Oracles of Bad Smells - a Systematic Literature Review

Rafael Prates Ferreira Trindade
Computer Science Department
Federal University of Minas Gerais
Belo Horizonte, M.G., Brazil
rafael.trindade@dcc.ufmg.br

Mariza Andrade da Silva Bigonha
Computer Science Department
Federal University of Minas Gerais
Belo Horizonte, M.G., Brazil
mariza@dcc.ufmg.br

Kecia Aline Marques Ferreira
Department of Computing
Federal Center for Technological Education of Minas Gerais
Belo Horizonte, M.G., Brazil
kecia@cefetmg.br

ABSTRACT

A bad smell is an evidence of a design problem that may be harmful to the software maintenance. Several studies have been carried out to aid the identification of bad smells, by defining approaches or tools. Usually, the evaluation of these studies' results relies on data of oracles bad smells. An oracle is a set of data of bad smells found in a given software system. Such data serves as a referential template or a benchmark to evaluate the proposals on detecting bad smells. The availability and the quality of bad smell oracles are crucial to assert the quality of detection strategies of bad smells. This study aims to compile the bad smell oracles proposed in the literature. To achieve this, we conducted a Systematic Literature Review (SLR) to identify bad smell oracles and their characteristics. The main result of this study is a catalog of bad smell oracles that may be useful for research on bad smells, especially the studies that propose tools or detection strategies for bad smells.

KEYWORDS

bad smell, code smell, design anomaly, benchmark, oracle, systematic literature review

ACM Reference Format:
Rafael Prates Ferreira Trindade, Mariza Andrade da Silva Bigonha, and Kecia Aline Marques Ferreira. 2020. Oracles of Bad Smells - a Systematic Literature Review. In *Proceedings of CBSoft (SBES'20)*. Brazil, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Beck and Fowler coined the term *bad smell* to identify quality issues in code that may be refactored to improve software maintainability. They refer *bad smells* as " structures in the code that suggest the possibility of refactoring" [18]. An excessive amount of bad smells in a software system makes it hard to maintain and evolve [50]. Identifying the parts of the system that should be refactored is need to control the software architecture complexity [27]. The most widespread definitions of bad smells in the literature are postulated by Fowler et al. [18], being a total of 22 bad smells. Brown et al. [6] describe 14 bad smells that have been categorized into three distinct phases of the software life cycle: management, architecture, and development. Identifying bad smells manually in large software systems is not viable. So, to overcome such difficulties, many studies carried out to define strategies and tools for bad smell detection. One of the first studies in this sense is by [33] that proposes a metric-based approach for detecting bad smells. The study of Fernandes et al. [15] found 84 tools, which intended to detect 61 bad smells in total. Overall, these tools apply six different bad smell detection techniques. Fernandes et al. [15] compared four tools regarding the detection of Large Class and Long Method bad smells. They found that the tools have divergent results when inspecting the same set of software systems.

The studies usually rely on the oracles of bad smells to evaluate the efficiency of detection approaches and tools. In this context, an oracle is a set of data considered as a benchmark, which is compared to a resulting set with the same category of data to evaluate the quality of this resulting set. Regarding bad smells, an oracle is a set of data, which reports actual instances of bad smells in a software system. For instance, the study of Fernandes et al. [15] used an oracle of two bad smells of a single software system.

According to Lavoie and Merlo [28], good oracles need to deal with systems large enough and will depend of a thorough analysis of the code to have any practical interest. We may generate a bad smell oracle by (i) manual inspection of a system, (ii) automatic scanning, or (iii) combining both techniques. In a manual approach, specialists point out instances of bad smells in the source code. As the oracles depend on manual inspection in this technique, its subjectiveness is high [44]. Therefore, it is complex to generate a trustful oracle. On the other hand, the evaluation of automatic approaches demands a previous oracle that needs to be proven precise. Therefore, the existence of reliable bad smell oracles is of central importance in the development of automatic techniques to detect bad smells.

This work aims to identify bad smell oracles proposed in the literature, how they were constructed, their main characteristics and where they are available. For this purpose, we carried out a Systematic Literature Review (SLR). The aim of a SLR is "identifying, assessing, and interpreting all available evidence relevant to a specific issue, thematic area, or phenomenon of interest" [26]. As a result, we identified 51 studies that directly or indirectly produced bad smell oracles. At least six programming languages have been the focus of the oracles: Java, C, CPP, JavaScript, and Scratch. We also have found that the automatic approach is the most popular, *i.e.*, usually bad smell oracles rely on tools. JDeodorant has been the most used tool to create bad smell oracles. Moreover, each oracle found in the literature focuses on specific bad smells Blob bad smell

SBES'20, August 2020, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

is the most commonly used. We also identified 12 oracles of bad smells available online.

The contributions of this work are mainly useful for researchers who carry out studies on bad smells, since oracles of bad smells may be used in the creation and evaluation of detection approaches, in the evaluation of tools, and in the conduction of empirical studies on bad smells.

We organize the remaining of this paper as follows. Section 2 presents the related work. Section 3 shows the criteria used to conduct this Systematic Literature Review and presents the research questions. Section 4 presents the results. Section 6 discusses the threats to the validity of our study. Section 7 brings a conclusion.

2 RELATED WORK

According Zhang et al. [61], most of the studies on bad smell was published between 2000 and 2009. Consider the definitions given by Fowler et al. [18], being Duplicated Code highlighted in more than 50% of the studies. Besides, approximately half (49%) of the studies propose a method or tool to detect bad smells. The works that propose techniques to detect bad smells usually compare their approaches with data of an oracle. An oracle is supposed to have all the real occurrences of a given bad smell in a given software system. The correctness of the oracle is, therefore, a severe threat to the studies on bad smell. Moreover, providing proper oracles is essential to ensure the validity of such studies.

Sobrinho et al. [52] conducted an extensive Systematic Literature Review to determine the state of the art on bad smells with 351 works produced between 1990 and 2017. Among the results, the authors identified that there are divergences on the impacts of bad smells on software maintainability. Some studies consider the use of code with a bad smell is the best option for development. Sobrinho et al. [52] also found that the most common motivation of the analyzed works, more than 30 % of the studies set, is creating a tool or technique for detecting bad smells. Among the future directions, the authors indicate the lack of representative benchmarks. In this sense, their findings justify the conduction of our work, since this SLR compiles the oracles of bad smells provided by the literature, as well as indicates its approaches, characteristics, and location.

Palomba et al. [44] point out the main dilemmas to be overcome in the construction of a manual oracle: the subjectivity, the difficulty in comparing different techniques, the inviability to generalize results, and the effort involved in the construction of an oracle.

Another problem is that there is no consensus about the bad smell, and, for this reason, oracles generated for the same software system may have disagreements. Olbrich et al. [37] indicate that a manual detection of bad smells through code inspection leads to three problems: time-consuming, non-replicable, and non-scalable.

It is important to identify previous works that have proposed bad smell oracles, considering all the difficulties to generate one. Palomba et al. [44] claim that they are not aware of a wider repository of bad smells than the one generated by them, whose volume comprises 20 software.

Our work differs from the ones presented in this section in the following points: we identify 51 oracles of bad smell proposed in the literature, being 12 available online. Moreover, we characterize the oracles we have found in three aspects: (i) bad smells considered in the oracle, (ii) programming language and size of the software systems used to construct the oracle, and (iii) approach used to create the oracle - manual, tool, or both.

3 STUDY DESIGN

The studies identified by an SLR may be classified as primary studies whereas the SLR itself is a secondary study. In this work, the primary studies are scientific publications.

The SLR presented in this paper addresses studies on bad smell oracles. The objective of this SLR is to identify works that produce oracles and the characteristics of the oracles. We perform the SLR in three steps: planning, execution, and analysis. In Section 3.1 we present the planning phase. Section 3.2 describes the execution, showing the steps and the results of the primary's studies selection process. Section 3.3 ehibits the most relevant points of some selected papers.

3.1 Planning Step

In this step, we defined all the necessary elements for the SLR elaboration: (i) the research questions, which we will investigate, (ii) the search string we will use, (iii) the inclusion and exclusion criteria of the primary studies, and (iv) the databases to be searched.

Research Questions. This study investigates five research questions (RQ) as follows.

RQ1: Which oracles for bad smells have been proposed in the literature?

RQ2: In which programming languages the systems composing the oracles are implemented?

RQ3: What are the size of the systems composing the oracles?

RQ4: Which bad smells are considered by the oracles?

RQ5: Which approaches have been used to create the oracles?

Search String. The search string is used to find primary studies in the selected databases. To define the search string, we have identified the most relevant keywords related to the proposed research questions and the synonyms of these keywords.

"Bad smell" is not the only term used in the literature. Other terms, such as "code smell" and "anomaly", have the same meaning, so they were also included in the definition of the search string as follows.

(("oracle" OR "benchmark") AND (("anti-pattern" AND "software") OR ((("bad" OR "design" OR "code" OR "architecture") AND ("smell")) OR (("design" OR "code") AND "anomaly"))))

To search in different electronic repositories, we have applied the search only in metadata, i.e., title, abstract, and keywords. Throughout the decision process of the search string, we made some adjustments in the considered terms. After the first round of search, we observed that the term "oracle" is frequently used in the context of studies on bad smells. We also found the term benchmark as a synonym for oracle. The term anti-pattern was the last one inserted in the string since we have realized that there are works that use

the term as a synonym for bad smell. However, to limit the universe in which this term appears, we inserted the term software. The searches have been carried out with all search string proposals, and we considered their results to form the basis of primary studies included in the SLR.

Inclusion and Exclusion Criteria. These criteria allow us to classify each paper found as a candidate to be included or excluded from the SLR, allowing it to restricted to the explored topic. Table 1 summarizes the inclusion and the exclusion criteria applied in this work.

Table 1: Inclusion and Exclusion Criteria

Inclusion Criteria	Exclusion Criteria
Works published in Computer Science	Duplicate Articles
Works written in English	Documents classified as tutorials, posters, panels, lectures or workshops
Works available in electronic format	Articles that could not be found
Works published in conferences or journals	Chapter of books published in conferences or magazines
Works related to the topic of this study	

Databases. We used the following digital libraries: ACM Digital Library[1], IEEE Xplore[2], Science Direct[3](SD), Scopus[4], Springer[5], Web of Science[6], Engineering Village[7], and Google Scholar[8]. We chosen them because they are electronic databases that have an extensive collection of full papers published at relevant conferences and journals to the academic community.

3.2 Execution Step

This step consists of applying the search string on the chosen databases, and the inclusion and the exclusion criteria of studies.

Search Process. We applied the search string to the search engine of each digital library and incorporated the results into a single worksheet for subsequent processing. In the results from the Science Direct and Web of Science, the data was included in the spreadsheet manually because such digital libraries did not export data to the spreadsheet format. To retrieve data from Google Scholar as a search source, we have used the Publish or Perish[9] program, which provides a list of relevant articles that fit the search string. Observe that we have used it because Google Scholar does not provide the functionality to download the results as a whole.

[1]http://dl.acm.org/
[2]http://ieeexplore.ieee.org/
[3]http://www.sciencedirect.com/
[4]http://www.scopus.com/
[5]https://link.springer.com/
[6]http://webofknowledge.com/
[7]http://www.engineeringvillage.com/
[8]https://scholar.google.com.br/
[9]https://harzing.com/resources/publish-or-perish

Table 2: Number of works returned by each digital library

Digital Library	Number of works
Google Scholar (Google)	901
Scopus (SC)	804
Engineering Village (EV)	240
ACM Digital Library (ACM)	210
IEEE Xplore (IEEE)	348
Web of Science (WoS)	164
Springer (SP)	1009
ScienceDirect (SD)	1272

In the total, 4,948 primary studies have been returned by executing the search string in each of the eight digital repositories, as exhibited in Table 2. Science Direct has returned 1,272, which is almost eight times more than Web of Science, which returned the lowest number of studies, 164.

Selection Process. After performing the search process, we have selected the primary studies to be considered. For this purpose, we have used the inclusion and exclusion criteria defined at the Planning Step.

Figure 1 illustrates the phases defined for selecting the works after executing the search string in the eight digital libraries. The initial search returned 4,948 studies. We have performed five refinements during the execution phase. The purpose of these refinements is to select a final set of primary studies that present an oracle for bad smell. The first refinement removed duplicate documents, resulting in 4382 documents. We discarded duplication of studies, i.e., works with the same title and authors.

The second refinement eliminates studies whose type of work is not in the scope of this SLR. We removed studies representing chapters of books, patent registrations, documents classified as tutorials, posters, panels, lectures or workshops, according to the exclusion criteria shown in the Table 1. After this stage, 4,128 articles remained. In the third refinement, we analyzed the remaining studies verifying if they met the purpose of this systematic literature review. For that, we read the titles, summaries, and conclusions of the articles. At the end of this phase, 426 articles remained.

Subsequently, we performed the fourth refinement. At this stage, we read all primary studies completely to ensure that they serve for our SLR. We performed the text reading to determine the studies that produced oracles of bad smells. In this process, we read the texts aiming to identify descriptions of software analysis process that identified bad smells, as well as, lists, tables or files indicating the occurrence of a bad smell by class or method. The resulting set has 46 studies. Then, we conducted the procedure known as backward snowballing [57] with the 46 articles. In this fifth refinement, we checked the reference list of each study. The goal was to find other studies not included in the existing set but was related to bad smell oracles. We included five studies after this stage, resulting in 51 articles at the end. We selected these 51 studies to Analysis Step.

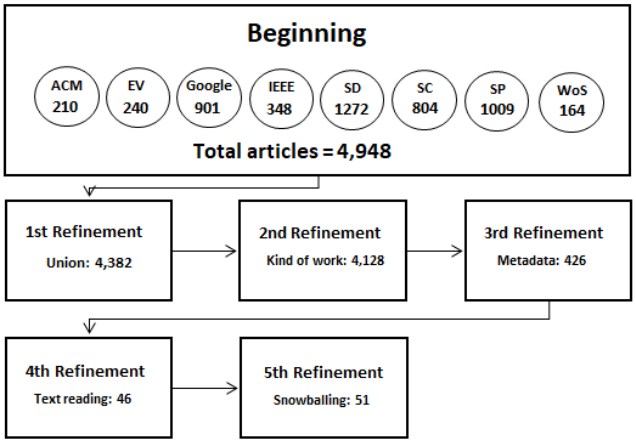

Figure 1: Phases of the selection of works.

Figure 2 illustrates the distribution of primary studies among each digital library in every refinement phase. IEEE Xplore is the library that most contributed to the research, accounting for more than half of the total. Despite returning the most amount of initial results, Science Direct provides only one article for the study. The Scopus and Web of Science databases ended up not having any article considered for the study as well. It is worth noting that the first refinement took into account the alphabetical ordering of the base name, which may have masked the contributions.

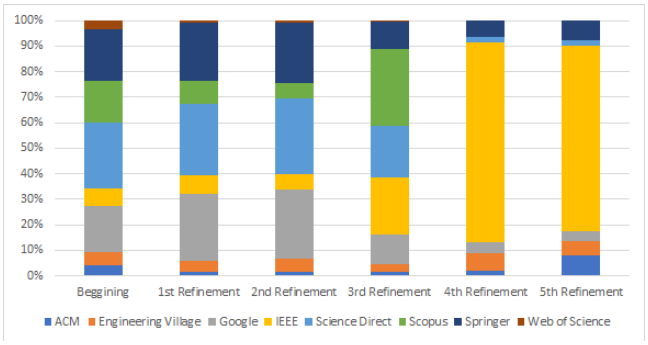

Figure 2: Distribution of the works after each refinement phase

3.3 Analysis Step

In this step, we analyze the results to answer the research questions with the data extracted from the 51 papers selected in the Execution Step.

Data Extraction. We conducted a careful reading of the 51 primary studies. For each one, we wrote a summary compiling the data needed to answer the research questions. Afterward, we created a sheet containing the bad smell name, the approach used to identify the bad smell, and characteristics of the software systems considered in the study. In this phase, we found out that it was only possible to know whether the oracle is available online when the paper reports this information.

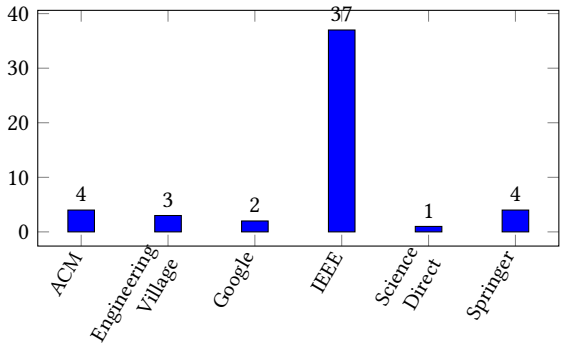

Figure 3: Final distribution of primary studies among the databases

Final Result. After the fifth refinement, we found 51 articles distributed among the databases, according to Figure 3. Two databases returned no contribution to the research theme of this SLR, namely Scopus and Web of Science. More than 70% of the studies come from the IEEE. Nevertheless, it is not possible to affirm that the studies are exclusive results of IEEE Digital Library since the first stage of the refinement classified the bases in alphabetical order, which may have somehow benefited the IEEE.

Studies published outside the traditional academic environment are known as "gray literature" [5]. This type of primary studies may be found by tools like Google Scholar. Google Scholar is also able to find any results that would already be found by other digital repositories. Nevertheless, since all results would go through the Selection Process, we are able to determine only studies that become relevant to answer the research questions in this SLR. The use of Google Scholar is justified, as it provided two studies used in the final set of primary studies considered.

Figure 4 shows the distribution of primary studies published over the years. This analysis considers the whole set of studies, including the five ones found by the snowballing process. We observed that research related to the oracles of bad smells is recent. The apex of publications occurred in 2015 and 2016. The first oracle found was published in 2004, five years after the definitions of bad smell given by Fowler et al. [18].

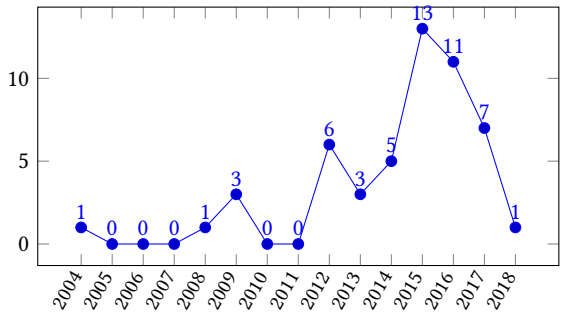

Figure 4: Distribution of primary studies by year of publication

4 RESULTS

This section presents the results of the study by answering the research questions.

RQ1: Which oracles for bad smells have been proposed in the literature?

All the 51 papers recovered in the search present an oracle for bad smells. However, not all oracles have data available online. In some cases, the authors clarify that they may let the oracles available if they receive a formal request. Table 3 presents the studies that define the oracles of bad smells.

Table 3: List of articles that have oracles not available online

References	Approach
[13], [58], [29], [31], [53], [56], [3], [51], [32], [35], [10], [9], [4], [22], [8], [1], [48], [2], [39], [60], [16], [38], [21], [47], [59], [17], [36]	Tool
[54], [37], [14]	Mixed
[20], [37], [55], [49], [62], [12], [43], [40], [11]	Manual

Despite 51 papers create bad smell oracles, only 12 of them are online. Among these, four used a manual approach to their generation, tools created six, and two used a mixed approach to their generation. Table 4 presents the characteristics of them. From left to right, the columns present: (i) reference to the paper, which proposed the oracle, (ii) the software systems in which the oracle is based; (iii) the bad smells considered in the oracle; (iv) the approach applied to construct the oracle - manual, tool, or mixed -, (v) the footnote number, which gives the access link to the oracle.

In some cases, even when a paper mentions a link to an oracle, it may not be found. This happens in the case of [30] whose page is online, but no oracle was found even after a thorough search on it. The link initially provided by [44] is no longer available, but we found the current link[16] after searching.

We noticed that the researcher Fabio Palomba is co-author of five out of the 12 works that resulted in online oracles ([41], [42], [44],[45],[46]). The oracles in these works have many software systems and bad smells in common. Therefore, they may share similar data.

The available bad smell oracles organize the results by software. Most of the studies provide the data in spreadsheets with a line per class and the bad smells found in the class ([25], [46], [24], [42], [45], [41]). The oracle of Hecht et al. [19] indicates the method where the bad smell is located. A single oracle (Chen and Jiang [7]) presents the data in a text file, using special characters to separate classes with a bad smell.

RQ2: In which programming languages the systems composing the oracles are implemented?

The oracles of bad smells usually are based on software systems developed in a particular programming language. With this research question, we aim at identifying the programming languages used to develop the systems of the oracles proposed in the literature. Figure 5 shows the number of oracles per programming language, where Java is the most used programming language, including Android Java, Java Web, or Java Ahead based software systems. Two oracles use Java and C, resulting in 45 oracles for Java.

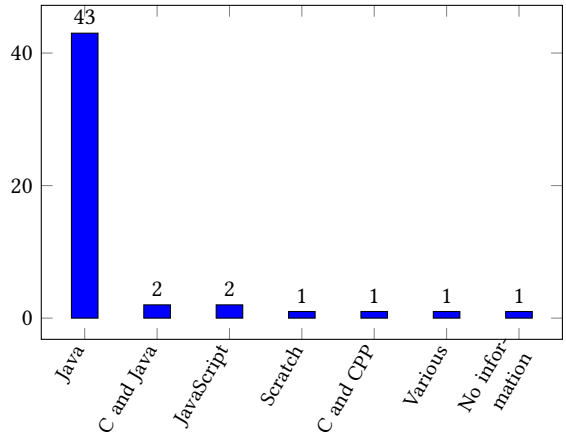

Figure 5: Programming language of the oracles' software systems.

We observed that the traditional definitions of bad smells are associated with object-oriented programming languages. Nevertheless, some studies use languages from others paradigms ([20], [13], [14]), but commonly, most of them make adaptations on the bad smells.

There are only three oracles for C, two for JavaScript, and one for Scratch. One paper mentions an oracle but does not indicate in which programming language it is based. In another paper ([29]), the oracle is constructed with four languages: C++, Java/AspectJ, C#.

RQ3: What are the size of the systems composing the oracles?

Most of the programs used in the oracles are large. Mannan et al. [31], for instance, use a software system with more than 16 million lines of code. In contrast, there are tiny systems used in some oracles, such as the Scratch-based oracle [20].

Some oracles use the same programs. Table 5 shows the most cited programs in descending order and size, considering the number of classes. We observed that projects from Apache Software Foundation are widely used to generate oracles of bad smell. The third column of Table 5 shows the total number of classes indicated by the articles. In many cases, the same software appears in different versions. Each time the two classes range appears in the Table

[10] http://www.ptidej.net/downloads/experiments/qsic09/
[11] http://dx.doi.org/10.6084/m9.figshare.1590962
[12] http://sofa.uqam.ca/paprika/mobilesoft16.phpCodeSmells
[13] http://www.ptidej.net/downloads/experiments/prop-WCRE09
[14] http://www.rcost.unisannio.it/mdipenta/papers/ase2013/
[15] https://dibt.unimol.it/staff/fpalomba/reports/maltesque/
[16] http://soft.vub.ac.be/landfill/
[17] https://dibt.unimol.it/staff/fpalomba/reports/badSmell-analysis/index.html
[18] http://nemo9cby.github.io/icse2017.html
[19] http://www.ptidej.net/download/experiments/ase12/
[20] http://www.ptidej.net/research/decor/
[21] http://www.iro.umontreal.ca/ sahraouh/papers/ASE2010/

Table 4: List of articles that propose oracles available online

Reference	Programs	Bad Smells	Approach	Link
[25]	GanttProject v1.10.2 and Xerces v2.7.0	Blob (God Class)	Manual	[10]
[46]	Apache Ant 1.8.0, aTunes 2.0.0, Eclipse Core 3.6.1, Apache Hive 0.9, Apache Ivy 2.1.0, Apache Lucene 3.6, JVLT 1.3.2, Apache Pig 0.8, Apache Qpid 0.18, Apache Xerces 2.3.0	Long Method, Feature Envy, Blob, Promiscuous Package and Misplaced Class	Tool	[11]
[19]	SoundWaves Podcast 0.112, Terminal Emulator for Android 1.0.70	Internal Getter/Setter (IGS), Member Ignoring Method (MIM) and HashMap Usage (HMU)	Tool	[12]
[24]	Azureus and Eclipse	AbstractClassHasChildren, LargeClass, LargeClassOnly, LongMethod, LongParameterListClass, LowCohesionOnly, ManyAttributes, MessageChainsClass, MethodNoParameter, MultipleInterface, NoInheritance, NoPolymorphism, NotAbstract, NotComplex, OneChildClass, ParentClassProvidesProtected, RareOverriding, TwoInheritance	Tool	[13]
[42]	Apache Ant, Apache Tomcat, jEdit and Android API [framework-opt-telephony, frameworks-base, frameworks-support, sdk, tool-base]	Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob and Feature Envy	Manual	[14]
[45]	ArgoUML, Ant, aTunes, Cassandra, Derby, Eclipse Core, Elastic Search, FreeMind, hadoop, HSQLDB, Hbase, Hibernate, Hive, Incubating, Ivy, Lucene, JEdit, JFreeChart, JBoss, JVlt, jSL, Karaf, Nutch, Pig, Qpid, Sax, Struts, Wicket, Xerces	Class Data Should Be Private (CDSBP), Complex Class, Feature Envy, God Class, Inappropriate Intimacy, Lazy Class, Long Method, Long Parameter List, Message Chain, Middle Man, Refused Bequest, Spahetti code, Speculative Generality	Mixed	[15]
[44]	Apache Ant, Apache Tomcat, jEdit, Android API [framework-opt-telephony, frameworks-base, frameworks-support, sdk, tool-base], Apache[Commons Lang, Cassandra, Commons Codec, Derby], Eclipse Core, Apache James Mime4j, Google Guava, Aardvark, And engine, Apache Commons IO, Apache Commons Logging, Mongo DB	Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob e Feature Envy	Manual	[16]
[41]	ArgoUML, Ant, aTunes, Cassandra, Derby, Eclipse Core, Elastic Search, FreeMind, Hadoop, HSQLDB, Hbase, Hibernate, Hive, Incubating, Ivy, Lucene, JEdit, JHotDraw, JFreeChart, JBoss, JVlt, jSL, Karaf, Nutch, Pig, Qpid, Sax, Struts, Wicket, Xerces	Class Data Should Be Private (CDSBP), Complex Class, Feature Envy, God Class, Inappropriate Intimacy, Lazy Class, Long Method, LPL, Message Chain, Middle Man, Refused Bequest, Spaghetti Code, Speculative Generality	Mixed	[17]
[7]	ActiveMQ, Hadoop and Maven	nullable objects, explicit cast, wrong verbosity level, Duplication with a method's definition, duplication with a local variable's definition, Malformed Output	Tool	[18]
[30]	ArgoUML, Azureus and Xerces	Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife	Tool	[19]
[34]	Xerces	Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife	Manual	[20]
[23]	GanttProject v1.10.2, Xerces v2.7.0, and JHotdraw v7.1	Spaghetti Code, Blob, Functional Decomposition	Tool	[21]

5 for a given program, it means that the total class values diverged between the articles due to the version used.

Table 5: Number of occurrences of most popular programs

Program	Citations	Number of classes
Apache Xerces	15	736
Eclipse	8	1181-17167
Apache Ant	7	846
GanttProject	7	188-245
JFreeChart	7	86-775
ArgoUML	6	777-1415
JHotDraw	6	159-679
Apache Lucene	6	1762-2246
Apache Nutch	6	183-259
Apache Cassandra	5	305-826
jEdit	5	228-520

RQ4: Which bad smells are considered by the oracles?

Most of the studies propose oracles for the bad smells defined by Fowler ([25],[46],[42],[45]).

In some cases, there is more than one definition for the same bad smell; for example, *Code Clone* or *Code Duplicated* refers to *Duplicated Code*. To overcome this problem, we have considered the bad smells and their alternative terms, as represented in Table 6.

Table 6: List of bad smells with their alternative terms

Bad Smell	Alternative Terms
Duplicated Code	*Code Clone*, *Clone Class*, *Clone Code*, *Cloned Code*, *Code Duplication*, *Duplicated Prerequisites*
Large Class	*Blob*, *Big Class*, *Complex Class*, *God Class*
Long Method	*Brain Method*, *God Method*

A total of 126 entries for bad smells have appeared. Table 7 displays the bad smells most considered by the oracles. *Blob* appears in more than 85% of the works, while *Long Method* appears in 46% of the oracles. Note that both bad smells appear in Table 6. Therefore, not necessarily, these terms are used in the papers. Table 4 reports all the bad smells considered by the oracles that are available online.

Table 7: List of most popular bad smells

Bad Smell	Number of papers
Large Class	45
Long Method	26
Feature Envy	22
Data Class	17
Long Parameter List	15
ShotGun Surgery	15
Spaghetti Code	11
Duplicated Code	10
Functional Decomposition	10

RQ5: Which approaches have been used to create the oracles?

We identified three approaches used to create the oracles: manual, tool, and mixed. Manual: consists of inspecting each class in the system. Tool: refer to the use of some tools to determine where the bad smells are in a software system. Mixed: is a combination of the manual and tool approaches. For instance, some classes are disregarded in the manual analysis when the previous data gathered by software metric tools indicate that the classes are well constructed. Figure 6 shows how oracles are distributed between these approaches. The far most common approach to generate oracles is using tool to detect the bad smell. Tools were used to create 33 oracles.

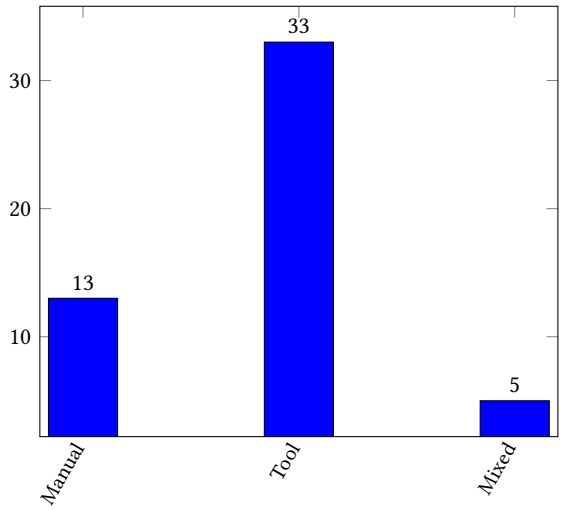

Figure 6: Approaches used to create the oracles

Among the approaches that use tools, many applied the same tool. The most common tool is JDeodorant, used in four of the oracles. The second most used tools are inFusion and PMD, which are used in three studies. CodePro Analytix and iPlasma are used in two works. In the primary studies analyzed in this study, more than 40 tools are explicitly cited.

5 DISCUSSION

Oracles may be used to evaluate strategies and tools for bad smell detection. The programming language most considered in the oracle available in the literature is Java that appears in 45 papers. Therefore, the Java-based strategies have more options for oracles to evaluate their efficiency.

However, most of those oracles have been defined in open-source Java projects. This fact restricts the use of the oracles in studies that consider other contexts. However, just a few oracles are available online. We could find only 12 oracles online. The non-availability of oracles online is an essential lack in the literature because they have two primary impacts: rework since researchers produce their oracles to apply in their works and non-replicability of the studies. A possible reason for researchers produces their oracles is that they do not know the oracle previously proposed in the literature. We believe that the results of this study may serve as a reference in this context.

The oracles available online cover a large number of bad smells and use many software systems. However, we noted that the terminology for bad smells in such oracles varies, most oracles base on tools' results. This fact is paradoxical since the tools need an oracle to have their results validated.

Oracles of bad smells in software systems should be considered a ground truth for those who use them. However, oracles produced by tools do not meet this prerogative, as the results pass through the filter of the implemented detection technique. The use of an oracle produced by a tool to evaluate the results of another tool can only verify if their results are the same. The mixed approach appears as a guarantee that the results given by a tool are true, but the analysis of a specialist only after previous class selection by a tool puts in doubt the presence of bad smells in the remaining classes of the software. If we compare the oracles obtained by the manual approach, it gains more relevance regarding the other two ways to produce them since each software is analyzed in its completeness by the expert's sieve.

We found four manual-based oracles and two oracles generated through a mixed approach available online. The main threat of applying these types of oracle is the subjectivity of human evaluation.

The comparison between the results of detection tools and bad smells oracles may provide insights to determine what are the flaws and inaccuracies of the detection methods. A dissection step may be performed by checking the existence of a pattern of failures among the set of bad smell instances not identified by a tool. The step results should guide an optimized effort to improve the precision of bad smell detection tools.

6 THREATS TO VALIDITY

In this section, we discuss the main threats to the validity of this study and the strategies we took to mitigate them. We took some of these actions to increase the confidence of the study.

We based our work on eight digital libraries. Nevertheless, other bases may exist and contain works proposing bad smell oracles. Nevertheless, we used well-known and reliable digital libraries.

Our work relies on a third-party tool, Publish and Perish, to get the results of Google Scholar in spreadsheets. Although the raw result values are the same in the online format and the one returned by the tool, no verification has been performed to ensure that all data have been considered.

One of the problems faced when carrying out this work was to determine which approaches have been used in the works to generate the oracles since, in some cases, the papers do not explicitly bring such information. For this reason, when no tool has been cited, the work was classified as a manual approach. Otherwise, the approach was classified as mixed when the papers refer to any tool and indicate that the data were manually treated. The approach was classified as a "tool" when no data treatment was explicit in the text.

There are primary studies that have used proprietary projects and, therefore, their authors avoid exposing data that are often sensitive. In these cases, we rely on the authors' terminology to determine the size of the software since we do not have access to the source code.

7 CONCLUSION

This work presented a Systematic Literature Review to identify oracles for bad smells. The primary motivation for this study is the fact that bad smells have been widely studied, and many studies rely on bad smell oracles obtained by tools that have not yet been properly proven to be precise. None of the works we found in this SLR performs a literature review on oracle of bad smells. Therefore, to the best of our knowledge, this is the first Systematic Literature Review to identify oracles of bad smells.

We considered eight digital libraries to perform the search process. The searches in the digital libraries returned 4,948 primary studies. However, a small number of these studies define oracles of bad smells, namely 51, being only 12 available in some online repository. Researchers may benefit from the results of this work to produce new studies, cross-refer data and evaluate their bad smell detection proposals. The bad smells defined by Fowler et al. [18] are the most used in oracles.

The oracles of bad smells available online have the main following characteristics: they involve at maximum 29 software systems, varying from 86 to 17,167 classes; the software systems they consider are mainly developed in Java; most of them rely on results provided by tools; they usually verify Large Class instances; and provide their results in a spreadsheet.

The results of this study show that there is still a gap to be filled in the literature, for instance:

(1) just a few oracles identify the methods where the bad smell is located, and
(2) most of the oracles are indeed defined employing tools, which is a relevant threat to their validity.

In future works, it is essential to generate oracles that overcome these fragilities. Also, an oracle consisting of software systems not considered previously may be of value.

REFERENCES

[1] R. Abílio, J. Padilha, E. Figueiredo, and H. Costa. 2015. Detecting Code Smells in Software Product Lines – An Exploratory Study. In *2015 12th International Conference on Information Technology - New Generations*. 433–438. https://doi.org/10.1109/ITNG.2015.76

[2] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro. 2015. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 261–269. https://doi.org/10.1109/ISSRE.2015.7381819

[3] M. Aniche, G. Bavota, C. Treude, A. V. Deursen, and M. A. Gerosa. 2016. A Validated Set of Smells in Model-View-Controller Architectures. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 233–243. https://doi.org/10.1109/ICSME.2016.12

[4] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and Experimenting Machine Learning Techniques for Code Smell Detection. *Empirical Softw. Engg.* 21, 3 (June 2016), 1143–1191. https://doi.org/10.1007/s10664-015-9378-4

[5] Lisa Börjesson. 2016. Research Outside Academia? An Analysis of Resources in Extra-Academic Report Writing *(ASIST '16)*. American Society for Information Science, USA, Article 36, 10 pages.

[6] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis: Refactoring Software, Architecture and Projects in Crisis* (1. auflage ed.). John Wiley & Sons.

[7] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing and Detecting Anti-patterns in the Logging Code. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 71–81. https://doi.org/10.1109/ICSE.2017.15

[8] Z. Chen, L. Chen, W. Ma, and B. Xu. 2016. Detecting Code Smells in Python Programs. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. 18–23. https://doi.org/10.1109/SATE.2016.10

[9] J. P. d. Reis, F. Brito e Abreu, and G. d. F. Carneiro. 2016. Code Smells Incidence: Does It Depend on the Application Domain?. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*. 172–177. https://doi.org/10.1109/QUATIC.2016.044

[10] P. Danphitsanuphan and T. Suwantada. 2012. Code Smell Detecting Tool and Code Smell-Structure Bug Relationship. In *2012 Spring Congress on Engineering and Technology*. 1–5. https://doi.org/10.1109/SCET.2012.6342082

[11] K. Dhambri, H. Sahraoui, and P. Poulin. 2008. Visual Detection of Design Anomalies. In *2008 12th European Conference on Software Maintenance and Reengineering*. 279–283. https://doi.org/10.1109/CSMR.2008.4493326

[12] Raimar Falke, Pierre Frenzel, and Rainer Koschke. 2008. Empirical Evaluation of Clone Detection Using Syntax Suffix Trees. *Empirical Softw. Engg.* 13, 6 (Dec. 2008), 601–643. https://doi.org/10.1007/s10664-008-9073-9

[13] A. M. Fard and A. Mesbah. 2013. JSNOSE: Detecting JavaScript Code Smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 116–125. https://doi.org/10.1109/SCAM.2013.6648192

[14] W. Fenske, S. Schulze, D. Meyer, and G. Saake. 2015. When code smells twice as much: Metric-based detection of variability-aware code smells. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 171–180. https://doi.org/10.1109/SCAM.2015.7335413

[15] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A Review-based Comparative Study of Bad Smell Detection Tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering* (Limerick, Ireland) *(EASE '16)*. ACM, New York, NY, USA, Article 18, 12 pages. https://doi.org/10.1145/2915970.2915984

[16] F. A. Fontana, V. Ferme, and S. Spinelli. 2012. Investigating the impact of code smells debt on quality code evaluation. In *2012 Third International Workshop on Managing Technical Debt (MTD)*. 15–22. https://doi.org/10.1109/MTD.2012.6225993

[17] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda. 2015. Towards a prioritization of code debt: A code smell Intensity Index. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. 16–24. https://doi.org/10.1109/MTD.2015.7332620

[18] Marting Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

[19] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2016. An Empirical Study of the Performance Impacts of Android Code Smells. In *Proceedings of the International Conference on Mobile Software Engineering and Systems* (Austin, Texas) *(MOBILESoft '16)*. ACM, New York, NY, USA, 59–69. https://doi.org/10.1145/2897073.2897100

[20] F. Hermans and E. Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10. https://doi.org/10.1109/ICPC.2016.7503706

[21] A. Kaur, K. Kaur, and S. Jain. 2016. Predicting software change-proneness with code smells and class imbalance learning. In *2016 International Conference on*

Advances in Computing, Communications and Informatics (ICACCI). 746–754. https://doi.org/10.1109/ICACCI.2016.7732136

[22] M. Kessentini and A. Ouni. 2017. Detecting Android Smells Using Multi-Objective Genetic Programming. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 122–132. https://doi.org/10.1109/MOBILESoft.2017.29

[23] Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. 2010. Deviance from Perfection is a Better Criterion Than Closeness to Evil when Identifying Risky Code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. ACM, New York, NY, USA, 113–122. https://doi.org/10.1145/1858996.1859015

[24] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. 2009. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. *Proceedings - Working Conference on Reverse Engineering, WCRE*, 75–84. https://doi.org/10.1109/WCRE.2009.28

[25] Foutse Khomh, Stéphane Vaucher, Yann gaël Guéhéneuc, and Houari Sahraoui. [n.d.]. A Bayesian Approach for the Detection of Code and Design Smells.

[26] B. Kitchenham and S Charters. 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering.

[27] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. 2005. *Object-Oriented Metrics in Practice*. Springer-Verlag, Berlin, Heidelberg.

[28] Thierry Lavoie and Ettore Merlo. 2011. Automated Type-3 Clone Oracle Using Levenshtein Metric. In *Proceedings of the 5th International Workshop on Software Clones* (Waikiki, Honolulu, HI, USA) *(IWSC '11)*. ACM, New York, NY, USA, 34–40. https://doi.org/10.1145/1985404.1985411

[29] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa. 2012. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In *2012 16th European Conference on Software Maintenance and Reengineering*. 277–286. https://doi.org/10.1109/CSMR.2012.35

[30] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabane, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. 2012. Support vector machines for anti-pattern detection. *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (2012), 278–281.

[31] Umme Ayda Mannan, Iftekhar Ahmed, Rana Abdullah M. Almurshed, Danny Dig, and Carlos Jensen. 2016. Understanding Code Smells in Android Applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems* (Austin, Texas) *(MOBILESoft '16)*. ACM, New York, NY, USA, 225–234. https://doi.org/10.1145/2897073.2897094

[32] M. V. Mantyla, J. Vanhanen, and C. Lassenius. 2004. Bad smells - humans as code critics. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 399–408. https://doi.org/10.1109/ICSM.2004.1357825

[33] R. Marinescu. 2001. Detecting design flaws via metrics in object-oriented systems. In *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39*. 173–182.

[34] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Softw. Eng.* 36, 1 (Jan. 2010), 20–36. https://doi.org/10.1109/TSE.2009.50

[35] Willian Oizumi, Alessandro Garcia, Leonardo da Silva Sousa, Bruno Cafeo, and Yixue Zhao. 2016. Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 440–451. https://doi.org/10.1145/2884781.2884868

[36] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. v. Staa. 2014. When Code-Anomaly Agglomerations Represent Architectural Problems? An Exploratory Study. In *2014 Brazilian Symposium on Software Engineering*. 91–100. https://doi.org/10.1109/SBES.2014.18

[37] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. 2009. The evolution and impact of code smells: A case study of two open source systems. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. 390–400. https://doi.org/10.1109/ESEM.2009.5314231

[38] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study. *ACM Trans. Softw. Eng. Methodol.* 25, 3, Article 23 (June 2016), 53 pages. https://doi.org/10.1145/2932631

[39] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Mohamed Salah Hamdi. 2015. Improving multi-objective code-smells correction using development history. *Journal of Systems and Software* 105 (2015), 18 – 39. https://doi.org/10.1016/j.jss.2015.03.040

[40] F. Palomba. 2015. Textual Analysis for Code Smell Detection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 769–771. https://doi.org/10.1109/ICSE.2015.244

[41] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (01 Jun 2018), 1188–1221. https://doi.org/10.1007/s10664-017-9535-z

[42] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting bad smells in source code using change history information. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013), 268–278.

[43] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. 2015. Mining Version Histories for Detecting Code Smells. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 462–489. https://doi.org/10.1109/TSE.2014.2372760

[44] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2015. Landfill: An Open Dataset of Code Smells with Public Evaluation. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (Florence, Italy) *(MSR '15)*. IEEE Press, Piscataway, NJ, USA, 482–485. http://dl.acm.org/citation.cfm?id=2820518.2820593

[45] Fabio Palomba, Rocco Oliveto, and Andrea De Lucia. 2017. Investigating code smell co-occurrences using association rule learning: A replicated study. *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)* (2017), 8–13.

[46] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. 2016. A Textual-Based Technique for Smell Detection. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10. https://doi.org/10.1109/icpc.2016.7503704 Exported from https://app.dimensions.ai on 2019/02/25.

[47] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. 2018. The Scent of a Smell: An Extensive Comparison Between Textual and Structural Smells. *IEEE Transactions on Software Engineering* 44, 10 (Oct 2018), 977–1000. https://doi.org/10.1109/TSE.2017.2752171

[48] R. Peters and A. Zaidman. 2012. Evaluating the Lifespan of Code Smells using Software Repository Mining. In *2012 16th European Conference on Software Maintenance and Reengineering*. 411–416. https://doi.org/10.1109/CSMR.2012.79

[49] Dilan Sahin. 2016. A Multi-Level Framework for the Detection, Prioritization and Testing of Software Design Defects.

[50] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158 – 173. https://doi.org/10.1016/j.jss.2017.12.034

[51] K. Sirikul and C. Soomlek. 2016. Automated detection of code smells caused by null checking conditions in Java programs. In *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. 1–7. https://doi.org/10.1109/JCSSE.2016.7748884

[52] Elder V. P. Sobrinho, A. Lucia, and M. Maia. 2018. A systematic literature review on bad smells — 5 W's: which, when, what, who, where. *IEEE Transactions on Software Engineering* (2018), 1–1.

[53] Gábor Szőke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2014. A Case Study of Refactoring Large-Scale Industrial Systems to Efficiently Improve Source Code Quality. In *Computational Science and Its Applications – ICCSA 2014*, Beniamino Murgante, Sanjay Misra, Ana Maria A. C. Rocha, Carmelo Torre, Jorge Gustavo Rocha, Maria Irene Falcão, David Taniar, Bernady O. Apduhan, and Osvaldo Gervasi (Eds.). Springer International Publishing, Cham, 524–540.

[54] Gustavo Vale, Danyllo Albuquerque, Eduardo Figueiredo, and Alessandro Garcia. 2015. Defining Metric Thresholds for Software Product Lines: A Comparative Study. In *Proceedings of the 19th International Conference on Software Product Line* (Nashville, Tennessee) *(SPLC '15)*. ACM, New York, NY, USA, 176–185. https://doi.org/10.1145/2791060.2791078

[55] G. A. D. Vale and E. M. L. Figueiredo. 2015. A Method to Derive Metric Thresholds for Software Product Lines. In *2015 29th Brazilian Symposium on Software Engineering*. 110–119. https://doi.org/10.1109/SBES.2015.9

[56] B. C. Wagey, B. Hendradjaya, and M. S. Mardiyanto. 2015. A proposal of software maintainability model using code smell measurement. In *2015 International Conference on Data and Software Engineering (ICoDSE)*. 25–30. https://doi.org/10.1109/ICODSE.2015.7436966

[57] Claes Wohlin. 2014. 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* (London, England, United Kingdom) *(EASE '14)*. ACM, New York, NY, USA, Article 38, 10 pages. https://doi.org/10.1145/2601248.2601268

[58] Aiko Yamashita. 2014. Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative and Qualitative Data. *Empirical Softw. Engg.* 19, 4 (Aug. 2014), 1111–1143. https://doi.org/10.1007/s10664-013-9250-3

[59] Aiko Yamashita and Leon Moonen. 2013. To what extent can maintenance problems be predicted by code smell detection? – An empirical study. *Information and Software Technology* 55, 12 (2013), 2223 – 2242. https://doi.org/10.1016/j.infsof.2013.08.002

[60] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter. 2015. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 121–130. https://doi.org/10.1109/ICSM.2015.7332458

[61] Min Zhang, Tracy Hall, and Nathan Baddoo. 2011. Code Bad Smells: A Review of Current Knowledge. *J. Softw. Maint. Evol.* 23, 3 (April 2011), 179–202. https:

//doi.org/10.1002/smr.521

[62] X. Zhao, X. Xuan, and S. Li. 2015. An Empirical Study of Long Method and God Method in Industrial Projects. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 109–114. https://doi.org/10.1109/ASEW.2015.15