingston, Canada.
16. Khomh F, Vaucher S, Guéhéneuc Y-G, Sahraoui H. BDTEX: a GQM-based Bayesian approach for the detection of antipatterns. *J Syst Softw*. 2011;84:559-572.
17. Moha N, Guéhéneuc Y-G, Duchien L, Le Meur A-F. DECOR: a method for the specification and detection of code and design smells. *IEEE Trans Softw Eng*. 2010;36:20-36.
18. Palma F, Moha N, Tremblay G, Guéhéneuc Y-G. Specification and detection of SOA antipatterns in web services. In: Proceedings of the European Conference on Software Architecture; 2014; Vienna, Austria.
19. Mateos C, Rodriguez JM, Zunino A. A tool to improve code-first web services discoverability through text mining techniques. *Softw Pract Exper*. 2015;45:925-948.
20. Coscia JLO, Mateos C, Crasso M, Zunino A. Anti-pattern free code-first web services for state-of-the-art Java WSDL generation tools. *Int J Web Grid Serv*. 2013;9:107-126.
21. Palma F, Dubois J, Moha N, Guéhéneuc Y-G. Detection of REST patterns and antipatterns: a heuristics-based approach. In: Proceedings of the International Conference on Service-Oriented Computing; 2014; Paris, France.
22. Palma F, Gonzalez-Huerta J, Moha N, Guéhéneuc Y-G, Tremblay G. Are RESTful APIs well-designed? Detection of their linguistic (anti) patterns. In: Proceedings of the International Conference on Service-Oriented Computing; 2015; Goa, India.

23. Palma F, Nayrolles M, Moha N, Guéhéneuc Y-G, Baudry B, Jézéquel J-M. SOA antipatterns: an approach for their specification and detection. *Int J Coop Inf Syst*. 2013;22:1341004.
24. Wangberg R. *A Literature Review on Code Smells and Refactoring* [master thesis]. Oslo, Norway: University of Oslo; 2010.
25. Zhang M, Hall T, Baddoo N. Code bad smells: a review of current knowledge. *J Softw Maint Evol Res Pract*. 2011;23:179-202.
26. Rasool G, Arshad Z. A review of code smell mining techniques. *J Softw: Evol Process*. 2015;27:867-895.
27. Al Dallal J. Identifying refactoring opportunities in object-oriented code: a systematic literature review. *Inf Softw Technol*. 2015;58:231-249.
28. Lehnert S. *A Review of Software Change Impact Analysis*. Technical Report. Ilmenau, Germany: Ilmenau University of Technology; 2011.
29. Soares G, Gheyi R, Murphy-Hill E, Johnson B. Comparing approaches to analyze refactoring activity on software repositories. *J Syst Softw*. 2013;86:1006-1022.
30. Hall T, Beecham S, Bowes D, Gray D, Counsell S. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans Softw Eng*. 2012;38:1276-1304.
31. Rattan D, Bhatia R, Singh M. Software clone detection: a systematic review. *Inf Softw Technol*. 2013;55:1165-1199.
32. Roy CK, Cordy JR. *A Survey on Software Clone Detection Research*. Technical Report. Kingston, Canada: School of Computing, Queen's University; 2007. TR No 2007-541.
33. Wake WC. *Refactoring Workbook*. Boston, MA: Addison-Wesley Professional; 2004.
34. Mäntylä M, Vanhanen J, Lassenius C. A taxonomy and an initial empirical study of bad smells in code. In: Proceedings of the International Conference on Software Maintenance (ICSM); 2003; Monterey, CA.
35. Jaafar F, Guéhéneuc Y-G, Hamel S, Khomh F. Mining the relationship between anti-patterns dependencies and fault-proneness. In: Proceedings of the 2013 20th Working Conference on Reverse Engineering (WCRE); 2013; Koblenz, Germany.
36. Brown WH, Malveau RC, McCormick HW, Mowbray TJ. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. Hoboken, NJ: John Wiley & Sons, Inc.; 1998.
37. Mens T, Tourwé T. A survey of software refactoring. *IEEE Trans Softw Eng*. 2004;30:126-139.
38. Misbhauddin M, Alshayeb M. UML model refactoring: a systematic literature review. *Empir Softw Eng*. 2015;20:206-251.
39. Laguna MA, Crespo Y. A systematic mapping study on software product line evolution: from legacy system reengineering to product line refactoring. *Sci Comput Program*. 2013;78:1010-1034.
40. Brereton P, Kitchenham BA, Budgen D, Turner M, Khalil M. Lessons from applying the systematic literature review process within the software engineering domain. *J Syst Softw*. 2007;80:571-583.
41. Hall T, Zhang M, Bowes D, Sun Y. Some code smells have a significant but small effect on faults. *ACM Trans Softw Eng Methodol*. 2014;23. Article No. 33.
42. Wohlin C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE); 2014; London, UK.
43. Khomh F, Di Penta M, Yann-Gaël G, Giuliano A. An Exploratory Study of the Impact of Software Changeability. 2009.
44. Kreimer J. Adaptive detection of design flaws. *Electron Notes Theor Comput Sci*. 2005;141:117-136.
45. Fontana FA, Braione P, Zanoni M. Automatic detection of bad smells in code: an experimental assessment. *J Object Technol*. 2012;11(2):1-38.
46. Dybå T, Dingsøyr T. Empirical studies of agile software development: a systematic review. *Inf Softw Technol*. 2008;50(9-10):833-859.
47. Erl T. *Service-Oriented Architecture: Concepts, Technology, and Design*. Boston, MA: Pearson Education Inc; 2005.
48. Oliveto R, Khomh F, Antoniol G, Guéhéneuc Y-G. Numerical signatures of antipatterns: an approach based on B-splines. Paper presented at: 2010 14th European Conference on Software Maintenance and Reengineering (CSMR); 2010; Madrid, Spain.
49. Cruzes DS, Dybå T. Research synthesis in software engineering: a tertiary study. *Inf Softw Technol*. 2011;53:440-455.
50. Kitchenham B, Brereton OP, Budgen D, Turner M, Bailey J, Linkman S. Systematic literature reviews in software engineering—a systematic literature review. *Inf Softw Technol*. 2009;51:7-15.
51. Atkinson P, Hammersley M. Ethnography and Participant Observation. 1994.
52. Bandi A, Williams BJ, Allen EB. Empirical evidence of code decay: a systematic mapping study. Paper presented at: 2013 20th Working Conference on Reverse Engineering (WCRE); 2013; Koblenz, Germany.
53. Yamashita A, Moonen L. To what extent can maintenance problems be predicted by code smell detection? - an empirical study. *Inf Softw Technol*. 2013;55:2223-2242.
54. Taba SES, Khomh F, Zou Y, Hassan AE, Nagappan M. Predicting bugs using antipatterns. Paper presented at: 2013 IEEE International Conference on Software Maintenance; 2013; Eindhoven, The Netherlands.
55. Liu H, Ma Z, Shao W, Niu Z. Schedule of bad smell detection and resolution: a new way to save effort. *IEEE Trans Softw Eng*. 2012;38:220-235.
56. Jaafar F, Khomh F, Guéhéneuc Y-G, Zulkernine M. Anti-pattern mutations and fault-proneness. In: Proceedings of the 2014 14th International Conference on Quality Software (QSIC); 2014; Dallas, TX.
57. Polášek I, Líška P, Kelemen J, Lang J. On extended similarity scoring and bit-vector algorithms for design smell detection. Paper presented at: 2012 IEEE 16th International Conference on Intelligent Engineering Systems (INES); 2012; Lisbon, Portugal.
58. Ganea G, Verebi I, Marinescu R. Continuous quality assessment with inCode. *Sci Comput Program*. 2015;134;19-36.

59. Kitchenham B, Charters S. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical Report. Durham, UK: University of Durham; 2007. Version 2.3: EBSE Technical Report EBSE-2007-01.
60. Coscia JLO, Mateos C, Crasso M, Zunino A. Avoiding WSDL bad practices in code-first web services. In: Proceedings of the 12th Argentine Symposium on Software Engineering (ASSE)-40th JAIIO; 2011; Córdoba, Argentina.
61. Fontana FA, Ferme V, Marino A, Walter B, Martenka P. Investigating the impact of code smells on system's quality: an empirical study on systems of different application domains. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM); 2013; Eindhoven, The Netherlands.
62. Palma F, An L, Khomh F, Moha N, Guéhéneuc Y-G. Investigating the change-proneness of service patterns and antipatterns. Paper presented at: 2014 IEEE 7th International Conference on Service-Oriented Computing and Applications (SOCA); 2014; Matsue, Japan.
63. Nguyen HV, Nguyen HA, Nguyen TT, Nguyen AT, Nguyen TN. Detection of embedded code smells in dynamic web applications. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering; 2012; Essen, Germany.
64. Mateos C, Crasso M, Zunino A, Coscia JLO. Revising WSDL documents: why and how, part 2. *IEEE Internet Comput*. 2013;17:46-53.
65. Marinescu R, Marinescu C. Are the clients of flawed classes (also) defect prone? Paper presented at: 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation; 2011; Williamsburg, VI.
66. Khomh F, Di Penta M, Guéhéneuc Y-G, Antoniol G. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empir Softw Eng*. 2012;17:243-275.
67. Sabané A, Di Penta M, Antoniol G, Guéhéneuc Y-G. A study on the relation between antipatterns and the cost of class unit testing. Paper presented at: 2013 17th European Conference on Software Maintenance and Reengineering; 2013; Genova, Italy.
68. Bavota G, De Lucia A, Di Penta M, Oliveto R, Palomba F. An experimental investigation on the innate relationship between quality and refactoring. *J Syst Softw*. 2015;107:1-14.
69. Walter B, Alkhaeir T. The relationship between design patterns and code smells: an exploratory study. *Inf Softw Technol*. 2016;74:127-142.
70. Arnaoudova V, Di Penta M, Antoniol G. Linguistic antipatterns: what they are and how developers perceive them. *Empir Softw Eng*. 2016;21:104-158.
71. Guerrouj L, Kermansaravi Z, Arnaoudova V, et al. Investigating the relation between lexical smells and change- and fault-proneness: an empirical study. *Softw Qual J*. 2015;25(3):641-670.
72. Abebe SL, Arnaoudova V, Tonella P, Antoniol G, Guéhéneuc Y-G. Can lexicon bad smells improve fault prediction? Paper presented at: 2012 19th Working Conference on Reverse Engineering; 2012; Kingston, Canada.
73. Jaafar F, Lozano A, Guéhéneuc Y-G, Mens K. Analyzing software evolution and quality by extracting asynchrony change patterns. *J Syst Softw*. 2017;131:311-322.
74. Ujhelyi Z, Szőke G, Horváth Á, et al. Performance comparison of query-based techniques for anti-pattern detection. *Inf Softw Technol*. 2015;65:147-165.
75. Llano MT, Pooley R. UML specification and correction of object-oriented anti-patterns. Paper presented at: Fourth International Conference on Software Engineering Advances; 2009; Porto, Portugal.
76. Rodriguez JM, Crasso M, Mateos C, Zunino A. Best practices for describing, consuming, and discovering web services: a comprehensive toolset. *Softw Pract Exper*. 2013;43:613-639.
77. Petrillo F, Merle P, Moha N, Guéhéneuc Y-G. Are REST APIs for cloud computing well-designed? An exploratory study. Paper presented at: The 14th International Conference on Service-Oriented Computing; 2016; Banff, Canada.
78. Wang H, Kessentini M, Ouni A. Prediction of web services evolution. Paper presented at: The 14th International Conference on Service-Oriented Computing; 2016; Banff, Canada.
79. Salvatierra G, Mateos C, Crasso M, Zunino A. Towards a computer assisted approach for migrating legacy systems to SOA. Paper presented at: 12th International Conference on Computational Science and Its Applications; 2012; Salvador de Bahia, Brazil.
80. Ouni A, Gaikovina Kula R, Kessentini M, Inoue K. Web service antipatterns detection using genetic programming. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation; 2015; Madrid, Spain.
81. Ouni A, Kessentini M, Inoue K, Cinnéide MO. Search-based web service antipatterns detection. *IEEE Trans Serv Comput*. 2015;10(4):603-617.
82. Nayrolles M, Moha N, Valtchev P. Improving SOA antipatterns detection in service-based systems by mining execution traces. Paper presented at: 2013 20th Working Conference on Reverse Engineering (WCRE); 2013; Koblenz, Germany.
83. Coscia JLO, Mateos C, Crasso M, Zunino A. Refactoring code-first web services for early avoiding WSDL anti-patterns: approach and comprehensive assessment. *Sci Comput Program*. 2014;89:374-407.
84. Ouni A, Kessentini M, Sahraoui H, Hamdi MS. Search-based refactoring: towards semantics preservation. Paper presented at: 2012 28th IEEE International Conference on Software Maintenance (ICSM); 2012; Trento, Italy.
85. Petticrew M, Roberts H. *Systematic Reviews in the Social Sciences: A Practical Guide*. Hoboken, NJ: John Wiley & Sons; 2008.
86. Kessentini M, Mahaouachi R, Ghedira K. What you like in design use to correct bad-smells. *Softw Qual J*. 2013;21:551-571.
87. Maneerat N, Muenchaisri P. Bad-smell prediction from software design model using machine learning techniques. Paper presented at: 2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE); 2011; Nakhon Pathom, Thailand.
88. Fontana FA, Ferme V, Spinelli S. Investigating the impact of code smells debt on quality code evaluation. In: Proceedings of the Third International Workshop on Managing Technical Debt; 2012; Zürich, Switzerland.
89. Danphitsanuphan P, Suwantada T. Code smell detecting tool and code smell-structure bug relationship. Paper presented at: 2012 Spring Congress on Engineering and Technology; 2012; Xi'an, China.

90. Kessentini W, Kessentini M, Sahraoui H, Bechikh S, Ouni A. A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Trans Softw Eng*. 2014;40:841-861.
91. Sjøberg DI, Yamashita A, Anda BC, Mockus A, Dybå T. Quantifying the effect of code smells on maintenance effort. *IEEE Trans Softw Eng*. 2013;39:1144-1156.
92. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D. Detecting bad smells in source code using change history information. Paper presented at: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE); 2013; Silicon Valley, CA.
93. Yamashita A, Moonen L. Exploring the impact of inter-smell relations on software maintainability: an empirical study. Paper presented at: 2013 35th International Conference on Software Engineering (ICSE); 2013; San Francisco, CA.
94. Ouni A, Kessentini M, Sahraoui H, Boukadoum M. Maintainability defects detection and correction: a multi-objective approach. *Autom Softw Eng*. 2013;20:47-79.
95. Liu H, Guo X, Shao W. Monitor-based instant software refactoring. *IEEE Trans Softw Eng*. 2013;39:1112-1126.
96. Ouni A, Kessentini M, Sahraoui H, Inoue K, Hamdi MS. Improving multi-objective code-smells correction using development history. *J Syst Softw*. 2015;105:18-39.
97. dos Santos Neto BF, Ribeiro M, da Silva VT, Braga C, de Lucena CJP, de Barros Costa E. *AutoRefactoring*: a platform to build refactoring agents. *Expert Syst Appl*. 2015;42:1652-1664.
98. Palomba F, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A. Mining version histories for detecting code smells. *IEEE Trans Softw Eng*. 2015;41(5):462-489.
99. Liu H, Liu Q, Niu Z, Liu Y. Dynamic and automatic feedback-based threshold adaptation for code smell detection. *IEEE Trans Softw Eng*. 2016;42:544-558.
100. Marinescu R. Measurement and quality in object-oriented design. Paper presented at: 21st IEEE International Conference on Software Maintenance (ICSM); 2005; Budapest, Hungary.
101. Marinescu R. Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM); 2004; Chicago, IL.
102. Munro MJ. Product metrics for automatic identification of "bad smell" design problems in Java source-code. Paper presented at: IEEE International Symposium on Software Metrics; 2005; Como, Italy.
103. Salehie M, Li S, Tahvildari L. A metric-based heuristic framework to detect object-oriented design flaws. Paper presented at: International Conference on Program Comprehension; 2006; Athens, Greece.
104. Stoianov A, Şora I. Detecting patterns and antipatterns in software using Prolog rules. Paper presented at: 2010 International Joint Conference on Computational Cybernetics and Technical Informatics (ICCC-CONTI); 2010; Timisoara, Romania.
105. Wong S, Cai Y, Kim M, Dalton M. Detecting software modularity violations. In: Proceedings of the 33rd International Conference on Software Engineering; 2011; Waikiki, HI.
106. Marinescu R, Raţiu D. Quantifying the quality of object-oriented design: the factor-strategy model. In: Proceedings of the 11th Working Conference on Reverse Engineering; 2004; Delft, The Netherlands.
107. Trifu A, Seng O, Genssler T. Automated design flaw correction in object-oriented systems. In: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR); 2004; Tempere, Finland.
108. Hassaine S, Khomh F, Guéhéneuc Y-G, Hamel S. IDS: an immune-inspired approach for the detection of software design smells. Paper presented at: 2010 Seventh International Conference on the Quality of Information and Communications Technology; 2010; Porto, Portugal.
109. Arnaoudova V, Di Penta M, Antoniol G, Guéhéneuc Y-G. A new family of software anti-patterns: linguistic anti-patterns. Paper presented at: 2013 17th European Conference on Software Maintenance and Reengineering; 2013; Genova, Italy.
110. Luo Y, Hoss A, Carver DL. An ontological identification of relationships between anti-patterns and code smells. Paper presented at: 2010 IEEE Aerospace Conference; 2010; Big Sky, MT.
111. Simon F, Steinbrückner F, Lewerentz C. Metrics-based refactoring. Paper presented at: Fifth European Conference on Software Maintenance and Reengineering; 2001; Lisbon, Portugal.
112. Moha N, Guéhéneuc Y-G, Le Meur A-F, Duchien L, Tiberghien A. From a domain analysis to the specification and detection of code and design smells. *Form Asp Comput*. 2010;22:345-361.
113. Sjøberg DI, Dybå T, Jorgensen M. The future of empirical methods in software engineering research. Paper presented at 2007 Future of Software Engineering (FOSE); 2007; Minneapolis, MN.
114. Bashir I, Goel AL. *Testing Object-Oriented Software: Life Cycle Solutions*. New York, NY: Springer Science & Business Media; 2012.
115. Fontana FA, Mäntylä MV, Zanoni M, Marino A. Comparing and experimenting machine learning techniques for code smell detection. *Empir Softw Eng*. 2016;21:1143-1191.
116. Lehnert S. A taxonomy for software change impact analysis. In: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution; 2011; Szeged, Hungary.
117. Karsai G, Krahn H, Pinkernell C, Rumpe B, Schindler M, Völkel S. Design guidelines for domain specific languages. arXiv preprint arXiv:1409.2378; 2014.
118. Král J, Žemlička M. Crucial service-oriented antipatterns. *Int J Adv Softw*. 2008;2:160-171. International Academy, Research and Industry Association.
119. Heß A, Johnston E, Kushmerick N. *ASSAM: A Tool for Semi-Automatically Annotating Semantic Web Services*. In: McIlraith SA, Plexousakis D, van Harmelen F, eds. Berlin, Germany: Springer; 2004:320-334. *Lecture Notes in Computer Science*; vol. 3298.

120. Etzkorn LH, Gholston SE, Fortune JL, et al. A comparison of cohesion metrics for object-oriented systems. *Inf Softw Technol*. 2004;46:677-687.

121. D'Ambros M, Gall H, Lanza M, Pinzger M. Analysing software repositories to understand software evolution. In: *Software Evolution*. Berlin, Germany: Springer; 2008:37-67.

122. Feldt R, Magazinius A. Validity threats in empirical software engineering research—an initial survey. Paper presented at: The 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE); 2010; Redwood City, CA.

123. Roy CK, Cordy JR, Koschke R. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *Sci Comput Program*. 2009;74:470-495.

124. Pate JR, Tairas R, Kraft NA. Clone evolution: a systematic review. *J Softw: Evol Process*. 2013;25:261-283.

125. Deb K, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans Evol Comput*. 2002;6(2):182-197.

SUPPORTING INFORMATION

Additional supporting information may be found online in the Supporting Information section at the end of the article.

How to cite this article: Sabir F, Palma F, Rasool G, Guéhéneuc Y-G, Moha N. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. *Softw Pract Exper*. 2018;1–37. https://doi.org/10.1002/spe.2639